Report on Kalman Filter applied on Neural Network for Finance

November 19, 2025

Mathys VINATIER - GitHub Project page
Supervisor:
Pr Kim Tae-Wan

Mathys VINATIER

Contents

Chapter 10

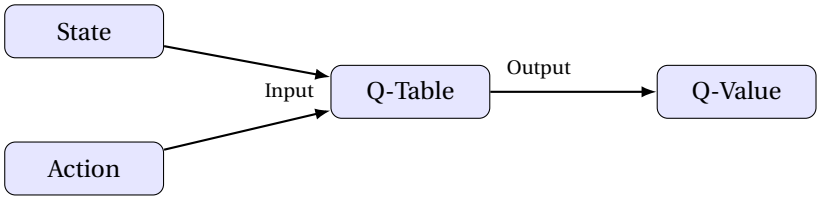
Week 10

Contents

10.1 Reminder on Greedy Policy and QLearning

Q-Learning is a **model-free, off-policy, value-based reinforcement learning algorithm** that trains an action-value function (Q-function) to find the optimal policy indirectly. The Q-function estimates the expected cumulative reward of taking a certain action in a given state and following the best policy thereafter.

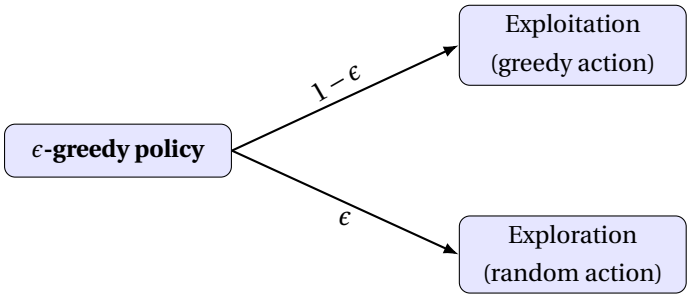
- **Q-table:** Internally, Q-Learning uses a Q-table storing values for each state-action pair. Initially, all values are zero, and the table is updated iteratively during training.



- **Epsilon-Greedy Strategy:** To balance exploration and exploitation, actions are chosen using an epsilon-greedy policy:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

  - With probability $1-\epsilon$, exploit by selecting the action with the highest Q-value.
  - With probability ϵ, explore by selecting a random action.
  - ϵ decays over time to shift from exploration to exploitation.



- **TD Update:** At each step, the Q-value for the current state-action pair $Q(s_t, a_t)$ is updated based on the immediate reward r_{t+1} plus the discounted maximum Q-value of the next state s_{t+1}:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right]$$

- **Off-policy Learning:** The policy used to select actions (epsilon-greedy) differs from the policy used to compute the target update (greedy). This distinction allows Q-Learning to learn optimal policies even when exploring randomly.

After enough training episodes, the Q-table converges to the optimal Q-function, which directly yields the optimal policy by choosing actions with the highest Q-values in each state.

10.2 Implementation Progress and Experiments

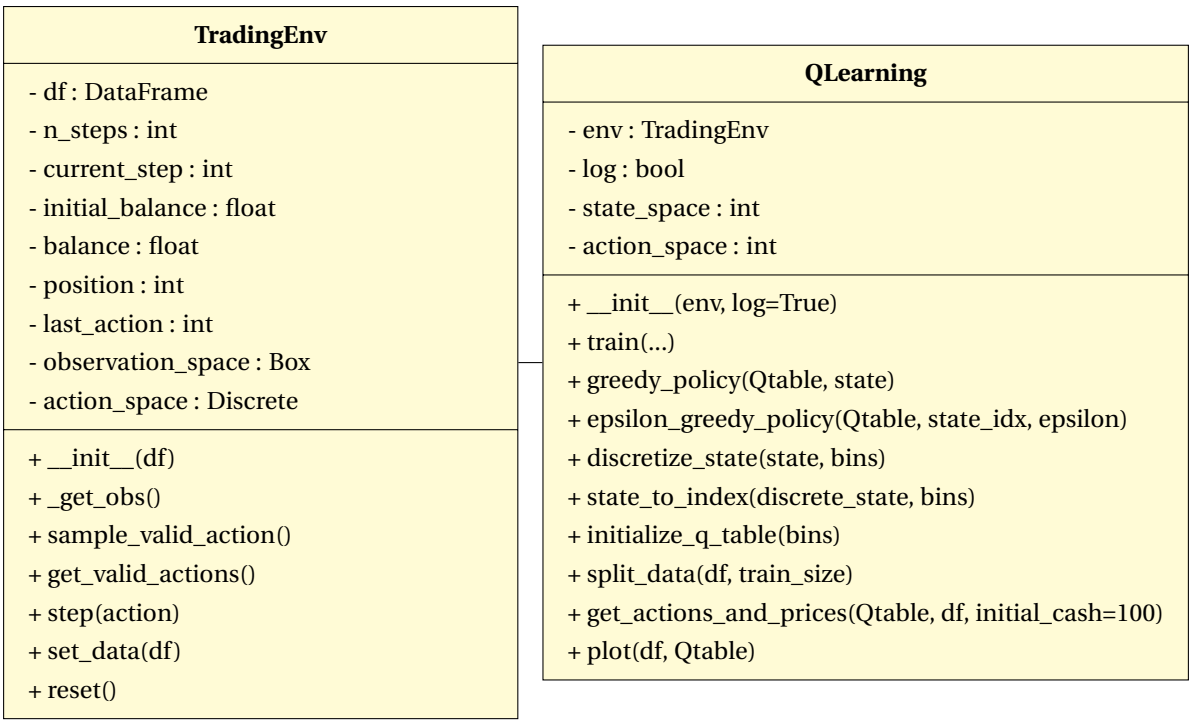
This week was dedicated to the initial implementation and experimentation phase of the project, focusing primarily on the Proximal Policy Optimization (PPO) basis algorithms. Using the Hugging Face classes, we developed a prototype with greedy policy agent and integrated it into a custom trading environment inspired by the concepts discussed in previous weeks.

10.2.1 Environment Setup

The custom environment was designed to simulate stock market trading dynamics with the following key elements :

- **State representation** including historical price data, technical indicators (moving averages, RSI) and Kalman filtered signals.
- **Action space** consisting of discrete trading actions: Buy, Hold or Sell.
- **Reward structure** based on portfolio value change and risk-adjusted metrics to incentivize both profit and stability.
- **Episode design** with fixed-length trading periods to allow episodic evaluation of agent performance.

The environment was implemented adhering to OpenAI Gym interfaces, allowing smooth integration with standard RL training pipelines. Here is the UML of the created environement :



Algorithm 1: Trading Environment Behavior

Input: Environment env, number of episodes N, exploration rate ϵ

Output: Episode value and rewards

for $episode \leftarrow 1$ **to** N
- $s \leftarrow env.reset()$;
- total_reward $\leftarrow 0$;
- **while** *True*
  - Get valid actions $A_{valid} \leftarrow env.get\_valid\_actions()$;
  - Select $a \leftarrow$ random choice from A_{valid};
  - Execute $(s', r, done, info) \leftarrow env.step(a)$;
  - total_reward $\leftarrow$ total_reward $+ r$;
  - Log $(episode, s, a, r, info["portfolio\_value"])$;
  - $s \leftarrow s'$;
  - **if** $done$ **then**
    - break
- Print episode summary : total_reward, final portfolio value;

Algorithm 2: Q-Learning Training

Input: Environment env, data frame df, training size $train\_size$, episode N

Output: Trained Q-table Q

Split data: $df_{train} \leftarrow train\_size$ of df;

Calculate bins for discretization : $bins \leftarrow$ compute bins from df_{train};

Initialize Q-table

for $episode \leftarrow 1$ **to** N
- Exploration rate : $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$;
- Reset environment : $state_{cont} \leftarrow env.reset()$;
- Discretize initial state : $state_{disc} \leftarrow discretize(state_{cont}, bins)$;
- Convert to index : $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$;
- **for** $step \leftarrow 1$ **to** max_steps
  - Choose action a using epsilon-greedy policy :

$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1-\epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

  - Take action : $(next_{state_{cont}}, r, done, info) \leftarrow env.step(a)$;
  - Discretize next state : $next_{state_{disc}} \leftarrow discretize(next_{state_{cont}}, bins)$;
  - Convert to index : $next_{state_{idx}} \leftarrow state\_to\_index(next_{state_{disc}}, bins)$;
  - Update Q-value :$Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha\left(r + \gamma \max_{a'} Q[next_{state_{idx}}, a'] - Q[state_{idx}, a]\right)$;
  - $state_{idx} \leftarrow next_{state_{idx}}$;
  - **if** $done$ **then**
    - **break**;

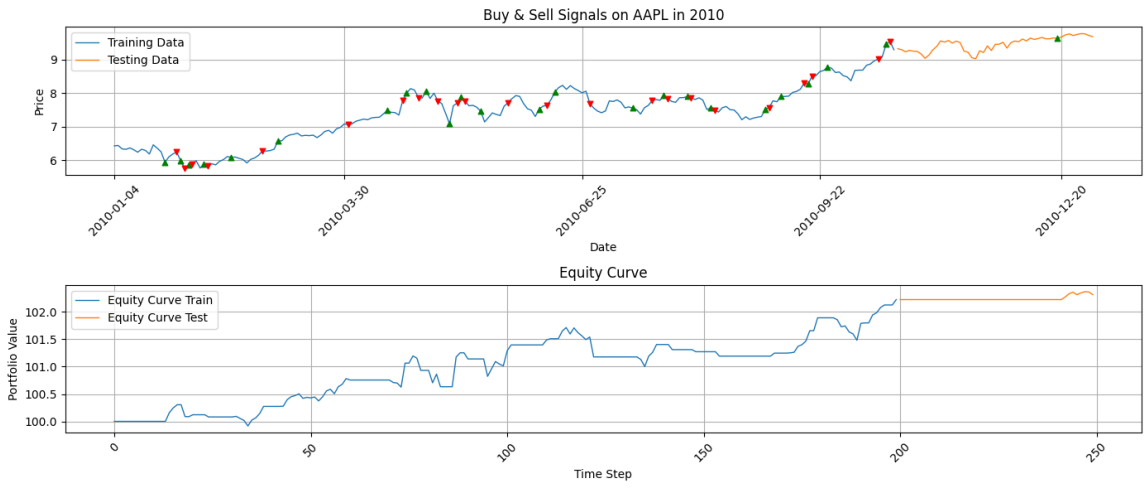
return Q

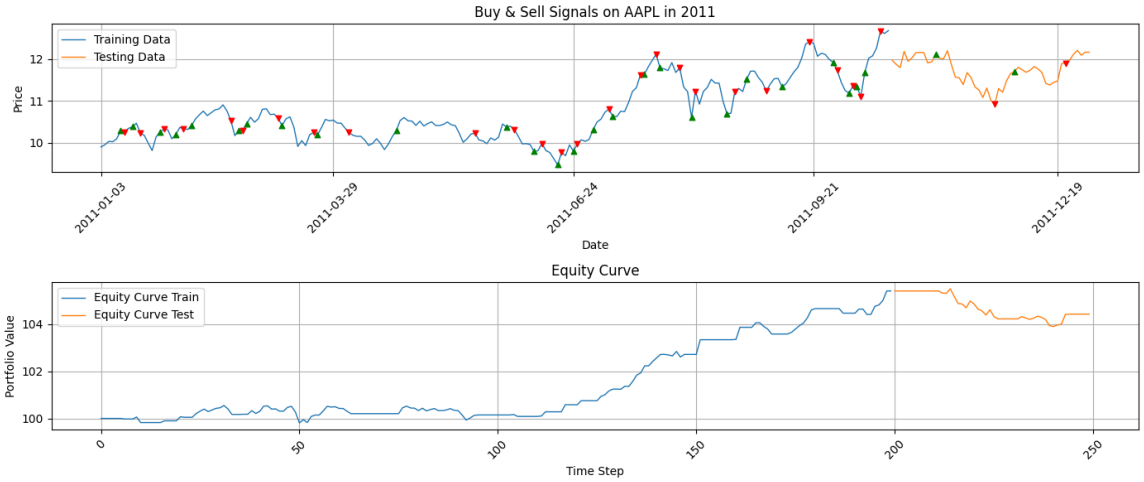
In the context of Q-learning for trading environments, bins are used to discretize the continuous state space into a finite number of discrete states. This discretization is essential because the Q-table requires a finite and manageable number of states to store and update the expected rewards for each state-action pair. Continuous variables, such as stock prices or technical indicators, have infinite possible values, which makes it impractical to represent them directly in a Q-table. To address this, we divide the range of each continuous feature into a fixed number of intervals called bins. Each bin represents a segment of the feature's value range, allowing us to map any continuous observation into a discrete category. For implementation, we analyze the training dataset and create evenly spaced bins (10 bins) for each feature by computing linearly spaced intervals between the minimum and maximum values of that feature. Then, during training or evaluation, the observed continuous state is converted into a discrete state by determining which bin each feature value falls into. This discretized state is subsequently converted into a unique index to access and update the Q-table. The use of bins thus enables effective Q-learning in continuous state environments by simplifying the state representation while preserving essential information.

10.2.2 Preliminary Results for Q-Learning Greedy Policy

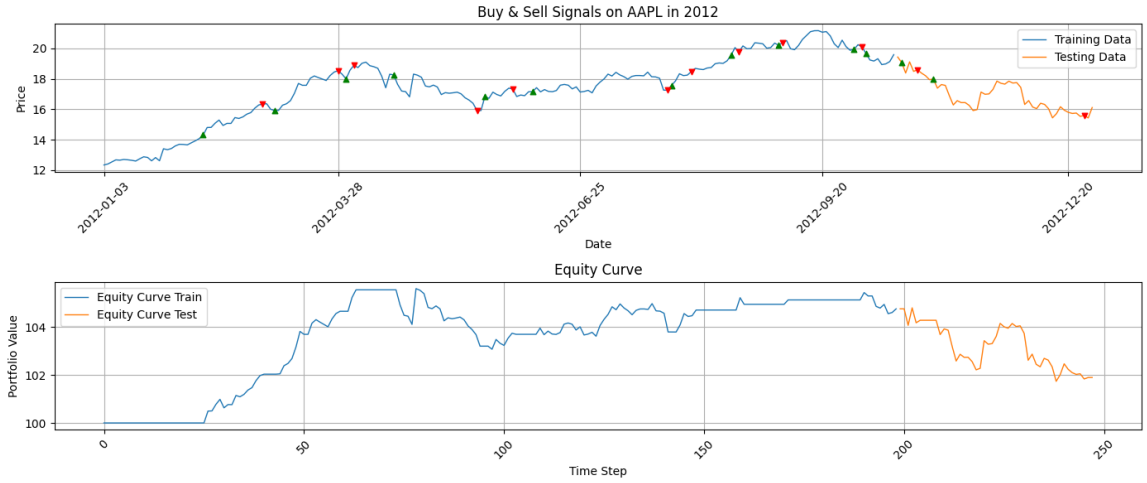
Initial experiments aimed to evaluate the training behavior and reward progression of the Q-Learning agent following a greedy policy. Results were compared over two time horizons : a shorter training period of 100 episodes and an extended training of 1000 episodes.

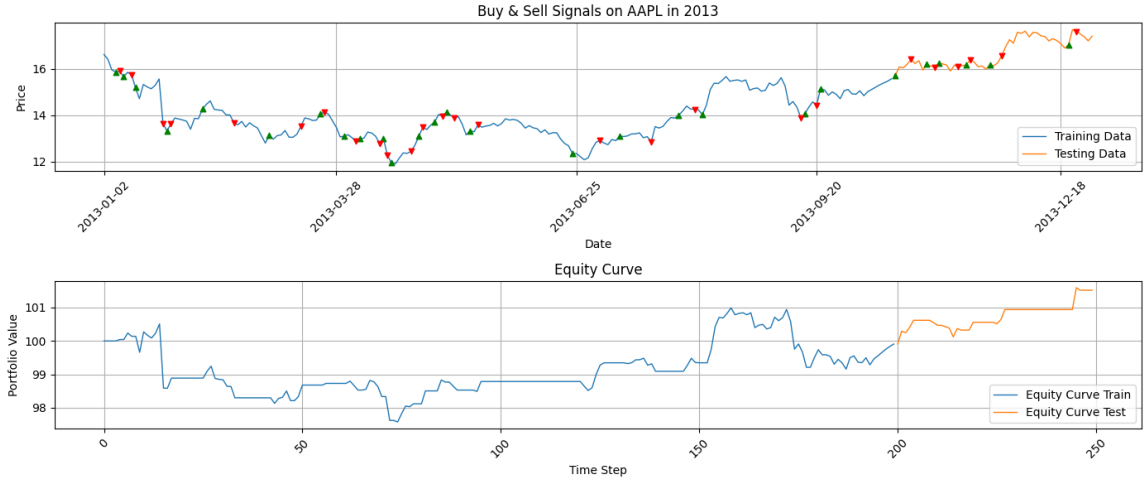
100 episodes

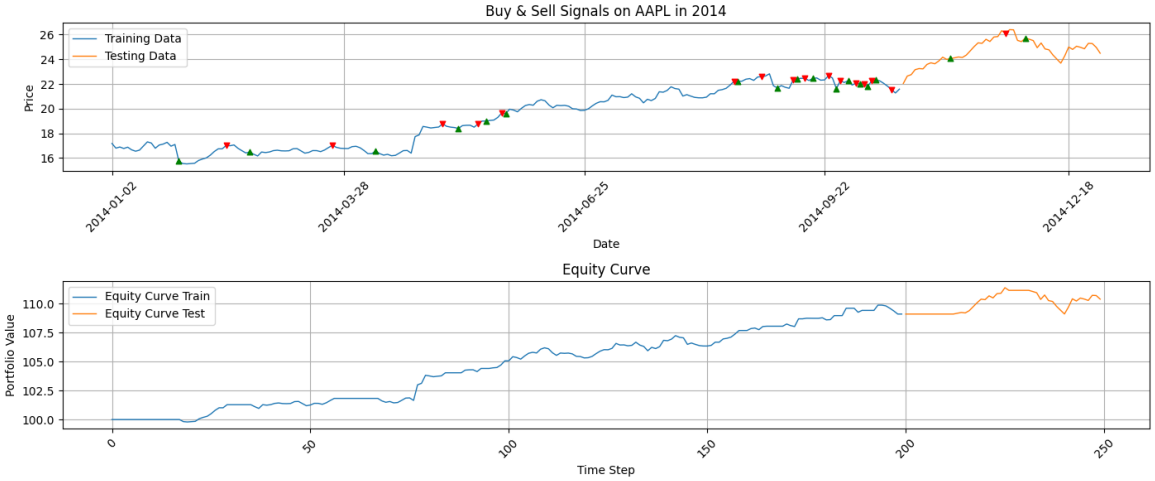


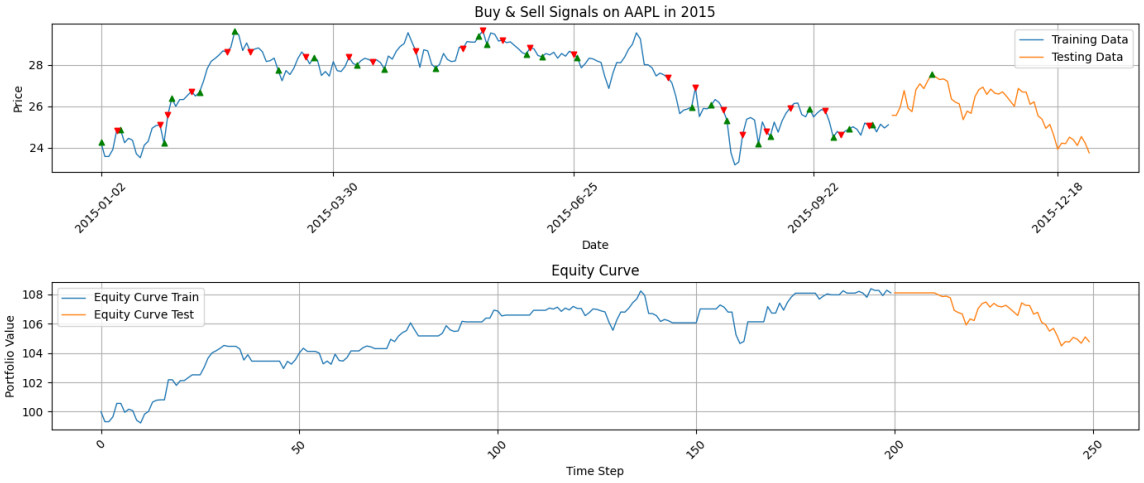

Buy & Sell Signals on AAPL in 2011
Training Data
Testing Data
Price
12
11
10
2011-01-03
2011-03-29
2011-06-24
2011-09-21
2011-12-19
Date

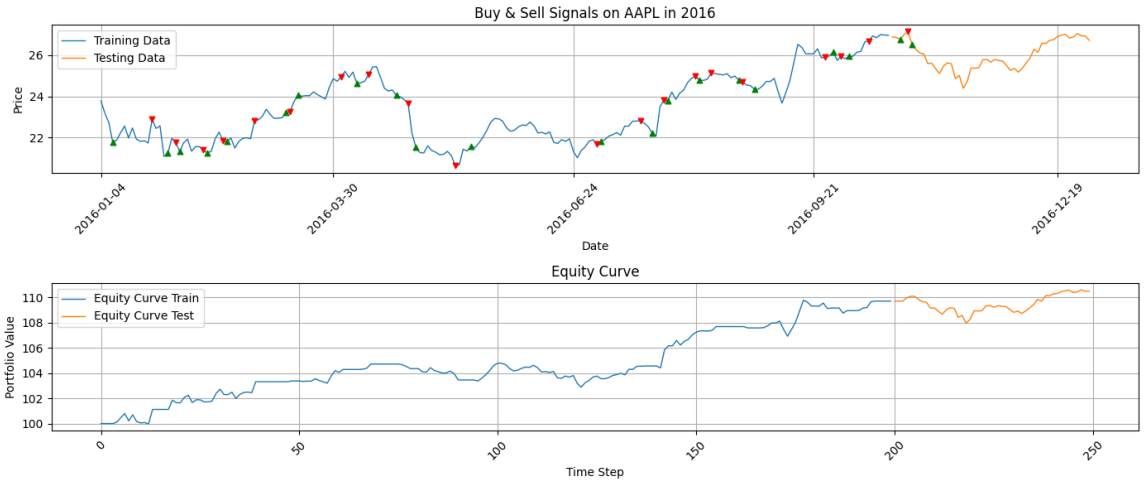
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
104
102
100
0
50
100
150
200
250
Time Step

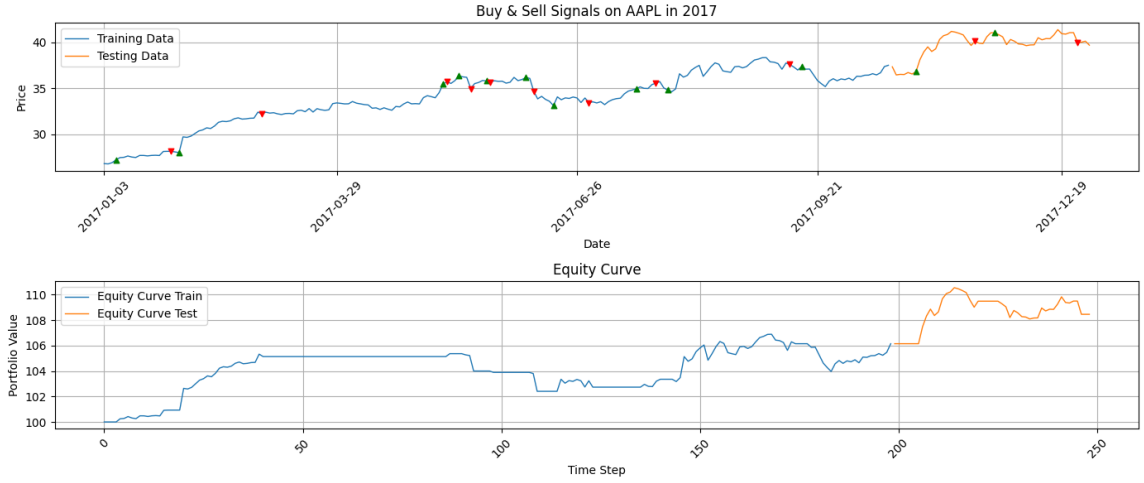

Buy & Sell Signals on AAPL in 2012
Training Data
Testing Data
Price
Date
2012-01-03
2012-03-28
2012-06-25
2012-09-20
2012-12-20
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

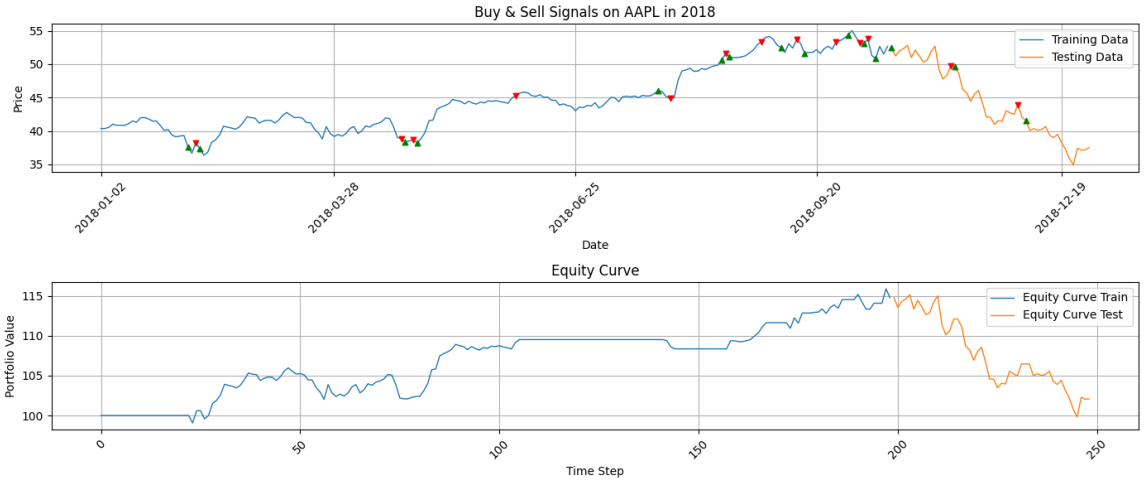

Buy & Sell Signals on AAPL in 2013
Training Data
Testing Data
Price
Date
2013-01-02
2013-03-28
2013-06-25
2013-09-20
2013-12-18
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step


Buy & Sell Signals on AAPL in 2014
Training Data
Testing Data
Price
Date
2014-01-02
2014-03-28
2014-06-25
2014-09-22
2014-12-18
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

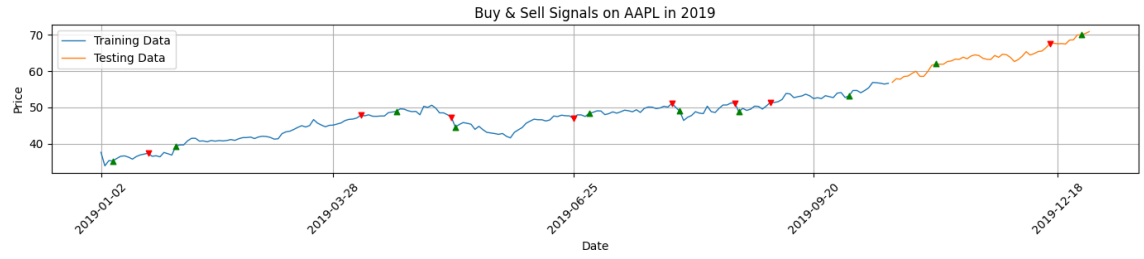

Buy & Sell Signals on AAPL in 2015
Training Data
Testing Data
Price
Date
2015-01-02
2015-03-30
2015-06-25
2015-09-22
2015-12-18
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

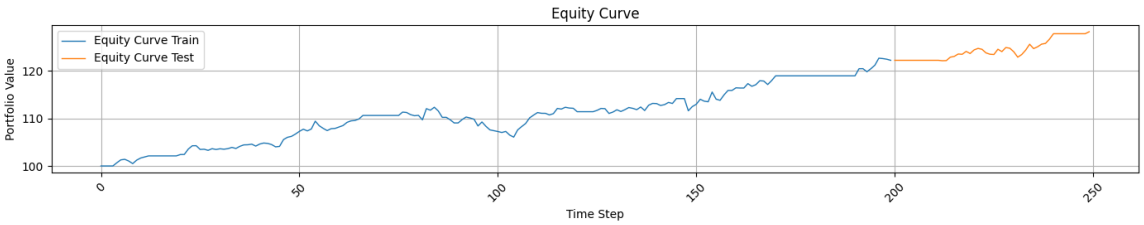

Buy & Sell Signals on AAPL in 2016
Training Data
Testing Data
Price
Date
2016-01-04
2016-03-30
2016-06-24
2016-09-21
2016-12-19
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

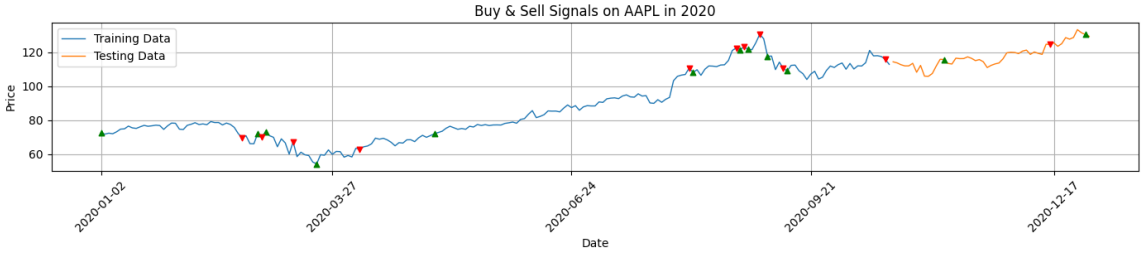

Buy & Sell Signals on AAPL in 2017
Training Data
Testing Data
Price
Date
2017-01-03
2017-03-29
2017-06-26
2017-09-21
2017-12-19
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

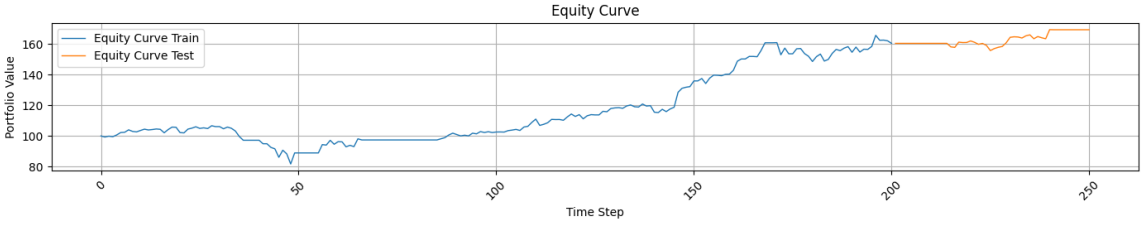

Buy & Sell Signals on AAPL in 2018
Training Data
Testing Data
Price
Date
2018-01-02
2018-03-28
2018-06-25
2018-09-20
2018-12-19

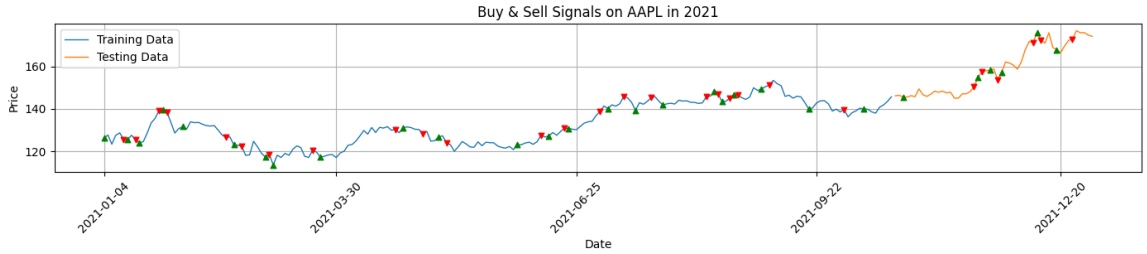
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

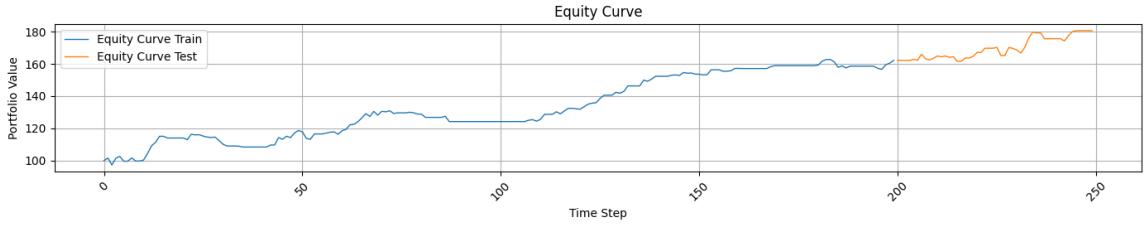

Buy & Sell Signals on AAPL in 2019
Training Data
Testing Data
Price
Date
2019-01-02
2019-03-28
2019-06-25
2019-09-20
2019-12-18

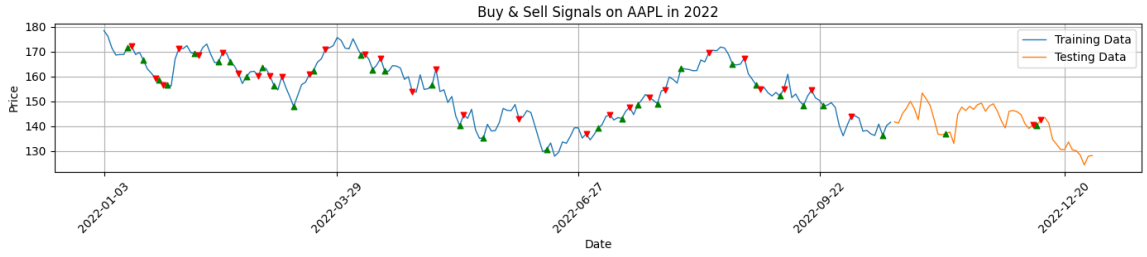

Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

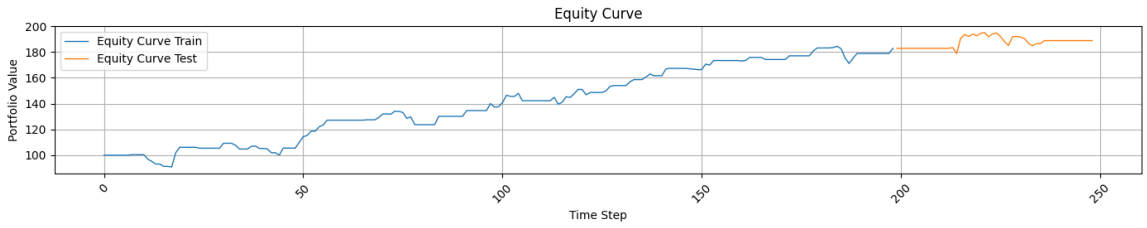

Buy & Sell Signals on AAPL in 2020
Training Data
Testing Data
Price
Date
2020-01-02
2020-03-27
2020-06-24
2020-09-21
2020-12-17


Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

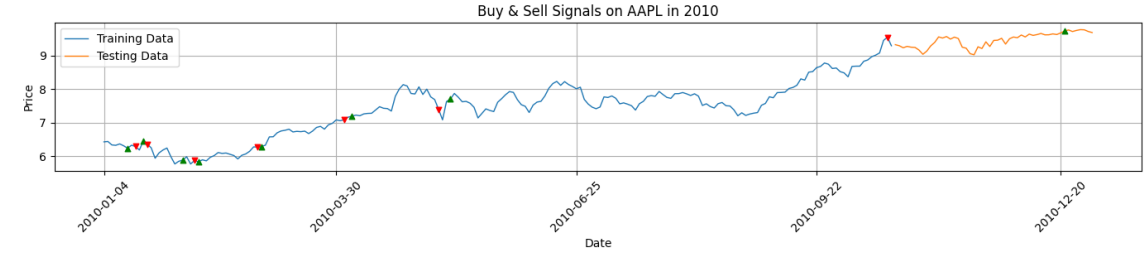


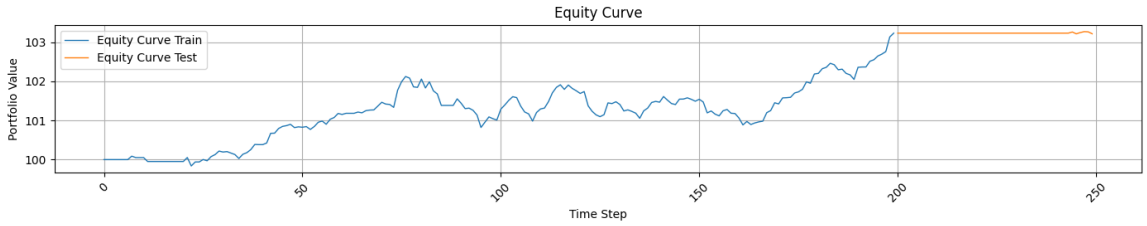


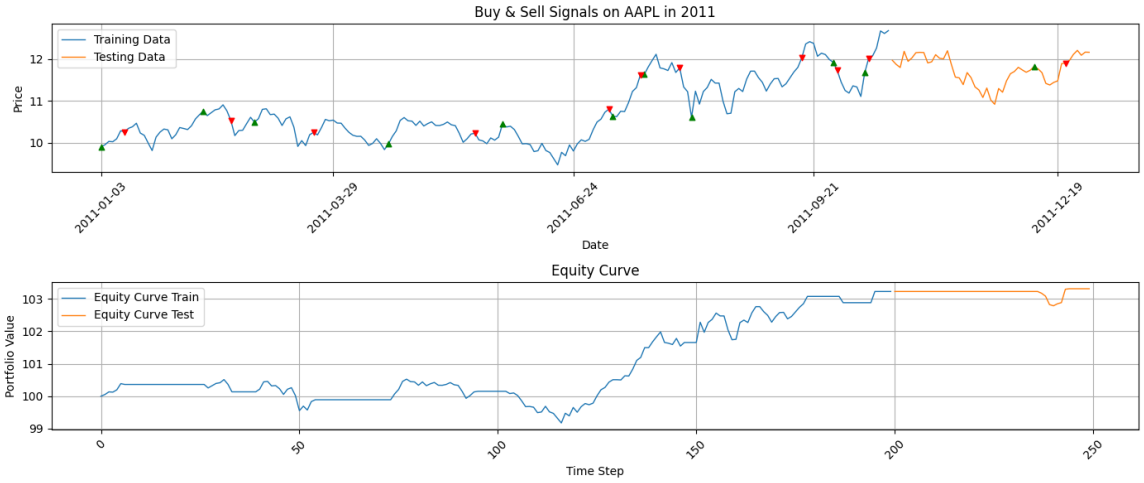


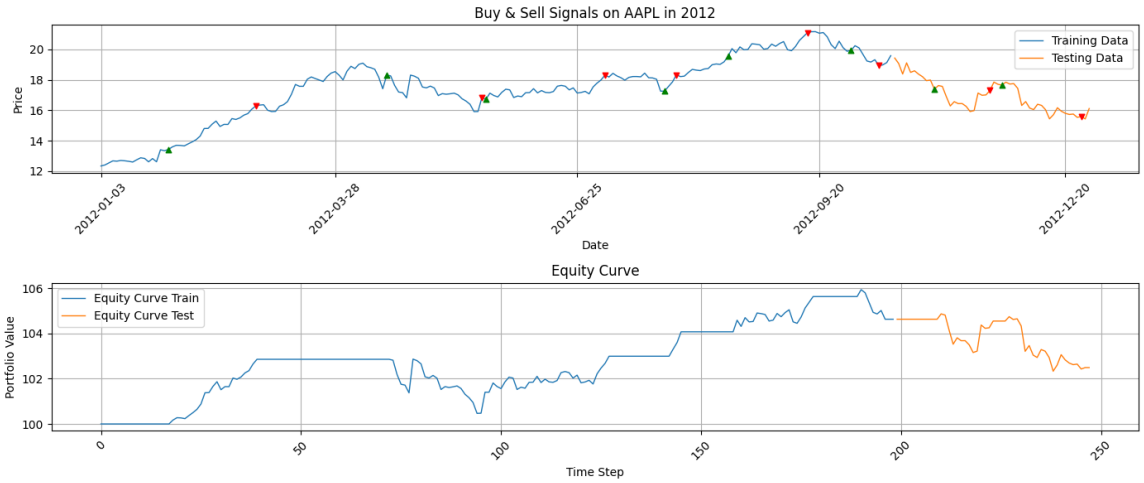


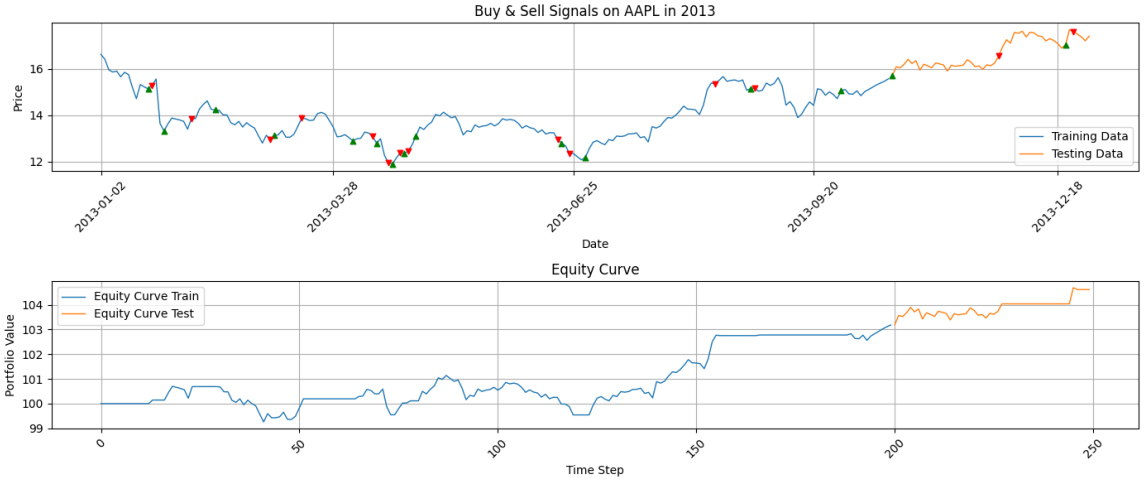
1000 episodes

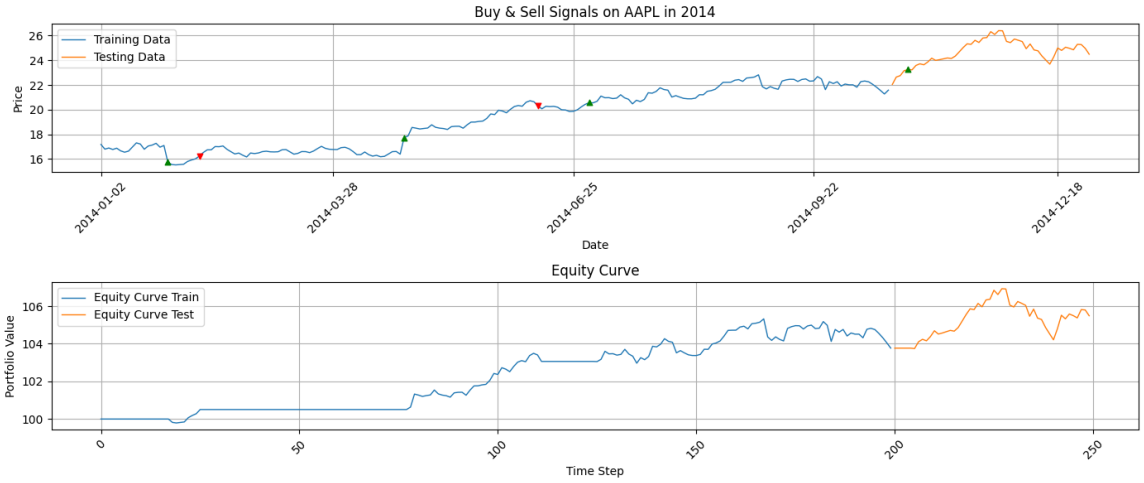




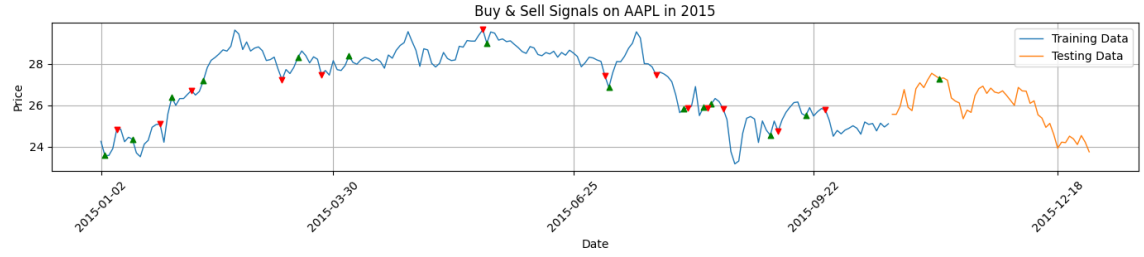

Buy & Sell Signals on AAPL in 2011
Training Data
Testing Data
Price
Date
2011-01-03
2011-03-29
2011-06-24
2011-09-21
2011-12-19
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

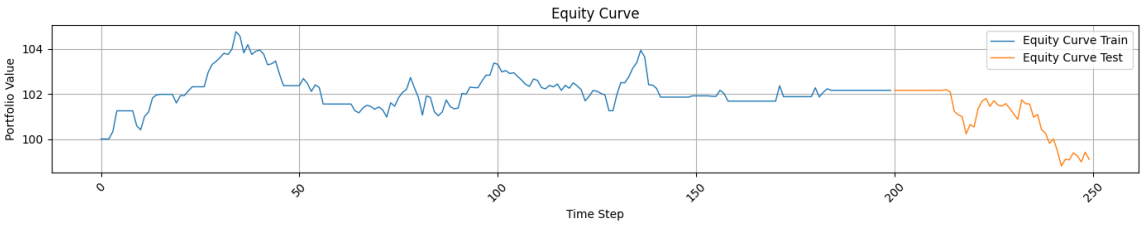

Buy & Sell Signals on AAPL in 2012
Training Data
Testing Data
Price
Date
2012-01-03
2012-03-28
2012-06-25
2012-09-20
2012-12-20
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

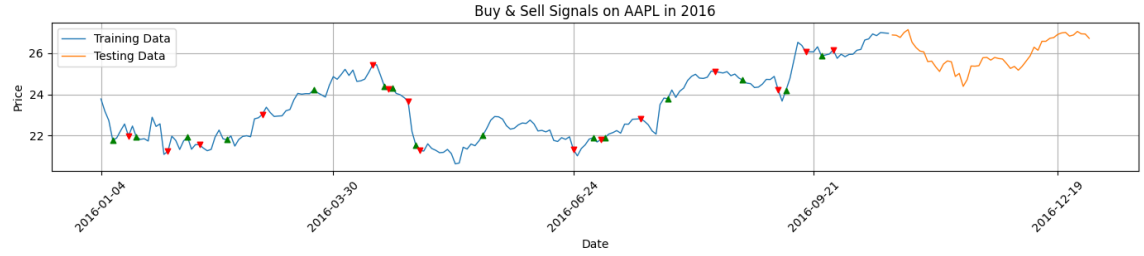

Buy & Sell Signals on AAPL in 2013
Training Data
Testing Data
Price
Date
2013-01-02
2013-03-28
2013-06-25
2013-09-20
2013-12-18
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

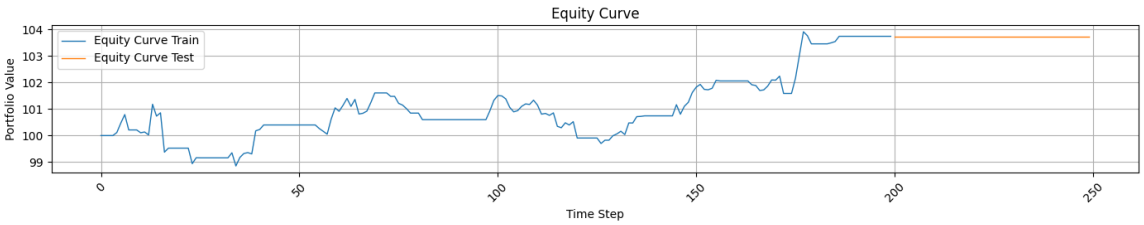

Buy & Sell Signals on AAPL in 2014
Training Data
Testing Data
Price
Date
2014-01-02
2014-03-28
2014-06-25
2014-09-22
2014-12-18

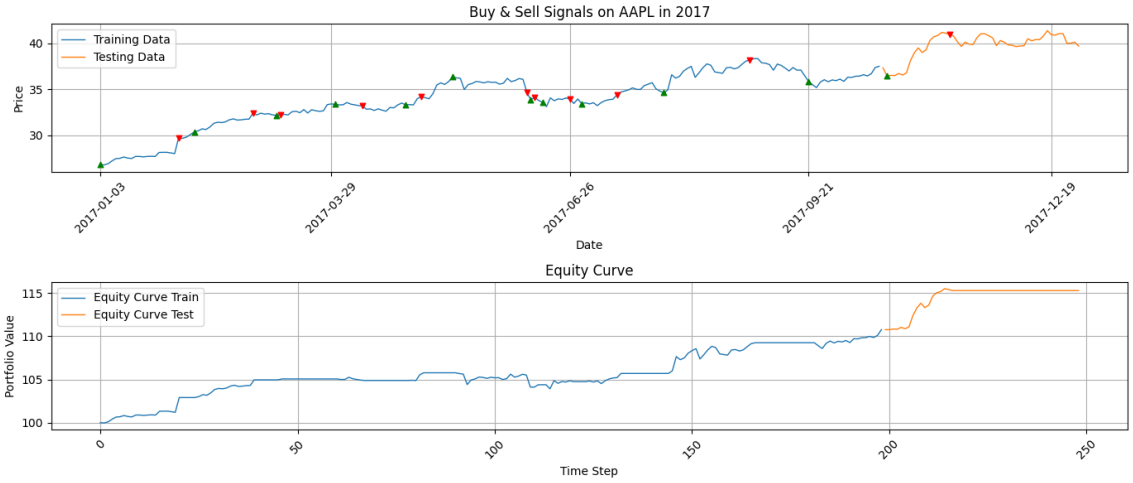
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

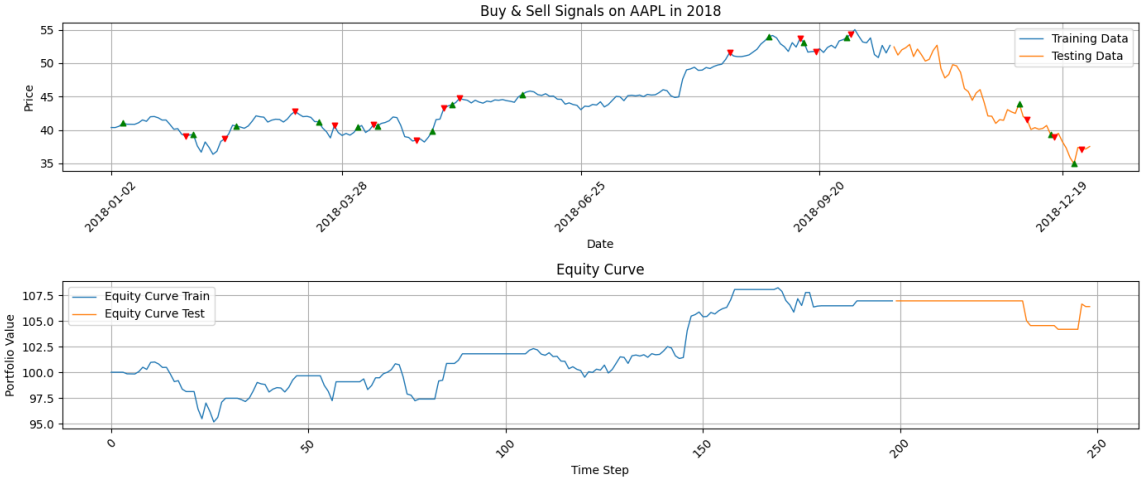

Buy & Sell Signals on AAPL in 2015
Training Data
Testing Data
Price
Date
2015-01-02
2015-03-30
2015-06-25
2015-09-22
2015-12-18

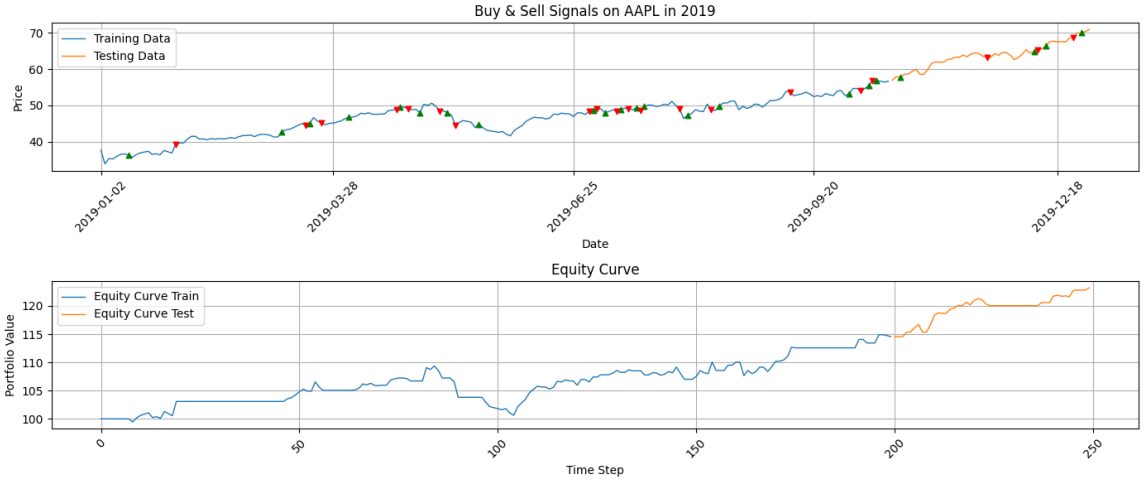

Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step

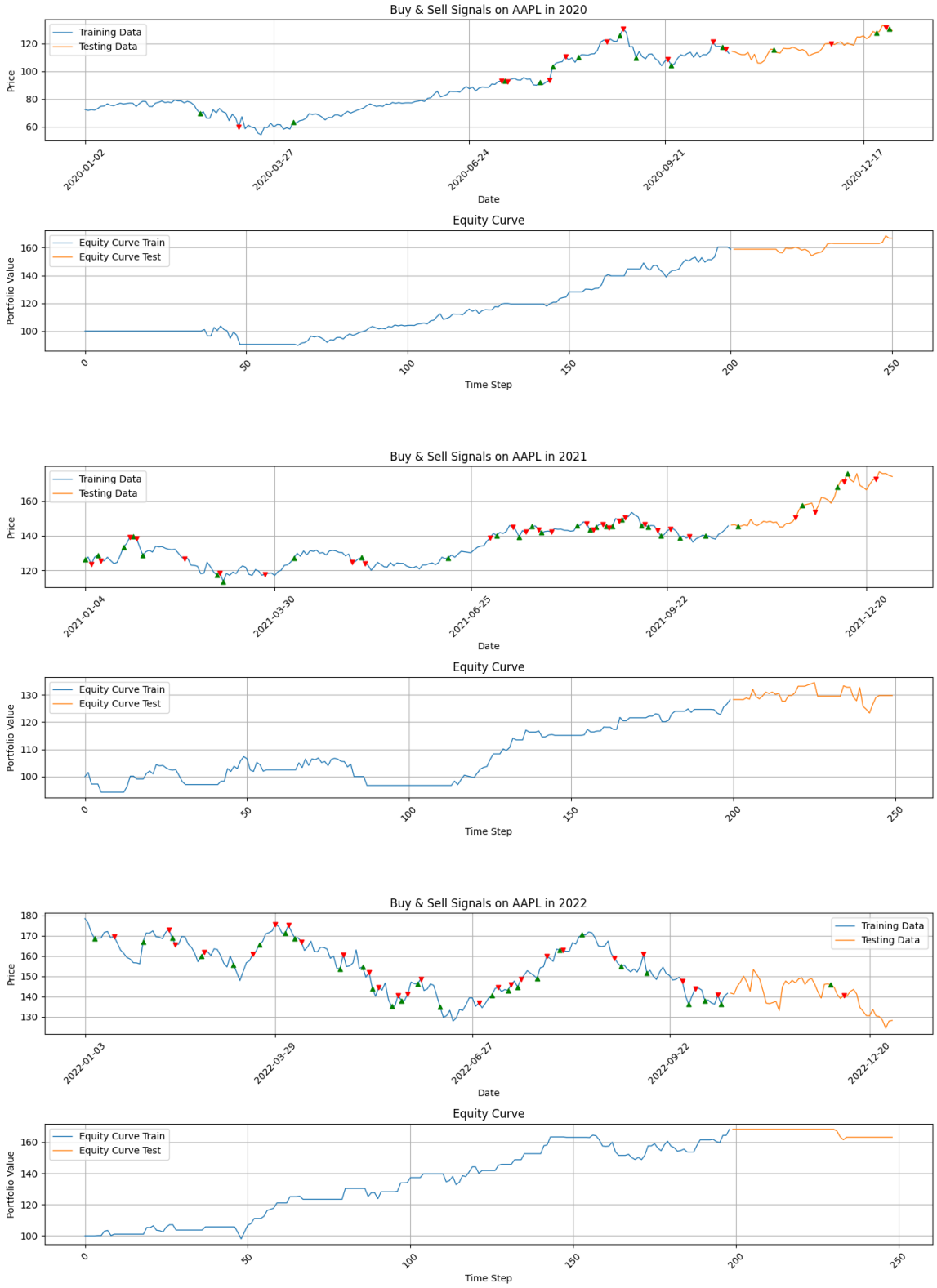

Buy & Sell Signals on AAPL in 2016
Training Data
Testing Data
Price
Date
2016-01-04
2016-03-30
2016-06-24
2016-09-21
2016-12-19


Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step


Buy & Sell Signals on AAPL in 2017
Training Data
Testing Data
Price
Date
2017-01-03
2017-03-29
2017-06-26
2017-09-21
2017-12-19
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step


Buy & Sell Signals on AAPL in 2018
Training Data
Testing Data
Price
Date
2018-01-02
2018-03-28
2018-06-25
2018-09-20
2018-12-19
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step


Buy & Sell Signals on AAPL in 2019
Training Data
Testing Data
Price
Date
2019-01-02
2019-03-28
2019-06-25
2019-09-20
2019-12-18
Equity Curve
Equity Curve Train
Equity Curve Test
Portfolio Value
Time Step



- Over 100 episodes, the cumulative rewards exhibited a modest upward trend, indicating initial learning progress, though with significant variability.
- Extending to 1000 episodes resulted showed that the agent is not taking position anymore, a convergence of the hold position shouyld be evaluated
- Value estimates became more consistent over time, though some fluctuations persisted due to the noisy environment dynamics (should add filtering or smoothing).

- The results highlight the trade-off between exploration and exploitation inherent in the greedy approach and motivate future exploration of strategies with controlled exploration. Noticed that the agent is not learning from the path but more on the direct previous value.

These preliminary findings establish a baseline for Q-Learning performance and set the stage for comparisons with other reinforcement learning methods such as Deep Q-Networks (DQN).

10.3 Challenges and Solutions

The Q-Learning experiments faced several challenges specific to the greedy policy and the trading environment :

- **Limited exploration due to greediness** : The strictly greedy policy limited the agent's ability to discover better actions, particularly early in training. Future work will include incorporating greedy or softmax action selection to balance exploration.
- **Reward noise and sparsity** : The stochastic environment produced noisy and sparse rewards, complicating learning.
- **Hyperparameter sensitivity** : Learning rate and discount factor tuning were critical, especially over longer training horizons. We will use optuna to get better hyperparameter in the future on more complexe models.
- **Computational efficiency** : Although less demanding than deep RL methods, Q-Learning with large state spaces required careful management of updates. We leveraged experience replay buffers and batch updates to accelerate convergence.

10.4 Next Steps

Building on the current progress with Q-Learning and the greedy policy, the immediate next steps focus on advancing the model complexity and preparing for PPO :

- Designing and integrating a deep neural network architecture to implement Deep Q-Learning (DQN), enabling function approximation for larger and continuous state spaces.
- Conducting experiments to compare the performance of DQN against the baseline Q-Learning greedy policy.
- Following the DQN implementation, proceeding with the development and fine-tuning of the PPO agent, including enhanced reward shaping and hyperparameter optimization.
- Expanding the trading environment to model more realistic market conditions such as transaction costs, slippage, and possibly varying liquidity.

- Performing ablation studies to isolate the effects of PPO-specific components like clipping, value function loss weighting, and entropy regularization.
- Enhancing visualization tools to track detailed performance metrics such as Sharpe ratio, maximum drawdown, and other risk-adjusted return measures.

These steps will establish a robust foundation for evaluating advanced reinforcement learning algorithms in the trading domain and inform subsequent research and publication efforts.

Chapter 11

Week 11

Contents

11.1 Deep Q-Learning

11.1.1 Problem Formulation

We cast single-asset trading as a finite-horizon Markov Decision Process (MDP) on time-indexed observations $\{o_t\}_{t=1}^{T}$. At each discrete time step t, the agent observes a state $s_t \in \mathscr{S}$ (Price, Close, High, Low,Open and Volume), takes an action $a_t \in \mathscr{A}$(Buy, Old or Sell), receives a reward $r_t \in \mathbb{R}$.

State. To incorporate temporal context, we define

$$s_t = \big[\phi(o_{t-w+1}), \ldots, \phi(o_t)\big] \in \mathbb{R}^{w\times d},$$

where w is a rolling window length and $\phi(\cdot)$ is a feature map including (but not limited to): log-returns, rolling z-scores, realized volatility estimates, microstructure features (e.g., imbalance), and technical indicators. To avoid lookahead bias, all rolling statistics are computed using past data only and fit on the training split.

Action Space. We consider a discrete policy with position control and an optional *hold* action:

$$\mathscr{A} = \{-K, \ldots, -1, 0, 1, \ldots, K\},$$

where a_t represents the target position (short to long) in normalized units subject to a position limit $|a_t| \le K$. An alternative is the set $\{\textsc{Short}, \textsc{Flat}, \textsc{Long}\}$ with an additional HOLD that preserves the prior position.

Reward. Let P_t be the execution price proxy, $\Delta P_{t+1} = P_{t+1} - P_t$, and $\Delta a_t = a_t - a_{t-1}$. A trading-aware reward that accounts for P&L, costs, and risk is

$$r_t \;=\; a_t \cdot \Delta P_{t+1} \;-\; c\,|\Delta a_t| \;-\; \lambda_{\text{risk}}\,\hat{\sigma}_t^2 \;-\; \lambda_{\text{dd}}\,\max(0, \text{DD}_t - \text{DD}_{\max}), \tag{11.1}$$

where c is per-unit transaction/slippage cost, $\hat{\sigma}_t^2$ is an ex-ante volatility estimate, and DD_t is running drawdown. The last term softly penalizes breach of a drawdown budget $\text{DD}_{\max}$.

11.1.2 Deep Q-Network (DQN)

DQN approximates the action-value function $Q^\star(s,a)$ with a neural network $Q_\theta(s,a)$ trained to minimize the temporal-difference (TD) error using a replay buffer $\mathscr{D}$ and a target network $Q_{\bar{\theta}}$.

Bellman Target. For terminal indicator $d_{t+1} \in \{0,1\}$,

$$y_t^{\text{DQN}} = r_t + \gamma(1-d_{t+1}) \max_{a'} Q_{\bar{\theta}}(s_{t+1}, a'), \tag{11.2}$$

$$\mathscr{L}(\theta) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1},d_{t+1})\sim\mathscr{D}}\left[\ell_\kappa\big(y_t - Q_\theta(s_t,a_t)\big)\right], \tag{11.3}$$

where $\ell_\kappa(\cdot)$ is the Huber loss for robustness to heavy-tailed TD errors. The discount $\gamma \in (0,1)$ should reflect the trading horizon (e.g., $\gamma \in [0.95, 0.999]$ for intraday vs. swing horizons).

Double DQN. To reduce overestimation, we use Double DQN with decoupled action selection/evaluation:

$$y_t^{\text{DDQN}} = r_t + \gamma(1-d_{t+1})\, Q_{\bar{\theta}}\Big(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a')\Big). \tag{11.4}$$

Architecture. For time-series, Q_θ can be (i) a 1D-CNN over the window w, (ii) an LSTM/GRU encoder with the last hidden state feeding an MLP head that outputs $|\mathscr{A}|$ Q-values, or (iii) a Transformer encoder for long-range dependencies. Layer normalization and dropout mitigate non-stationarity; `ReLU/GELU` activations are typical.

Stabilization. We employ (i) target network smoothing with soft updates $\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}$, $\tau \ll 1$, (ii) prioritized replay with sampling probability $\propto |\delta_t|^\alpha$ and importance weights, and (iii) action masking when risk or inventory limits are hit.

11.1.3 Exploration and Constraints

We use ϵ-greedy with linear or cosine decay from $\epsilon_{\max}$ to $\epsilon_{\min}$; in practice, *noisy layers* can replace explicit ϵ for state-dependent exploration. Trading-specific constraints (max position K, max turnover, exposure to news embargo windows) are enforced via an action mask $m_t(a) \in \{0,1\}$ and

$$a_t = \arg\max_{a \in \mathscr{A}:\, m_t(a)=1} Q_\theta(s_t, a).$$

11.1.4 Training Algorithm

11.1.5 Practical Considerations for Time-Series

Data Splitting & Leakage. Use chronological splits and *walk-forward* evaluation: rolling train/validation windows with a held-out test period. All preprocessing (scalers, PCA, feature selection) must be fit on training only and applied forward.

Stationarity & Regimes. Markets are non-stationary; periodic target network updates (τ) and shorter replay horizons help. Consider re-training or fine-tuning across regimes and adding a *regime feature* (e.g., volatility state) to s_t.

Costs & Slippage. Model costs explicitly in r_t and optionally inject execution noise during training to bridge the sim-to-real gap. Limit turnover via the $|\Delta a_t|$ penalty.

Risk Controls. In addition to reward penalties, enforce hard caps: max position K, max leverage, and a circuit breaker when rolling drawdown exceeds $\text{DD}_{\max}$.

11.1.6 Evaluation Metrics

Let $\{R_t\}$ be realized returns from the executed strategy. Report:

- Annualized Sharpe: $\mathrm{SR} = \dfrac{\sqrt{A}\,\mathbb{E}[R_t]}{\mathrm{Std}[R_t]}$ with A the periods-per-year factor.
- Sortino, Calmar, hit ratio, average trade, turnover, max drawdown, and profit factor.
- Stability: rolling SR and drawdown; sensitivity to cost c; ablations (no costs, no risk term, no mask).

Always compare to baselines (buy-and-hold, momentum/mean-reversion heuristics) and include a *purged, embargoed* cross-validation if you use overlapping windows.

11.1.7 Model Variants (Optional)

- **Dueling DQN:** Decompose $Q_\theta(s,a) = V_\theta(s) + A_\theta(s,a) - \frac{1}{|\mathscr{A}|}\sum_{a'} A_\theta(s,a')$ to stabilize value estimation.
- **N-step Returns:** Replace y with n-step target $r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_{a'} Q_{\tilde{\theta}}(s_{t+n}, a')$.
- **Distributional RL:** Learn the return distribution $Z(s,a)$ for better risk-sensitive control.
- **Noisy Nets:** Parameterized noise in linear layers for exploration without ϵ-schedules.

11.1.8 Reference Hyperparameters (Typical Ranges)

Parameter	Typical value
Window length w	32–256 steps
Discount γ	0.95–0.999
Optimizer / LR	Adam, 10^{-4} to $3\cdot 10^{-4}$
Batch size B	64–256
Replay size $\lvert\mathscr{D}\rvert$	10^5–10^6
Target update	soft $\tau \in [10^{-3}, 10^{-2}]$ (or hard every 1–5k steps)
ϵ schedule	from 1.0 to 0.05 over 10^5 steps (or NoisyNets)
Cost c	set by venue; stress $\times$2–4 for robustness
Risk weights	$\lambda_{\text{risk}}, \lambda_{\text{dd}}$ via grid search on validation

11.1.9 Limitations

DQN assumes a stationary $Q^\star$ and Markovian dynamics, both often violated in markets. Performance can degrade under regime shifts, changing costs/liquidity, or adversarial feedback. Robustness checks (stress costs, volatility spikes, delayed fills) and conservative deployment (small capital, shadow trading) are essential.

11.2 Transformer-Based Deep Q-Learning for Time-Series Trading

11.2.1 Problem Formulation

We frame single-asset trading as a finite-horizon Markov Decision Process (MDP) over price and feature sequences $\{o_t\}_{t=1}^{T}$. At each time t, the agent observes a state $s_t \in \mathscr{S}$, selects an action $a_t \in \mathscr{A}$, receives reward r_t, and transitions to s_{t+1}.

State. The state is a sequence of past observations:

$$s_t = \big[\phi(o_{t-w+1}), \ldots, \phi(o_t)\big] \in \mathbb{R}^{w \times d},$$

where w is the window size, d the feature dimension, and $\phi(\cdot)$ includes log-returns, volatility, and other technical indicators.

Action Space. We use a discrete set of position actions:

$$\mathscr{A} = \{-K, \ldots, -1, 0, 1, \ldots, K\},$$

representing target positions (short to long) subject to max position K.

Reward. The reward accounts for P&L, trading costs, and risk:

$$r_t = a_t \cdot \Delta P_{t+1} - c|\Delta a_t| - \lambda_{\text{risk}} \hat{\sigma}_t^2 - \lambda_{\text{dd}} \max(0, \text{DD}_t - \text{DD}_{\max}).$$

11.2.2 Transformer Q-Network

Instead of a traditional CNN/LSTM, we use a Transformer encoder to model long-range dependencies in time-series. The network $Q_\theta(s_t, a_t)$ is parameterized as:

- Input: sequence of feature vectors s_t.
- Positional encodings added to preserve temporal order.
- Stacked Transformer encoder layers with multi-head attention.
- Final MLP head outputs $|\mathscr{A}|$ Q-values.

Formally, let $\text{Transformer}_\theta(\cdot)$ denote the output embedding for the last token:

$$Q_\theta(s_t, a) = \text{MLP}_\theta\big(\text{Transformer}_\theta(s_t)\big)_a.$$

11.2.3 Training with Double DQN Targets

The Transformer Q-network is trained using Double DQN targets:

$$y_t = r_t + \gamma(1 - d_{t+1}) Q_{\bar{\theta}}\Big(s_{t+1}, \arg\max_{a'} Q_\theta(s_{t+1}, a')\Big),$$

where $\bar{\theta}$ is a target network updated softly: $\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}$.

The loss is the Huber loss over a replay buffer $\mathscr{D}$:

$$\mathscr{L}(\theta) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}, d_{t+1}) \sim \mathscr{D}}\big[\ell_\kappa(y_t - Q_\theta(s_t, a_t))\big].$$

11.2.4 Exploration and Constraints

- **Exploration:** ϵ-greedy or parameter noise (NoisyNet layers) in the MLP head.
- **Constraints:** Action masks enforce max position, turnover, and risk limits.

11.2.5 Practical Considerations

- **Data Splitting:** Walk-forward evaluation to prevent lookahead bias.
- **Stationarity:** Transformers can capture longer temporal dependencies but may still require retraining on regime shifts.
- **Costs & Slippage:** Include in the reward function to improve robustness.
- **Hyperparameters:** Window length w, number of Transformer layers, number of attention heads, hidden dimensions, learning rate, batch size, replay buffer size.

11.2.6 Evaluation Metrics

Use the same trading metrics as before: Sharpe, Sortino, maximum drawdown, hit ratio, turnover, and profit factor. Compare against buy-and-hold and heuristic baselines.

11.2.7 Remarks

Replacing the RNN/CNN with a Transformer enables the agent to capture longer-range dependencies in time-series, which is beneficial for assets with complex temporal patterns or irregular cycles. Care must be taken to limit overfitting due to increased model capacity.

Algorithm 3: Deep Q-Learning Training

Input: Environment env, data frame df, training size $train\_size$, episode N

Output: Trained Q-table Q

Split data: $df_{train} \leftarrow train\_size$ of df;

Calculate bins for discretization : $bins \leftarrow$ compute bins from df_{train};

Initialize Q-table

for $episode \leftarrow 1$ **to** N

  Exploration rate : $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$;

  Reset environment : $state_{cont} \leftarrow env.reset()$;

  Discretize initial state : $state_{disc} \leftarrow discretize(state_{cont}, bins)$;

  Convert to index : $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$;

  for $step \leftarrow 1$ **to** max_steps

    Choose action a using epsilon-greedy policy :

$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1-\epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

    Take action : $(next_{state_{cont}}, r, done, info) \leftarrow env.step(a)$;

    Discretize next state : $next_{state_{disc}} \leftarrow discretize(next_{state_{cont}}, bins)$;

    Convert to index : $next_{state_{idx}} \leftarrow state\_to\_index(next_{state_{disc}}, bins)$;

    Update Q-value

    $:Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha\left(r + \gamma \max_{a'} Q[next_{state_{idx}}, a'] - Q[state_{idx}, a]\right)$;

    $state_{idx} \leftarrow next_{state_{idx}}$;

    if $done$ **then**

      break;

return Q

Chapter 12

Week 12

Contents

12.1 Deep Q-Learning

12.1.1 Deep Q-Learning introduction

In order to approximate optimal decision-making in complex environments, we employ Deep Q-Learning, an extension of Q-Learning that integrates state discretization with function approximation techniques. Unlike traditional tabular Q-Learning, which directly maintains a Q-table over discrete states, Deep Q-Learning is capable of handling continuous or high-dimensional state spaces by discretizing them into manageable bins. The algorithm balances exploration and exploitation using an ϵ-greedy strategy with exponential decay, ensuring sufficient exploration during early episodes while gradually converging toward exploitation of learned policies. At each training step, the Q-table is updated via the Bellman equation, incorporating observed rewards and estimated future returns. The following pseudocode outlines the full training procedure.

Algorithm 4: Deep Q-Learning Training

Input: Environment env, data frame df, training size $train\_size$, episode N

Output: Trained Q-table Q

Split data: $df_{train} \leftarrow train\_size$ of df;

Calculate bins for discretization : $bins \leftarrow$ compute bins from df_{train};

Initialize Q-table

for $episode \leftarrow 1$ **to** N

  Exploration rate : $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$;

  Reset environment : $state_{cont} \leftarrow env.reset()$;

  Discretize initial state : $state_{disc} \leftarrow discretize(state_{cont}, bins)$;

  Convert to index : $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$;

  for $step \leftarrow 1$ **to** max_steps

    Choose action a using epsilon-greedy policy :

$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1-\epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

    Take action : $(next_{state_{cont}}, r, done, info) \leftarrow env.step(a)$;

    Discretize next state : $next_{state_{disc}} \leftarrow discretize(next_{state_{cont}}, bins)$;

    Convert to index : $next_{state_{idx}} \leftarrow state\_to\_index(next_{state_{disc}}, bins)$;

    Update Q-value

    $:Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha\left(r + \gamma \max_{a'} Q[next_{state_{idx}}, a'] - Q[state_{idx}, a]\right)$;

    $state_{idx} \leftarrow next_{state_{idx}}$;

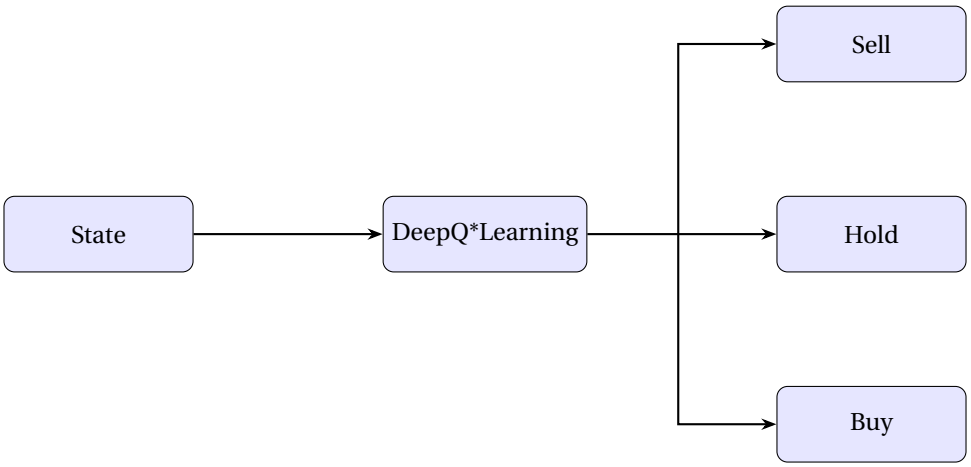
    if $done$ **then**

      break;

return Q

12.1.2 Deep Q-Learning Process

To implement this method, we will use the same Q-Learning algorithm object and change during the training to a Neural Network. As a first example we will use a MLP.



12.1.3 MLP layer

The Multi-Layer Perceptron (MLP) layer serves as the core of the neural network, responsible for learning and transforming input data through a series of connected, dense layers. It is composed of multiple fully-connected layers, each followed by a non-linear activation function. This non-linearity is crucial as it allows the network to model complex, non-linear relationships in the data that a simple linear model could not capture.

The `MLP` class is defined as a PyTorch module, which is a standard approach for building neural network components. The constructor, `__init__`, initializes the network's architecture. It takes three key arguments: `input_dim` (the dimensionality of the input data), `output_dim` (the number of output units), and `hidden_dims` (a tuple specifying the number of units in each hidden layer, which defaults to `(128, 128)`). The implementation uses a loop to dynamically build the hidden layers. For each dimension specified in `hidden_dims`, it adds a `nn.Linear` layer (a fully-connected layer) followed by a `nn.ReLU` activation function. The Rectified Linear Unit (ReLU) is chosen for its computational efficiency and its effectiveness in preventing the vanishing gradient problem.

After the hidden layers are constructed, a final `nn.Linear` layer is added. This last layer maps the output of the final hidden layer to the desired `output_dim` of the network. The entire sequence of layers is then encapsulated into a single `nn.Sequential` container, which ensures that the data will be passed through the layers in the correct order during the forward pass.

Layer	Type
`nn.Linear`	Linear Layer
`nn.ReLU`	Activation Function
`nn.Sequential`	Container

Table 12.1: Explanation of the MLP layers

After a 2-hour training period, the model's performance was evaluated on a test dataset. The results are visualized in the following plot :

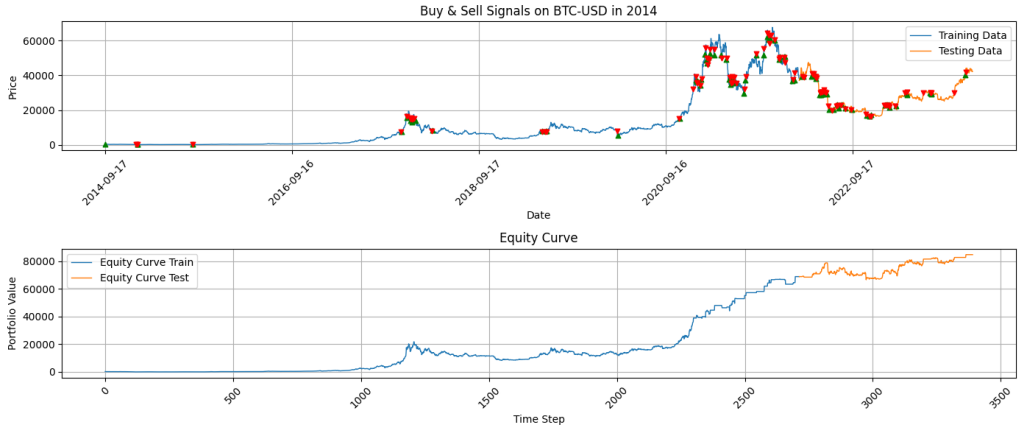


Figure 12.1: Performance of the Q-Learning Model on a BTC-USD test set

The plot demonstrates that the model was able to generate positive returns, indicating a degree of success in the training. However, it is critical to note that this test was conducted on data from an overall uptrend market. This suggests the model's profitability may be a result of the market's general direction rather than its ability to make strategic decisions in both bullish and bearish conditions.

A closer inspection of the training log and the trading history reveals that the model made a significant number of "illegal moves." These unauthorized actions, which violate the predefined trading constraints or rules of the environment, are a clear indication of a failure in the model's policy and training process. This result suggests that while the model found a way to profit, it did so by exploiting an oversight in the environment or by not correctly learning the full set of trading rules. Further work is required to correct these behaviors and ensure the model operates within the defined constraints, leading to a more robust and generalizable trading policy.

To further analyze the model's performance across different market conditions, we present additional trading simulations on various stock datasets :

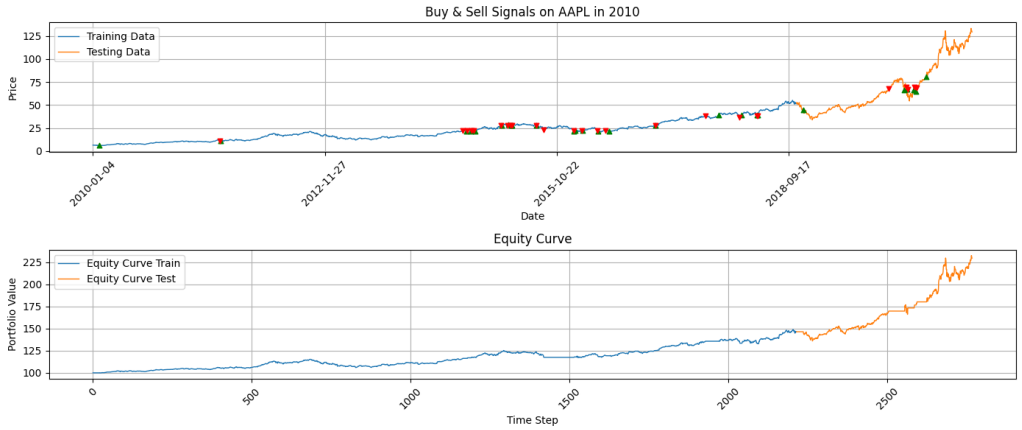
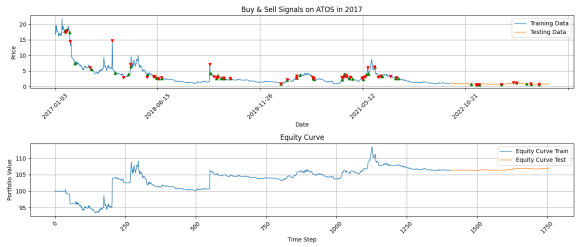
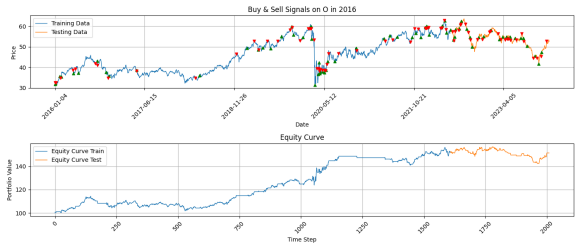


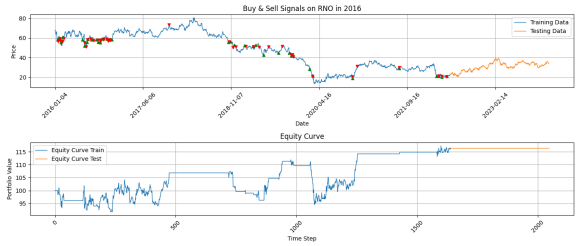
Figure 12.2: Performance on Apple (AAPL) data from 2010.



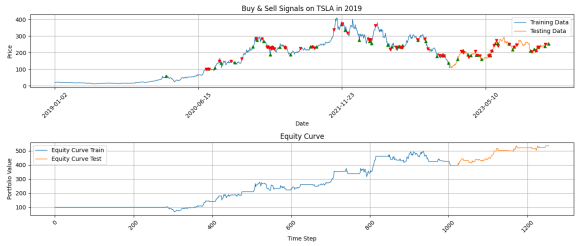
(a) Performance on ATOS data from 2017.



(b) Performance on Realty Income (O) data from 2016.



(c) Performance on Renault (RNO) data from 2016.



(d) Performance on Tesla (TSLA) data from 2019.

Figure 12.3: Model performance across various market conditions.

The plots demonstrate that the model was able to generate positive returns, indicating a degree of success in the training. However, it is critical to note that the test on the BTC-USD data was conducted on an overall uptrend market. This suggests the model's profitability may be a result of the market's general direction rather than its ability to make strategic decisions in both bullish and bearish conditions.

A closer inspection of the training log and the trading history reveals that the model made a significant number of "illegal moves." These unauthorized actions, which violate the predefined trading constraints or rules of the environment, are a clear indication of a failure in the model's policy and training process. This result suggests that while the model found a way to profit, it did so by exploiting an oversight in the environment or by not correctly learning the full set of trading rules. Further work is required to correct these behaviors and ensure the model operates within the defined constraints, leading to a more robust and generalizable trading policy.

12.1.4 Decision Transformer Layer

Unlike a traditional Multi-Layer Perceptron (MLP) that processes a single state at a time, the DecisionTransformerQ layer is designed to handle sequential data, leveraging the powerful architecture of a transformer. This approach frames reinforcement learning as a sequence modeling problem, where the model learns to predict future actions based on a history of past states and desired returns.

The `DecisionTransformerQ` class is built upon a pre-trained transformer model from Hugging Face's library, which serves as the core 'backbone'. The constructor, `__init__`, initializes a configuration for the transformer and loads the corresponding model ('DecisionTransformerGPT2Model'). This approach allows the network to benefit from the pre-trained weights, which are already effective at capturing complex patterns in sequential data.

Before the input is fed into the transformer, it passes through a 'self.input_proj' linear layer. This layer's purpose is to project the raw input data, which has a dimensionality of `input_dim`, into an embedding space that matches the transformer's hidden size. This ensures the input is correctly formatted for the transformer architecture.

The `forward` method outlines the data flow. First, it ensures the input tensor `x` is in the correct shape for the transformer (a 3D tensor representing a batch, sequence of steps, and features). The input is then passed to the 'self.input_proj' layer for projection. The core of the computation happens when the projected data is fed into the transformer 'backbone'. The transformer processes the entire sequence and its output's 'last_hidden_state' is used. This final hidden state is a rich representation that has attended to the entire sequence history, making it ideal for the final decision. Finally, the 'self.q_head' linear layer takes this comprehensive representation and projects it to the desired `output_dim`, providing the predicted Q-values.

Layer	Type
`DecisionTransformerGPT2Model`	Transformer Backbone
`nn.Linear` (Input)	Linear Layer
`nn.Linear` (Q-Head)	Linear Layer

Table 12.2: Explanation of the Decision Transformer layers

After a 4-hour training period, the model's performance was evaluated on a test dataset. The results are visualized in the following plot:

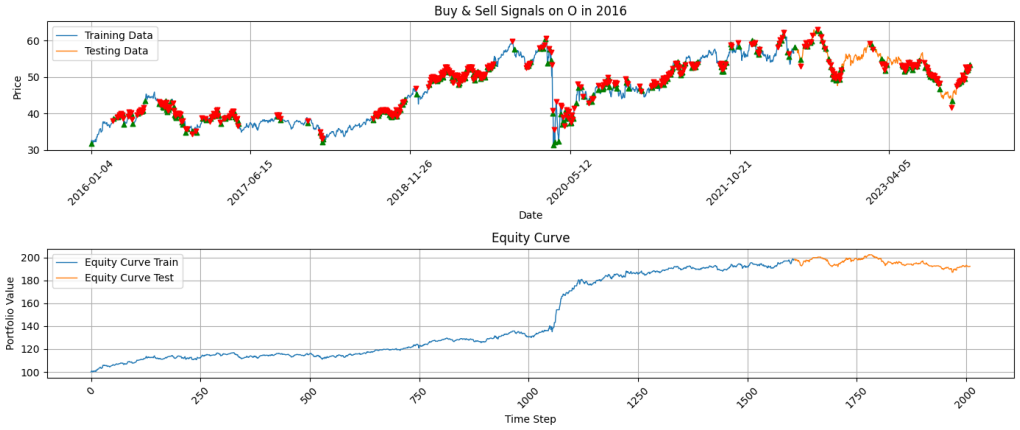


Figure 12.4: Performance of the Deep Q-Learning Model on Realty Income (O) data from 2016.

This model achieved its best results on the Realty Income (O) stock during both training and testing. While the model was able to generate positive returns, further analysis is required because the stock's volatility is high due to a large number of trades. The next step will be to add broker fees to the model to limit the number of trades and make the results more representative of real-world trading conditions.

Chapter 13

Week 13

Contents

13.1 Deep Q-Learning

13.1.1 Decision Transformer Layer

Unlike a traditional Multi-Layer Perceptron (MLP) that processes a single state at a time, the DecisionTransformerQ layer is designed to handle sequential data, leveraging the powerful architecture of a transformer. This approach frames reinforcement learning as a sequence modeling problem, where the model learns to predict future actions based on a history of past states and desired returns.

The `DecisionTransformerQ` class is built upon a pre-trained transformer model from Hugging Face's library, which serves as the core 'backbone'. The constructor, `__init__`, initializes a configuration for the transformer and loads the corresponding model ('DecisionTransformerGPT2Model'). This approach allows the network to benefit from the pre-trained weights, which are already effective at capturing complex patterns in sequential data.

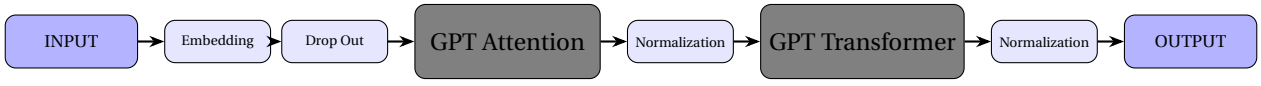
Before the input is fed into the transformer, it passes through a 'self.input_proj' linear layer. This layer's purpose is to project the raw input data, which has a dimensionality of `input_dim`, into an embedding space that matches the transformer's hidden size. This ensures the input is correctly formatted for the transformer architecture.

The `forward` method outlines the data flow. First, it ensures the input tensor `x` is in the correct shape for the transformer (a 3D tensor representing a batch, sequence of steps, and features). The input is then passed to the 'self.input_proj' layer for projection. The core of the computation happens when the projected data is fed into the transformer 'backbone'. The transformer processes the entire sequence and its output's 'last_hidden_state' is used. This final hidden state is a rich representation that has attended to the entire sequence history, making it ideal for the final decision. Finally, the 'self.q_head' linear layer takes this comprehensive representation and projects it to the desired `output_dim`, providing the predicted Q-values.

Layer	Type
`DecisionTransformerGPT2Model`	Transformer Backbone
`nn.Linear` (Input)	Linear Layer
`nn.Linear` (Q-Head)	Linear Layer

Table 13.1: Explanation of the Decision Transformer layers

13.1.2 GPT2 Transformer description



As we can see, the GPT2 transformer is a basic transformer, however the good performance of this particular transformer is that the model has been created using the GPT2 opensource transformer available on hugging face. Here are some results of the model on different stocks.

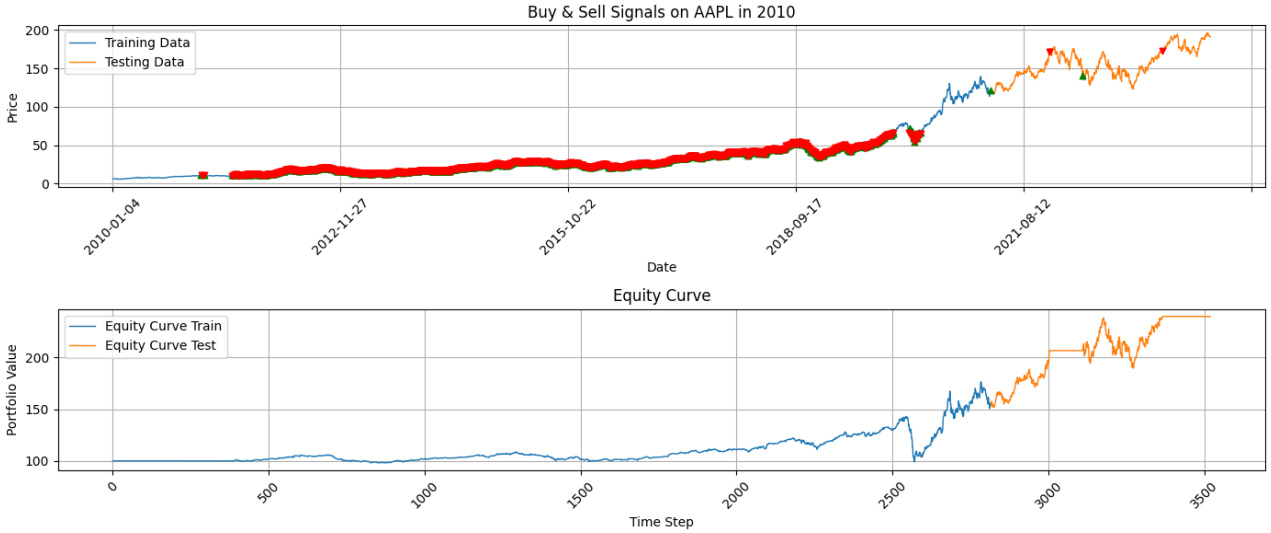


Figure 13.1: Performance of the Transofrmer Model on Apple data from 2010.

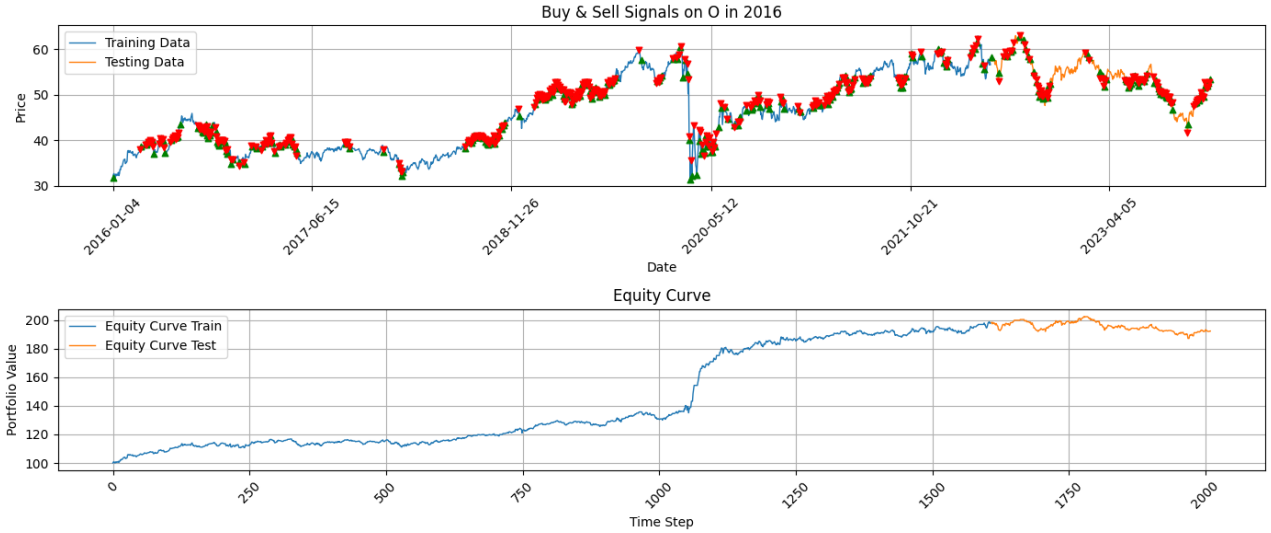


Figure 13.2: Performance of the Transofrmer Model on Realty income data from 2016.

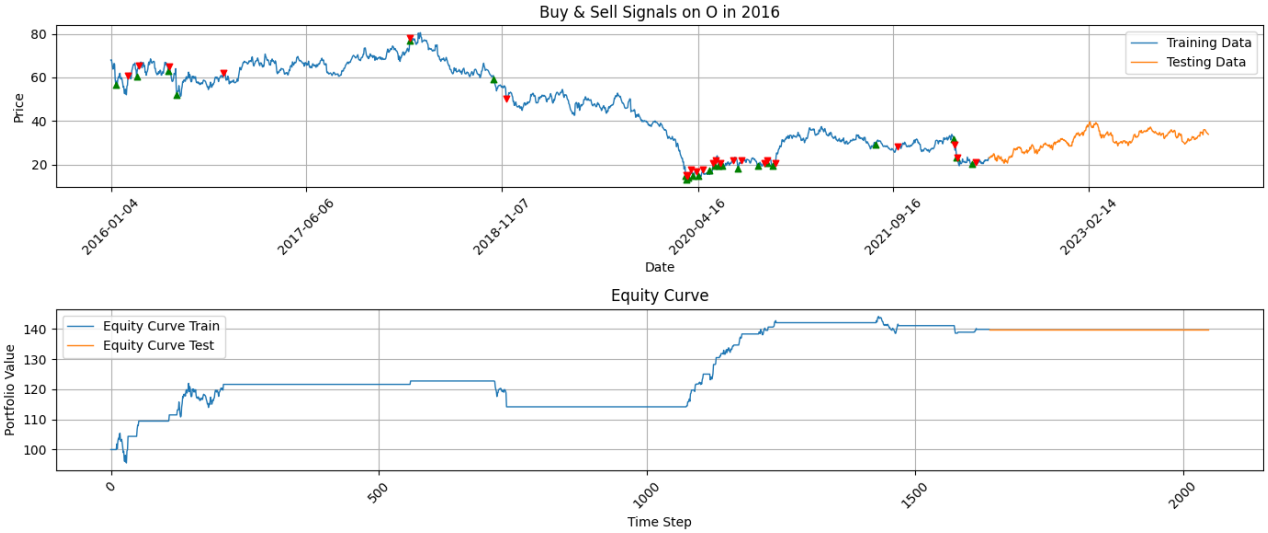


Figure 13.3: Performance of the Transofrmer Model on Renault data from 2016.

Figure 13.4: Performance of the Transofrmer Model on Tesla data from 2019.

As we can see, there is a too many trades that are made during the training and testing part, this is an issue caused by the penalty of the model ; there is no penalty on the number of trade the bot will make, this suggest that we need to find a way to make the bot understand that a trade should be made wiser. For that we suggest to add the broker's commission for a buy or sell action. Let's take a 10% commission which make the bot considering the number of trade made.

13.1.3 Result for QLearning (with commission)

We changed the environment behavior to provide a 10 % commission for each buy and sell movements. Also we changed the reward to provide only the difference between the initial wallet et the current portfolio so that the agent learn also on the market behavior directly without any outside penalties that could makes him learn false dynamic. The problem here could be that the agent does not learn about the rules of the set of action. For example, the issue here could be that the model do lot of illegals moves such as buying twice in a row, if that happen, we will change the reward back with penalties. Here are the results on the QLearning agent :

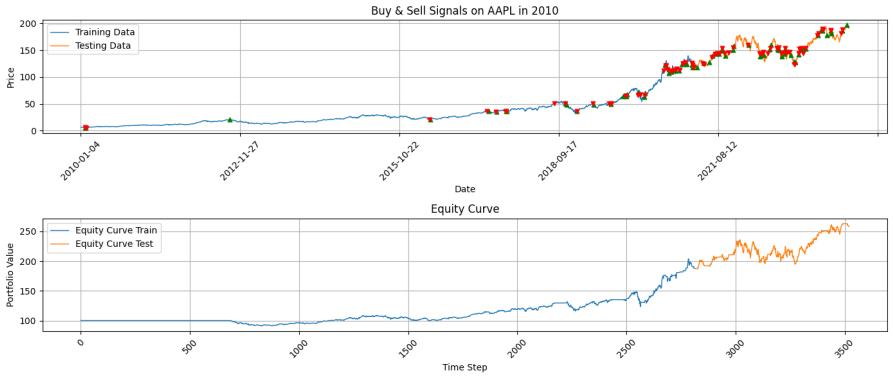


Figure 13.5: Performance of the QLearning Model on Apple data from 2010 with commission

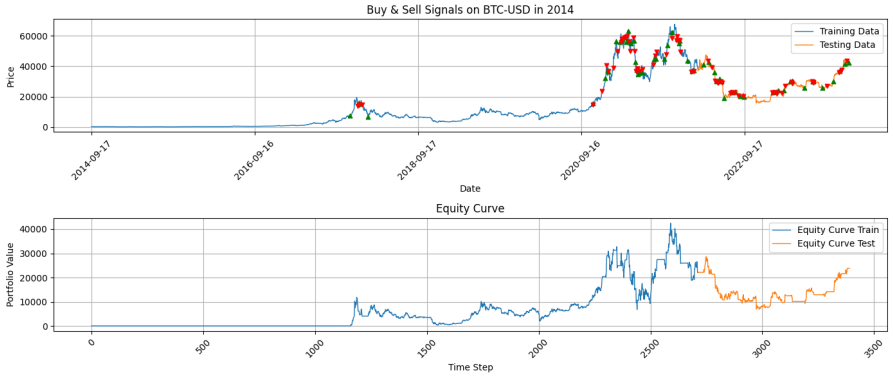


Figure 13.6: Performance of the QLearning Model on Bitcoin data from 2014 with commission

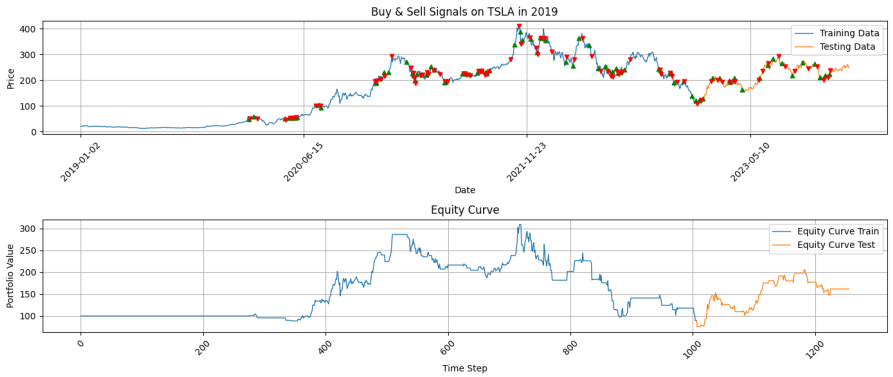


Figure 13.7: Performance of the QLearning Model on Tesla data from 2019 with commission

As we can see the number of trades reduced, however the performance is still poor since it is still following the trend of the actual stock. This last observation could be explained by the fact that QLearning is a basic Machine Learning algorithm.

13.1.4 Result for DeepQLearning - GPT Transformer (with commission)

We did the same experiment on the transformer model. However this model takes a while to train, we could not have a lot of output results so we will analyze only two experiments :

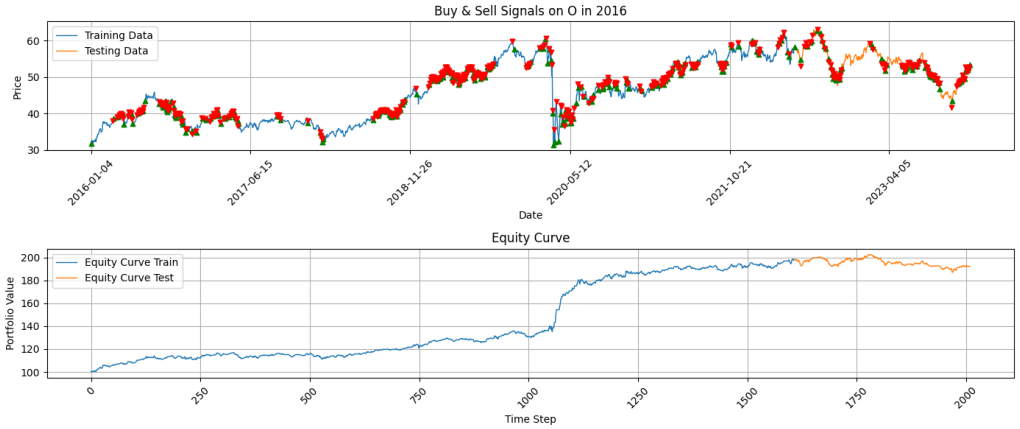


Figure 13.8: Performance of the Transformer Model on Real income data from 2016 with commission

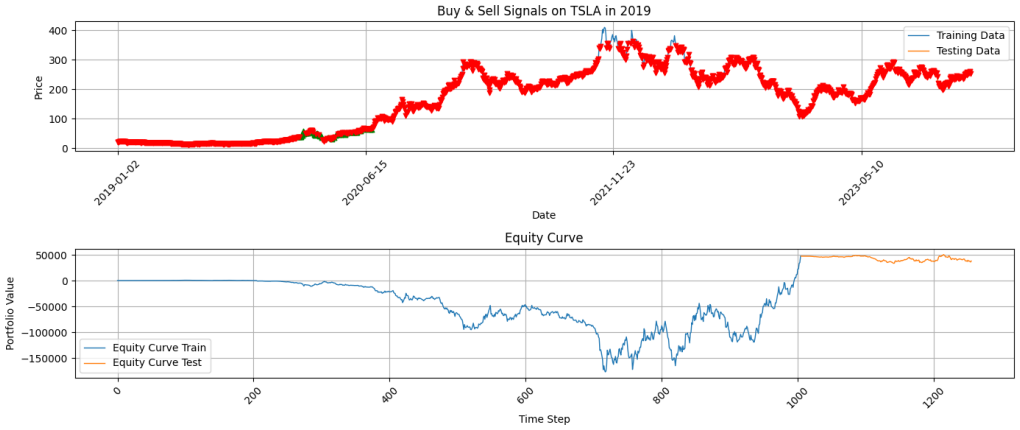


Figure 13.9: Performance of the Transformer Model on Tesla data from 2019 with commission

As we can see the performance on the training part of Real income is really good, the agent seems to understand the dynamic of the stock as a comparison, the current market made 80% profit on this period while our model made 98%, which means that on the training part our model get very well the dynamic aspect of the market. However we can see the limitation of the model during the testing part where our model made -2% and the current market lost 6% which indicates that our model also outperform the market but made us lose money still.

On another hand, on the Tesla market, the agent has completely been broke on his behaviour. The difference with the previous agent is that the environement is giving a reward based on the difference in between the initial wallet and the actual portfolio, which later in a report, is suggested as a better reward based for a trading bot using transformer. This could also be an issue from the transofrmer model itself (bad intergation of the model / bad purpose of the model), we will analyze this issue later.

13.2 Paper : Transformers in Reinforcement Learning : A survey

Transformers have emerged as a dominant architecture in modern machine learning, revolutionizing natural language processing, computer vision, and a growing number of other domains. Their ability to model long-range dependencies, capture contextual information, and scale effectively has prompted increasing interest in applying them to reinforcement learning (RL). In RL, agents must learn sequential decision-making policies under uncertainty, a setting that aligns naturally with the sequence modeling capabilities of transformers. Over the past few years, researchers have proposed a variety of transformer-based methods for RL tasks, ranging from policy optimization and value estimation to model-based approaches.

This survey, written in 2023 by Pranav Agarwal (École de Technologie Supérieure/Mila, Canada), Aamer Abdul Rahman (École de Technologie Supérieure/Mila, Canada), Pierre-Luc St-Charles (Mila, Applied ML Research Team, Canada), Simon J.D. Prince (University of Bath, United Kingdom) and Samira Ebrahimi Kahou (École de Technologie Supérieure/Mila/CIFAR, Canada), provides a comprehensive overview of this rapidly developing research area. The paper reviews how transformers have been adapted for reinforcement learning, highlights emerging architectures and methodologies, and categorizes them according to their applications. It also discusses empirical results, key challenges such as scalability and sample efficiency, and potential future research directions.

By situating transformer-based RL within the broader landscape of sequence modeling and decision-making, this survey contributes to a deeper understanding of both the opportunities and limitations of applying transformers in reinforcement learning. It serves as a reference point for researchers and practitioners seeking to build on the foundations of this promising intersection of fields. This paper talk about the transformer on many different fields but the interesting part for us is the trading part.

13.2.1 Transformer RL in Trading

Portfolio optimization aims to balance returns and risks when selecting assets but this is difficult due to market volatility and external factors. Reinforcement learning (RL) has been explored to automate trading decisions by learning from historical market data, such as price trends, volumes and sentiment.

Transformers are particularly suited for this task because they can capture both sequential patterns in asset prices and correlations between different assets. It has been introduced that the Relation-Aware Transformer (RAT) for portfolio selection, where the encoder extracts sequential and relational features, and the decoder makes trading decisions (including leverage and short sales). The approach was evaluated on real-world stock and cryptocurrency data, showing competitive performance compared to state-of-the-art portfolio selection methods. The given model would is described as an encoder-decoder transformer like so :

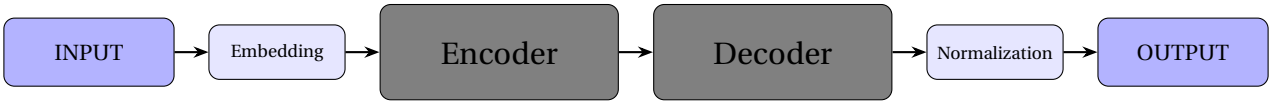


Figure 13.10: Encoder-Decoder Stack of the transofrmer model

Let us now dive into the encoder stack to understand how these sequential and relational features are extracted and represented for reinforcement learning-based portfolio optimization.

In the context of trading and portfolio optimization, the encoder plays a crucial role in capturing the complex patterns present in historical market data. It processes sequences of asset prices, trading volumes and other market indicators to model both short-term trends and long-term dependencies. By leveraging the self-attention mechanism inherent to transformers, the encoder can identify important relationships not only within a single asset's time series but also across multiple assets, effectively capturing correlations that are critical for informed trading decisions.

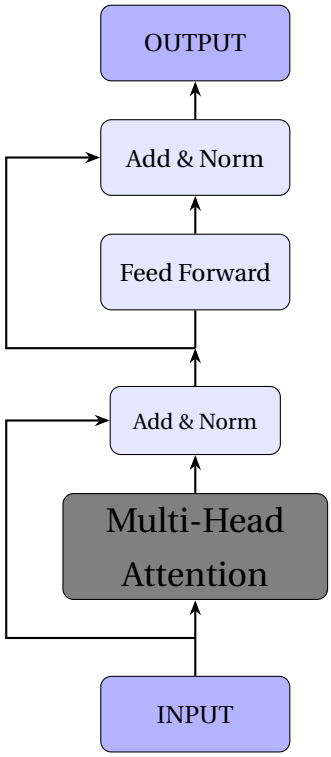


Figure 13.11: Encoder stack

In the context of a transformer model, we are using the Attention architecture to capture temporal dependencies and correlations in sequential trading data. The attention mechanism allows the model to weigh the importance of past states and actions when predicting future trading decisions, which is particularly useful in financial markets where certain events or trends can have long-term effects. The Multi-Head Attention (MHA) layer enables the model to simultaneously focus on different aspects of the input sequences, such as price trends, trading volumes and market sentiment. By splitting the attention into multiple heads, the network can learn diverse representations of the data, improving its ability to capture complex relationships between assets.

Let's now describe the two main component of these layers which are the Multi-Head Attention layer and the Feed Forward layer. We will also describe the basis of original Attention layer that is included into the Multi-Head Attention :

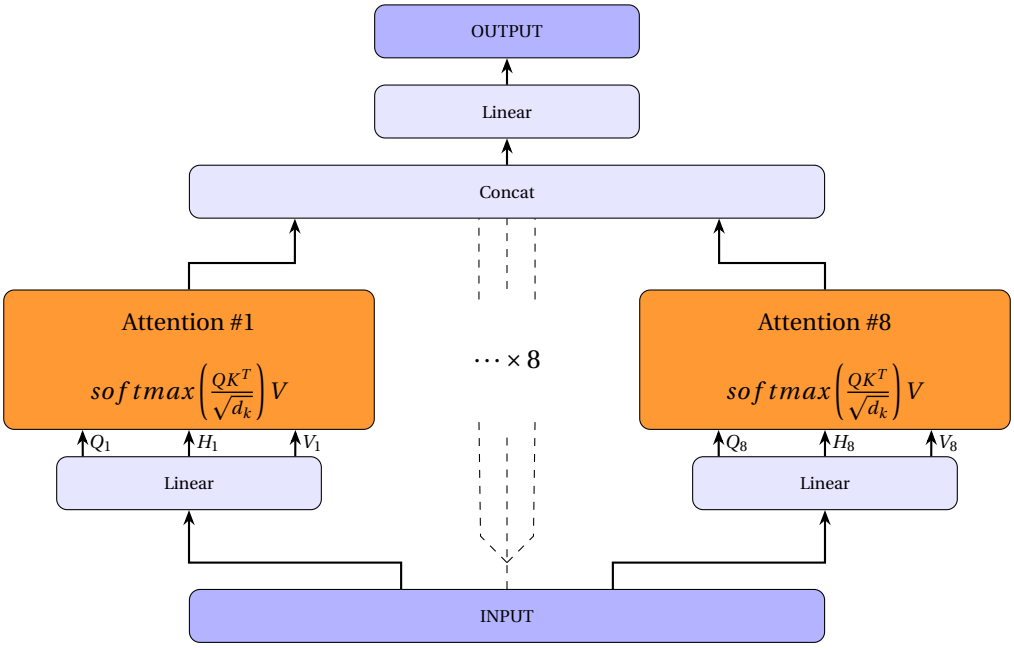


Figure 13.12: Multi-Head Attention stack (x8 heads)

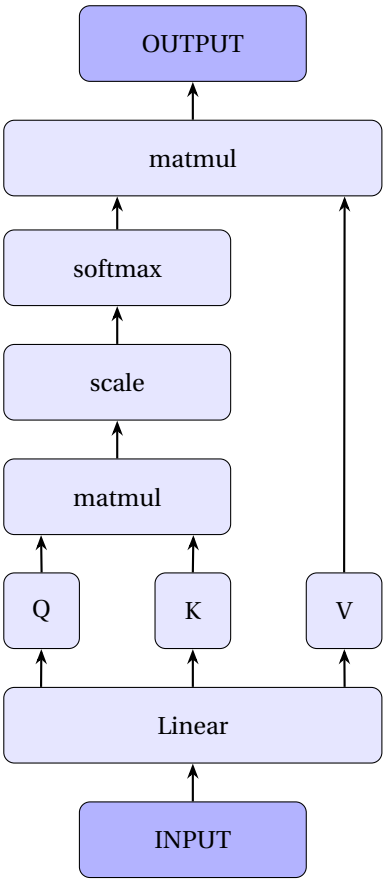


Figure 13.13: Attention stack

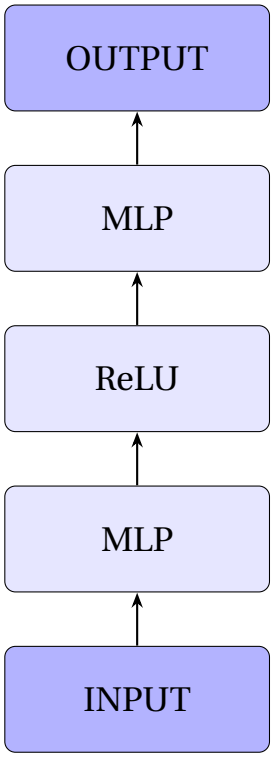


Figure 13.14: Feed-Forward stack

Finally, after implementing these layers in PyTorch, we will be able to start building the algorithm for a Deep Q-Learning agent using our own model, which should be capable of training on trading markets. The training strategy is as follows: the price assets are used as input to the encoder, which extracts features that are then passed to the decoder and the decision-making layer to produce the final action for a given input. We then compare the resulting portfolio with the initial one and assign a reward based on the evolution of the portfolio's value.

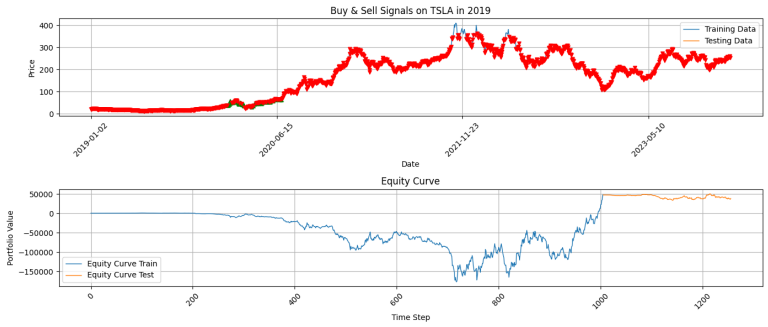
Chapter 14

Week 14

Contents

14.0.1 Model Behavior Issues

After adding the commission fees, we noticed several issues, as shown below:



14.0.2 Environment Issues

We initially suspected that the problems might come from the environment's behavior and reward design. The original reward was defined as simply returning the portfolio value (equity curve), so that the agent maximizes its final value. This produced the following reward:

Figure 14.1: Reward based on the portfolio value

However, this reward led to issues in position selection. Another idea was to maximize the immediate action by comparing the past portfolio with the current one, encouraging the agent to increase portfolio value step by step:

Figure 14.2: Reward based on portfolio differences

The problem with this approach is that it does not account for the overall final portfolio value. To address this, we kept the portfolio-based reward but multiplied it by the sign of the portfolio's derivative since the last decision, thereby favoring decisions aligned with short-term trends:

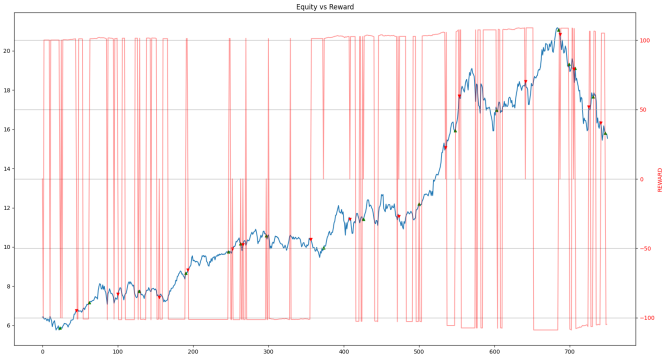


Figure 14.3: Reward based on the portfolio value and its derivative sign

Yet, this method still ignored the final portfolio value. Instead of just multiplying, we then added the portfolio value multiplied by the derivative, combining long-term growth with short-term movement:

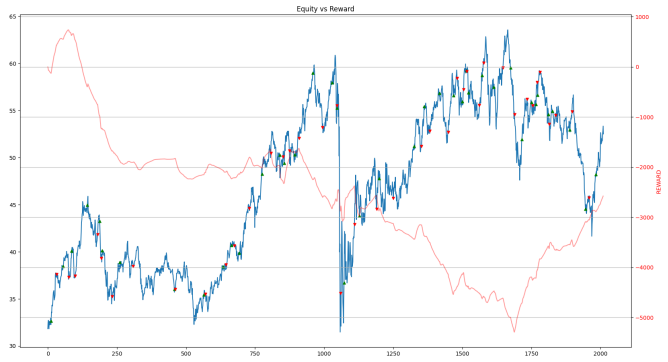


Figure 14.4: Reward combining portfolio value and its derivative

This reward performed better, but to further emphasize step-to-step differences, we applied a logarithmic transformation:

Figure 14.5: Reward with logarithmic adjustment

14.1 Fixing the Issues

To systematically address the previously identified issues, we are currently designing a series of experiments to compare the performance of standard Q-Learning across the different reward formulations discussed in the previous section. The goal is to identify which reward structure provides the most stable learning process and the best trade-off between short-term decision-making and long-term portfolio growth.

Our methodology proceeds in two main stages:

1. **Reward Function Evaluation.** Each proposed reward function (portfolio value, portfolio differences, portfolio with derivative sign, additive combination, and logarithmic adjustment) will be tested under identical market conditions and training parameters. Performance metrics will include cumulative portfolio return, maximum drawdown, Sharpe ratio, and policy stability across training runs. This evaluation will allow us to assess whether a reward function encourages overfitting to short-term fluctuations, ignores long-term profitability, or achieves a balance between the two.

2. **Hyperparameter Optimization.** After identifying the most promising reward functions, we will perform systematic hyperparameter tuning to further improve model performance. For this purpose, we will use *Optuna*, a state-of-the-art hyperparameter optimization framework that leverages efficient sampling and pruning strategies. Key hyperparameters to be tuned include learning rate, discount factor (γ), exploration rate (ϵ) and its decay schedule, as well as network architecture parameters (e.g., number of layers and neurons in the Deep Q-Learning setting).

By combining reward function selection with rigorous hyperparameter optimization, we aim to obtain a more reliable and generalizable reinforcement learning agent. Ultimately, this approach should reduce the instability observed when transaction costs are introduced and lead to a policy that remains robust under realistic market conditions.

Chapter 15

Week 15

Contents

15.1 The Best Reward

Last week, we investigated which reward function would be most effective for our Q-Learning model in learning the dynamics of a given market. To evaluate the best reward, we developed a script that simultaneously launches 100 training sessions, allowing us to gather statistical measures for comparing each reward. All training sessions are conducted on the same dataset.

Since one batch of 100 training sessions takes an entire day to complete, we restricted our experiments to an upward trending market : the Apple stock market from 2010 to 2024. Each batch will be compared using three main criteria : the average equity curve output, the reward function output and the distribution of final profits. These metrics will help assess the robustness of each reward function.

In addition, we will analyze the maximum drawdown observed during training. To achieve this, we will compare the drawdown estimates and their distribution across all training batches.

15.1.1 Reward on Raw Portfolio Returns

The "Raw Portfolio Return" reward is defined as the direct change in the portfolio value over time. In this setup, the reward signal is proportional to the basic evolution of the equity curve, without any normalization or transformation. This approach provides the most straightforward way to measure performance, as the agent's decisions are directly linked to increases or decreases in portfolio value.

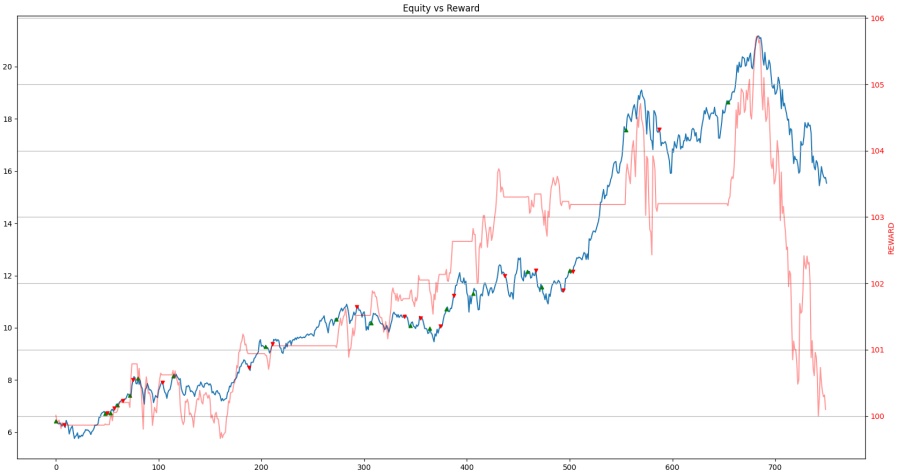


Figure 15.1: Reward based on the portfolio value.

By construction, this type of reward strongly reflects the underlying dynamics of the market. If the agent is able to capture upward trends and avoid downturns, the reward signal will naturally increase, reinforcing profitable strategies. Conversely, poor decisions that reduce the portfolio value will be immediately penalized.

One expected advantage of this reward is that it encourages the agent to directly maximize final profit, as the cumulative reward is aligned with absolute portfolio growth. However, this alignment also comes with limitations. Since the reward is unscaled, extreme variations in portfolio value may dominate the training signal, potentially leading to instability or overfitting. Additionally, because the reward does not explicitly account for risk-adjusted performance, the agent may adopt overly aggressive strategies that expose the portfolio to large drawdowns.

In summary, the raw portfolio return reward is intuitive and easy to implement, making it a natural baseline for comparison. It emphasizes pure profitability and offers a clear benchmark against which more sophisticated reward functions can be evaluated, particularly those designed to balance profit with risk management.

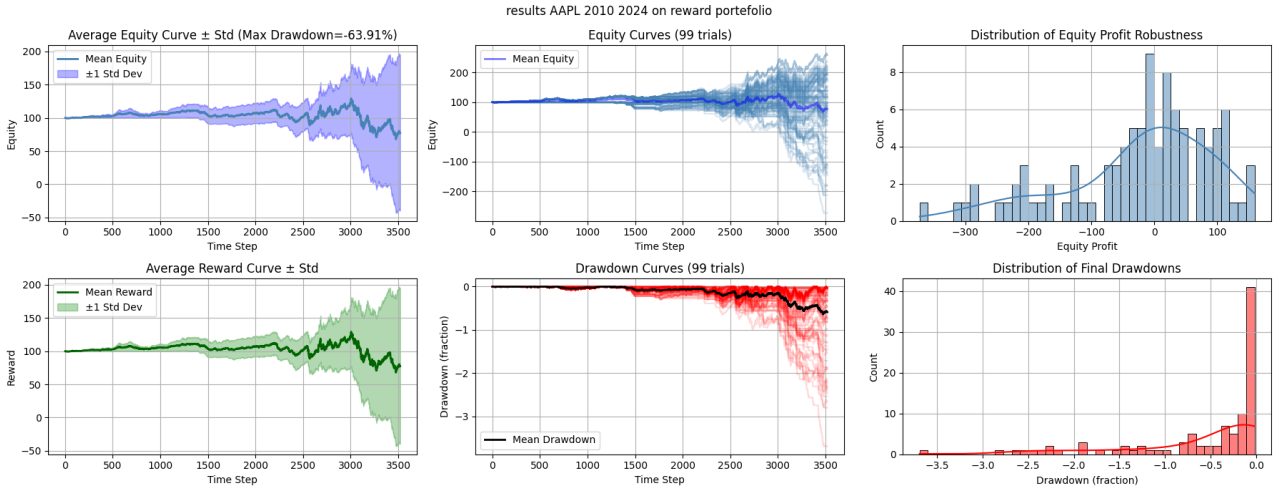


Figure 15.2: Analysis of the portfolio value reward.

The results indicate that using the raw portfolio value as a reward leads to a model with inconsistent performance. The profit distribution is centered around zero, suggesting that in most cases the agent's probability of generating profit is close to random (approximately 50%). Such behavior is undesirable for a trading system, as it implies that the model is not consistently learning profitable strategies.

In terms of risk, the analysis reveals a maximum drawdown of −63%, while the average drawdown remains close to zero as confirmed by the distribution plot. This pattern suggests that although many training runs do not experience large losses, when drawdowns occur (−18%), they can be severe and catastrophic for the portfolio.

One of the key issues with this type of reward is that it provides the model with a very limited learning signal. Since the reward is tied only to the absolute portfolio value at each step, the agent struggles to evaluate the true contribution of individual actions within the broader market dynamics. In other words, the model cannot easily distinguish whether a specific action was beneficial or harmful in the long run, especially when short-term fluctuations dominate the reward signal.

To address this limitation, it would be necessary to design reward functions that place greater emphasis on the relative impact of actions, for example by considering both past performance and expected future outcomes. Overall, while the raw portfolio reward offers a direct link to profitability, its lack of sensitivity to action-level contributions and its exposure to extreme drawdowns make it an unreliable choice for robust trading strategies.

15.1.2 Reward on Portfolio Differences

From the previous experiment, we observed that the raw portfolio value as a reward does not adequately represent the impact of individual actions. To better capture this impact, we introduce the "Portfolio Difference" reward. This reward is defined as the change in portfolio value between two consecutive steps ; the difference between the portfolio value after the current action and that after the previous action.

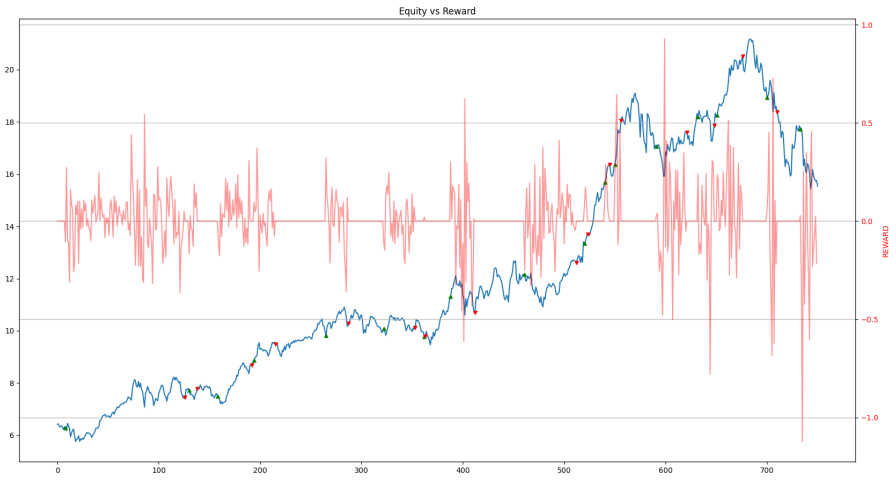


Figure 15.3: Reward based on portfolio differences.

By construction, this reward function places greater emphasis on the immediate consequence of a decision. A profitable action will yield a positive reward, while a poor action will be penalized instantly. This local sensitivity is expected to increase the exploitation capacity of the learning agent, as it provides clearer feedback on which decisions are beneficial at each time step.

$$\mathscr{R} = x_{portfolio}[t] - x_{portfolio}[t-1]$$

However, the portfolio difference reward also introduces certain limitations. Since it focuses only on the relative change from one step to the next, it does not directly optimize for long-term profitability. In other words, the agent may learn to favor short-term improvements in portfolio value without necessarily maximizing the final cumulative profit. Furthermore, this reward function may encourage reactive rather than strategic behavior, as the agent optimizes primarily with respect to the most recent outcome rather than considering broader market dynamics.

Despite these drawbacks, the portfolio difference reward offers a valuable intermediate step in reward design. It provides sharper feedback for the agent's actions compared to raw portfolio value, making it easier to distinguish between good and bad decisions in the short run. This reward can therefore serve as a useful benchmark, especially when combined or compared with more sophisticated functions that explicitly incorporate risk management and long-term profit objectives.

The results show that the mean equity curve is improved compared to the raw portfolio reward, with fewer portfolios ending below 0% profit, let us notice that the average equity curve is always above 0%. This improvement is also reflected in the profit distribution, which centers around 25%, suggesting that the portfolio difference reward produces more robust outcomes than the previous approach.

In terms of risk, the average drawdown remains relatively similar and stable, indicating that the strategy is not yet explicitly accounting for risk management. Nevertheless, we observe a significant improvement in the maximum drawdown, which decreases to -25.06%. This indicates that while large losses can still occur, their severity has been substantially reduced compared to the raw portfolio reward.

However, as noted earlier, this reward design does not explicitly maximize the final portfolio value. Instead, it

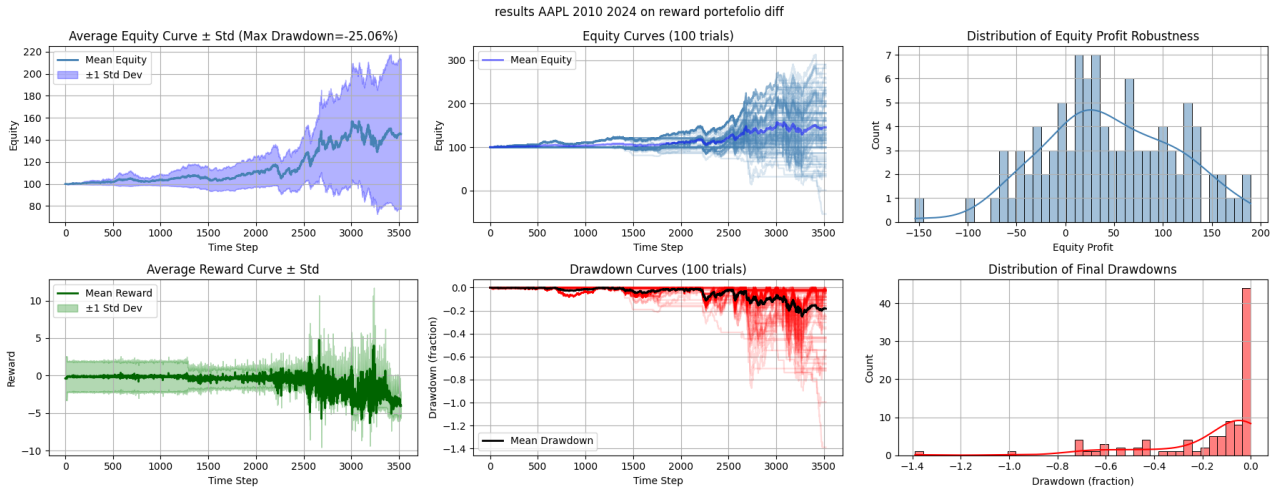


Figure 15.4: Analysis of the portfolio value reward

emphasizes the relative impact of each action with respect to the previous step. While this provides clearer short-term learning signals, it may limit the agent's ability to optimize for long-term growth and overall profitability.

These findings suggest that while the portfolio difference reward improves robustness and reduces extreme losses, it remains incomplete as a standalone solution. A potential improvement would be to design a hybrid reward that combines short-term action sensitivity with long-term profitability objectives. Such a function could better capture the dynamic relationship between individual decisions and the final portfolio value, thereby aligning immediate learning signals with strategic performance goals.

15.1.3 Reward on Slope Sign

Building on the previous reward functions, we introduce the "Slope Sign" reward, which attempts to combine the benefits of both raw portfolio returns and portfolio differences. The idea is to evaluate the direction of the equity curve by computing the slope between the portfolio value after the previous action and the current portfolio value. The sign of this slope (positive or negative) is then multiplied by the current portfolio value to generate the reward signal. In this way, profitable actions are reinforced, while unprofitable ones are directly penalized.

The main advantage of this reward function is that it provides clear and immediate feedback regarding the quality of an action. Good decisions that align with upward movements in the equity curve are strongly rewarded, while poor decisions are heavily punished. This sharp distinction creates a larger gap between successful and unsuccessful actions, which can accelerate the learning process and improve the agent's ability to exploit profitable opportunities.

$$\mathscr{R} = \text{sign}\left(\frac{\delta}{\delta t}\left(x_{portfolio}[t_{\text{last action}}] - x_{portefolio}[t]\right)\right) * x_{portfolio}[t]$$

However, this approach also comes with limitations. Since the slope sign is highly sensitive to short-term fluctuations, the reward signal can be noisy and unstable. Market oscillations that do not reflect meaningful long-term trends may mislead the agent, causing it to overreact to minor changes rather than learning robust strategies. In addition, because this reward focuses primarily on the direction of the immediate slope, it may

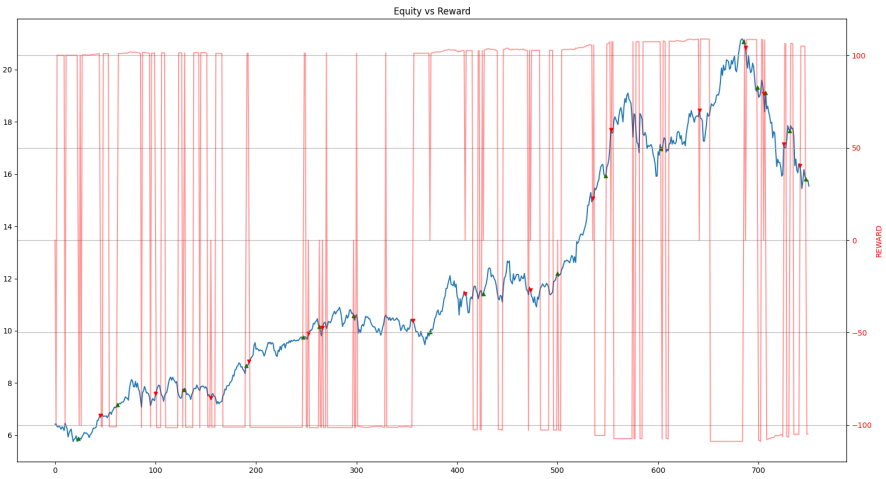


Figure 15.5: Reward based on the portfolio value and its derivative sign

neglect broader portfolio dynamics such as cumulative profit growth or drawdown control.

In summary, the slope sign reward introduces a more decisive feedback mechanism compared to previous designs, enhancing the distinction between good and bad actions. Nevertheless, its sensitivity to market oscillations makes it less reliable in volatile environments.

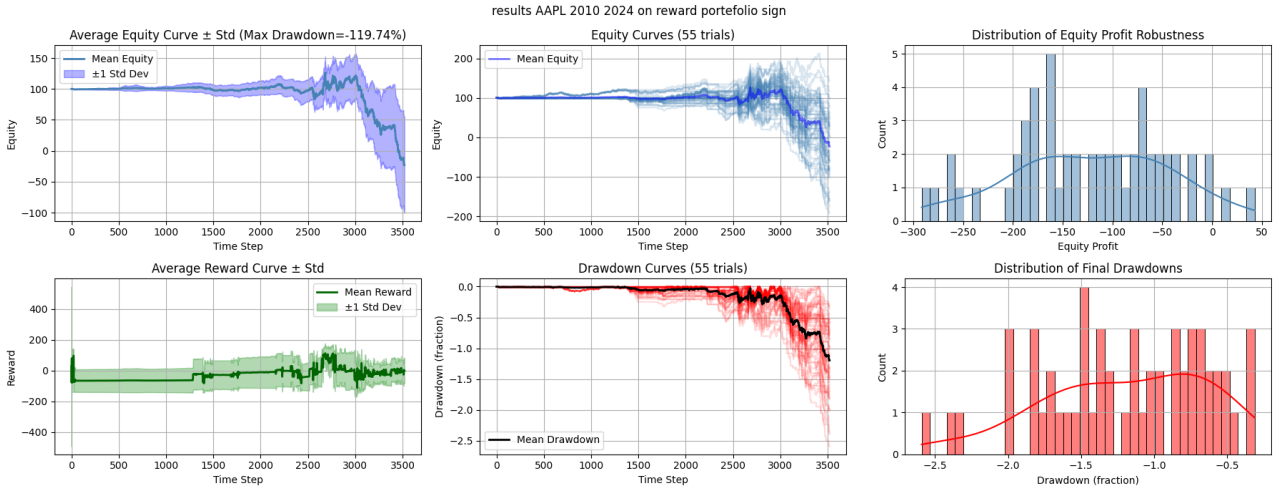


Figure 15.6: Analysis of the portfolio value reward

This type of reward performs poorly. The likely reason is the high frequency of changes in the portfolio, which results in excessive volatility. Consequently, we observe a high drawdown and negative profits across nearly all trials. Due to these unfavorable outcomes, this reward function will not be used in subsequent analyses.

15.1.4 Reward on Portfolio Value and Direct Slope

To reduce the instability introduced by using only the slope sign, we extend the reward definition by incorporating the actual slope value. Specifically, the reward is computed as the sum of the current portfolio value and the slope of the equity curve, scaled by the portfolio value at each step. This formulation preserves the influence of market trends while reducing the sensitivity to minor oscillations.

With this design, the reward function captures both profitability and directional momentum. On one hand,

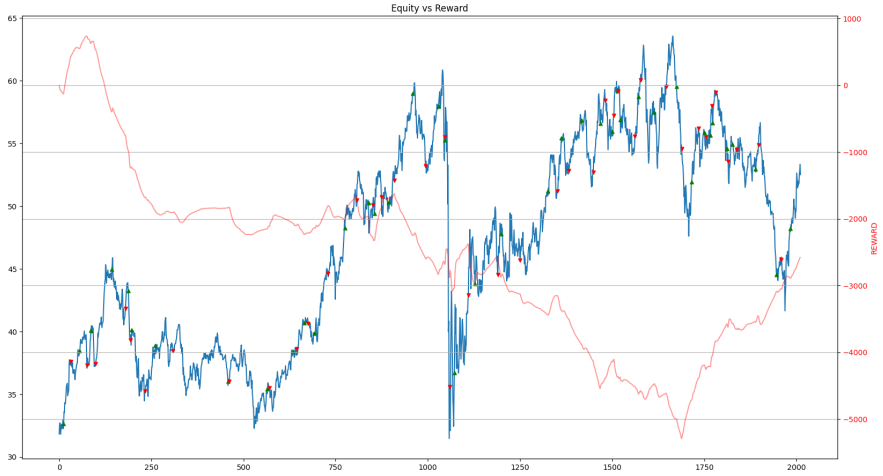


Figure 15.7: Reward combining portfolio value and its derivative.

the portfolio value ensures that the agent remains focused on long-term growth. On the other hand, the slope provides immediate feedback on whether the most recent action is aligned with the current market trend. Wrong decisions are punished proportionally to the negative slope, while good decisions are reinforced when the trend is favorable.

$$\mathscr{R} = \frac{\delta}{\delta t}\left(x_{portfolio}[t_{\text{last action}}] - x_{portefolio}[t]\right) * x_{portfolio}[t]$$

The intuition behind this approach is inspired by human-like reasoning in trading : investors do not only look at their total portfolio value but also assess whether their most recent actions are still performing well relative to the ongoing market movement. By combining both perspectives, this reward function aims to balance long-term profit maximization with short-term action validation.

However, this method is not without limitations. The slope remains sensitive to volatility, and sudden short-term fluctuations can still distort the reward signal. Additionally, because the reward is scaled by the portfolio value, large portfolios may produce exaggerated feedback compared to smaller ones, potentially leading to unstable learning dynamics. A possible refinement would be to normalize the slope contribution or apply smoothing techniques to better filter out noise from transient oscillations.

In summary, the portfolio value and direct slope reward offers a more balanced feedback mechanism than previous formulations, combining overall profitability with immediate trend sensitivity. It provides a promising middle ground, though further adjustments may be required to improve stability in highly volatile markets.

As we can see, the profit is often positive ; however, there are also significant periods of negative returns. This suggests that the reward function is not as robust as initially expected when compared to the second approach. Moreover, the maximum drawdown is particularly severe (-87%), which is considerably higher than what we typically observed in previous experiments. Finally, the strategy tends to converge to a profit level close to 50% of the original wallet value.

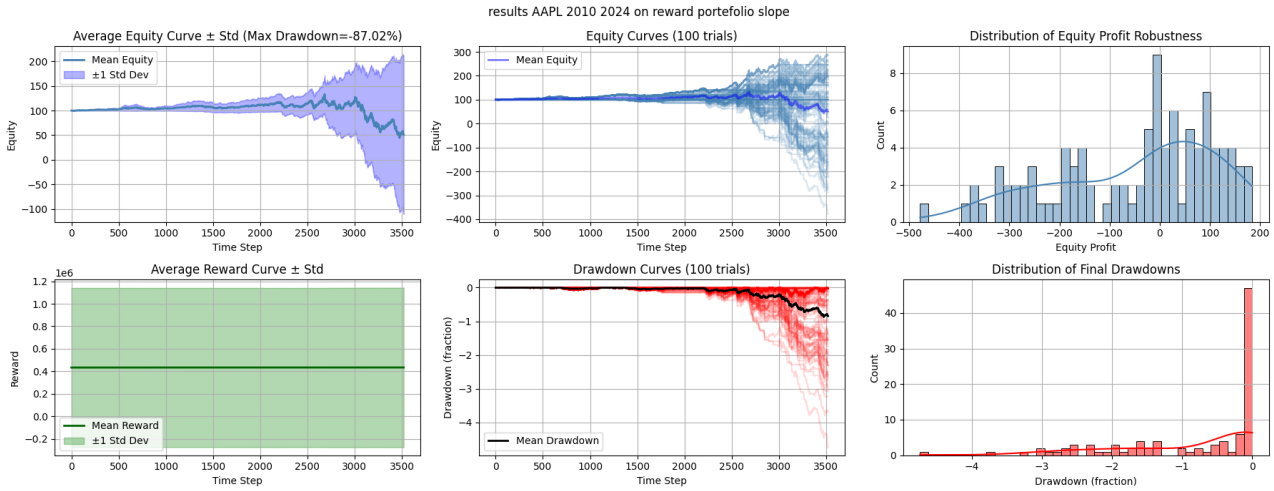


Figure 15.8: Analysis of the portfolio value reward

15.1.5 Reward on Portfolio Value and Direct Slope (Logarithmic Function)

Having established a good balance between portfolio value and immediate action feedback, we introduce a logarithmic adjustment to further refine the reward. This modification allows us to leverage the punishment depending on the magnitude of the slope, emphasizing significant deviations while reducing sensitivity to minor fluctuations.

Figure 15.9: Reward with logarithmic adjustment.

The core idea behind applying a logarithmic function is to amplify the impact of large positive or negative slopes. Large upward movements are rewarded more strongly, encouraging the agent to capitalize on strong trends. Conversely, steep downward movements are penalized more severely, ensuring that poor decisions with major negative consequences are recognized and discouraged.

$$\mathscr{R} = \log\left(\frac{\delta}{\delta t}\left(x_{portfolio}[t_{\text{last action}}] - x_{portefolio}[t]\right) * x_{portfolio}[t]\right)$$

By scaling the slope contribution logarithmically, the agent receives a more nuanced learning signal. Small fluctuations, which might otherwise generate noise in the reward, have a reduced impact, while extreme events exert a proportionally greater influence. This approach helps the agent focus on meaningful actions

that significantly affect portfolio growth, rather than reacting to minor market oscillations.

In summary, the logarithmic adjustment enhances the previous reward formulation by improving sensitivity to important market movements while dampening the effect of minor, potentially misleading fluctuations. This makes the reward signal more informative and robust, leading to better alignment between immediate action evaluation and long-term portfolio performance.

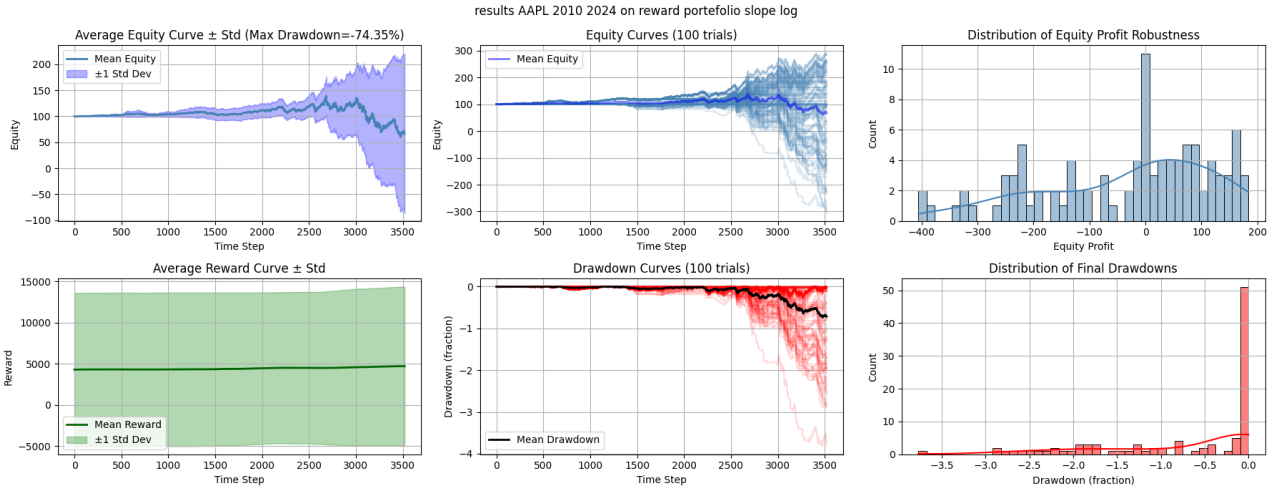


Figure 15.10: Analysis of the portfolio value reward

This new reward yields results that are broadly similar to the previous case, but with slight improvements. In particular, the maximum drawdown is reduced to −74%, and the profit distribution is more balanced, indicating greater robustness. Additionally, the proportion of positive outcomes has increased, with profits tending to stabilize around 70% of the original wallet value.

Chapter 16

Week 16

Contents

16.1 VIX index experiments

This week we focused our agent to work on the VIX index. The **VIX Index**, also known as the *CBOE Volatility Index*, is a widely followed measure of market expectations of near-term volatility conveyed by S&P 500 stock index option prices. Introduced by the Chicago Board Options Exchange (CBOE) in 1993, the VIX is often referred to as the "fear gauge" because it tends to rise when markets become uncertain or experience stress.

Mathematically, the VIX represents the market's expectation of the annualized volatility over the next 30 days, derived from a wide range of S&P 500 index options. Higher VIX values indicate greater expected market volatility, while lower values suggest calmer market conditions.

investors and risk managers use the VIX as a tool for hedging, portfolio allocation, and to gauge market sentiment. It is widely used in both academic research and practical trading strategies to understand and anticipate market risk.

16.1.1 Finding the best reward for VIX index

As we did previously, we need to find the best reward for the VIX index. We will keep the same kind of reward than previously and run a 100 curves for each reward to get the best possible reward.

Portfolio Returns Reward

This reward is based on the return of the porfolio which is suppose to maximize the final value of the equity curves associated to the trained agent.

$$\mathscr{R} = x_{portfolio}$$

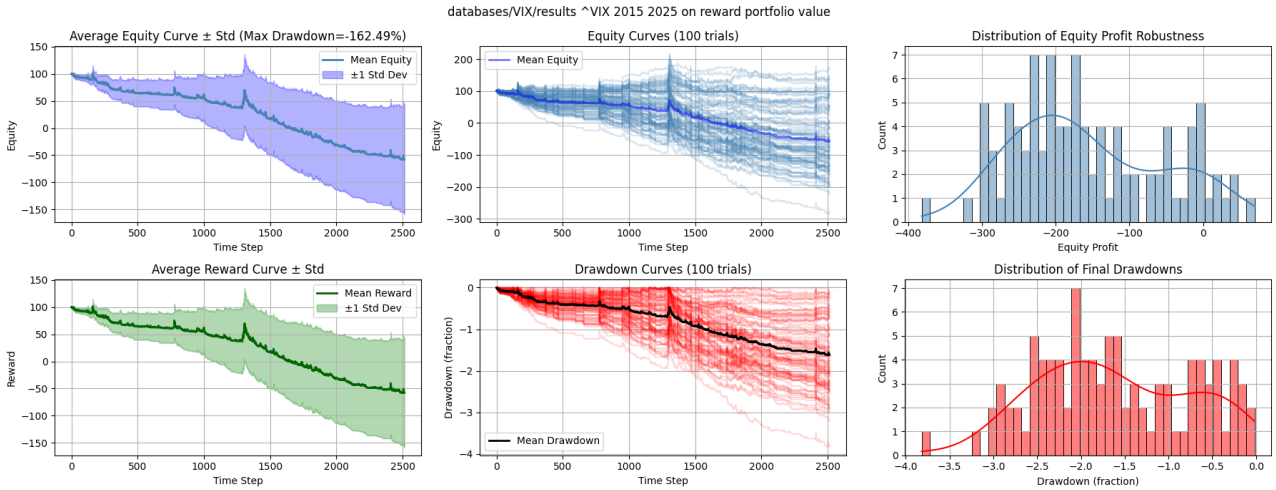


Figure 16.1: Analysis of the portfolio value reward

This reward will not be considered as it has inly negative profit.

Portfolio Difference Reward

$$\mathscr{R} = x_{portfolio}[t] - x_{portfolio}[t-1]$$

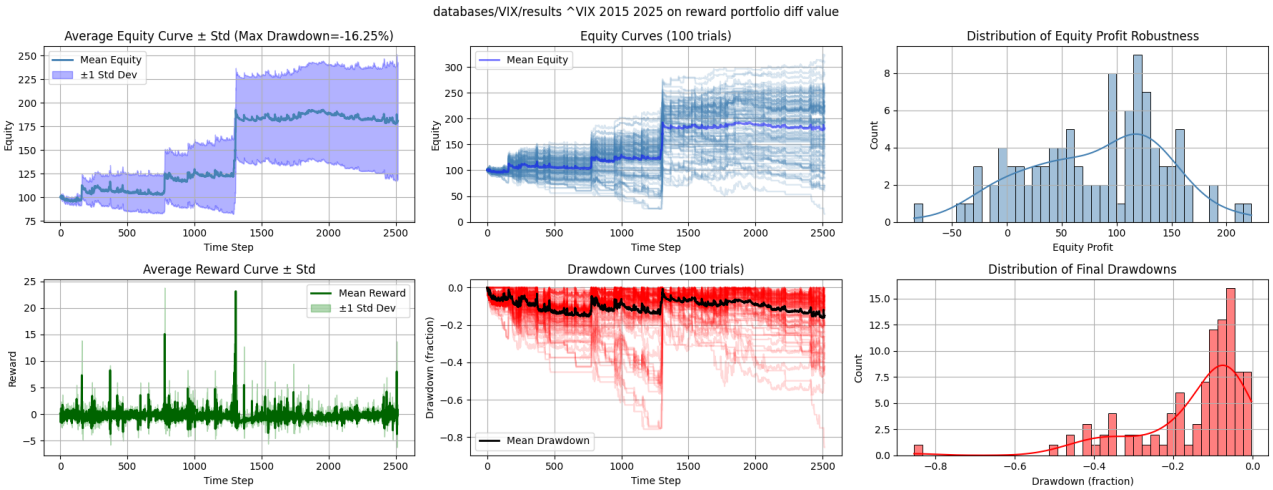


Figure 16.2: Analysis of the portfolio value reward

This reward is a good candidate since it has the best trained agent profit that reached almost 300%. Also, the drawdown is very low in the distribution graph.

Slope Sign Reward

$$\mathscr{R} = \text{sign}\left(\frac{\delta}{\delta t}\left(x_{portfolio}[t_{\text{last action}}] - x_{portefolio}[t]\right)\right) * x_{portfolio}[t]$$

The slope sign reward will not be considered since it has to many negative profit.

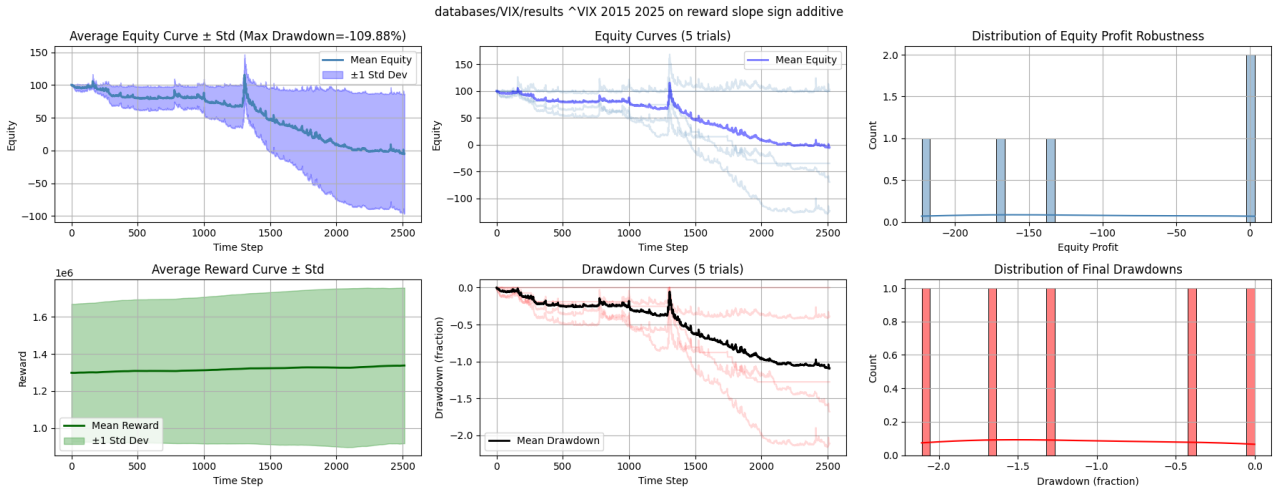


Figure 16.3: Analysis of the portfolio value reward

Portfolio Value and Direct Slope Reward

$$\mathscr{R} = \frac{\delta}{\delta t}\left(x_{portfolio}[t_{\text{last action}}] - x_{portefolio}[t]\right) * x_{portfolio}[t]$$

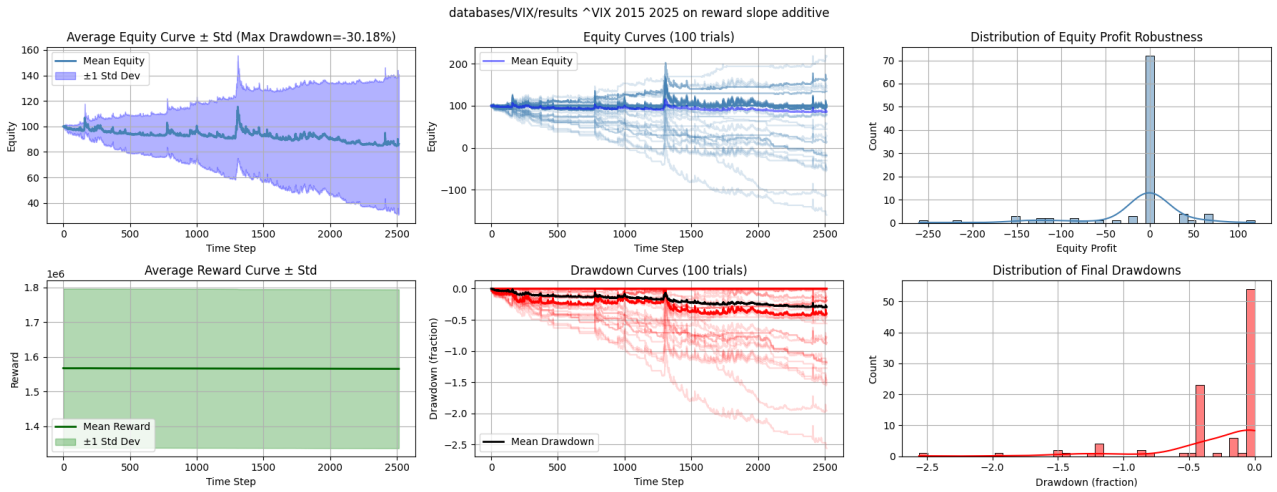


Figure 16.4: Analysis of the portfolio value reward

This reward has an excellent drawdown compared to the last good candidates but the best agent is only reaching 200% of profit which is worse than the previous candidates.

As reward we will use the portfolio difference reward since it has the best profit output and drawdown, which is not the best out of all agents, that is good enough to be considered in real life scenarios.

16.1.2 Fine-Tuning Q-Learning for the VIX Index

Now we found the best reward, we need to fine-tune the Q-Learning agent to optimize the Q-matrix outputs for making decisions on a testing dataset. To achieve this, we perform approximately 120 trials of the agent in the same environment in order to identify the best set of hyperparameters across all trials.

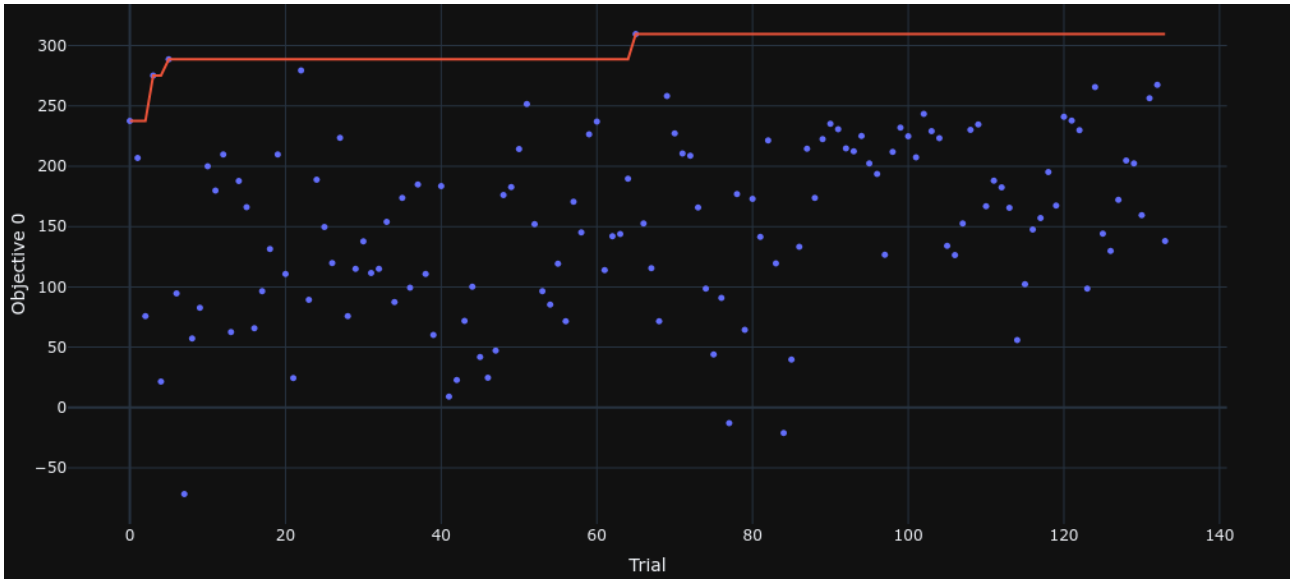


Figure 16.5: Optuna output for trials

As observed, the best trial reached a profit exceeding 300%, indicating that the corresponding parameters are capable of training an agent with strong performance.

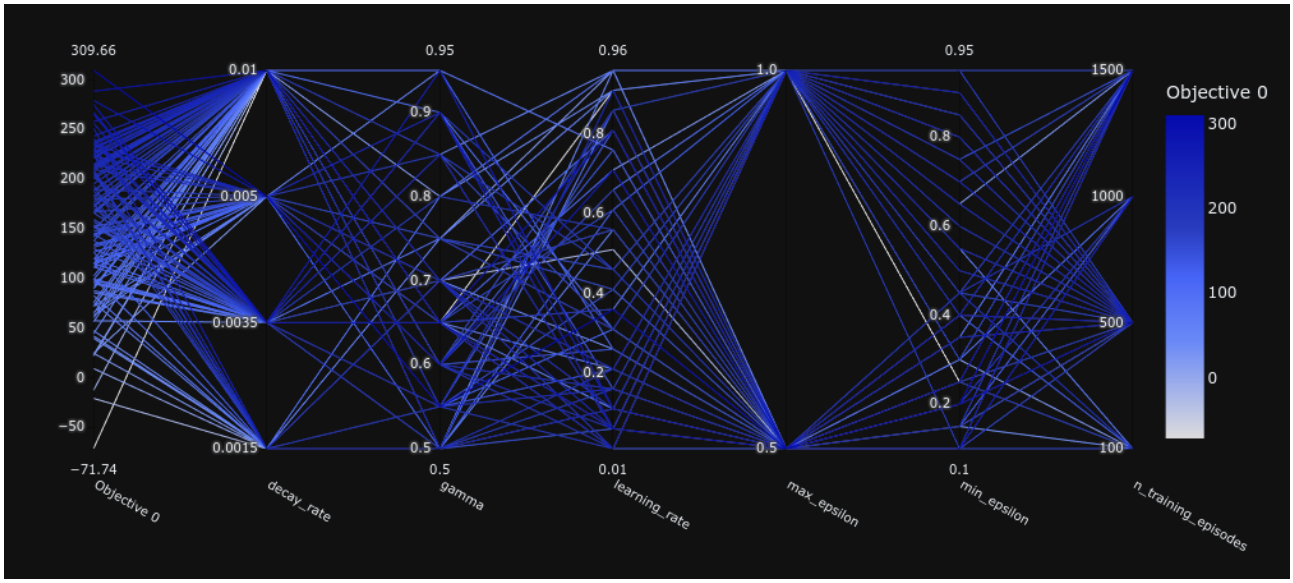


Figure 16.6: Optuna parameter optimization results

However, due to the stochastic nature of the greedy policy, agents trained with the same parameters may produce different returns. This means that the 300% profit is not guaranteed in subsequent training runs. To address this variability, we train a set of 100 agents to compare their performance and select the best among them.

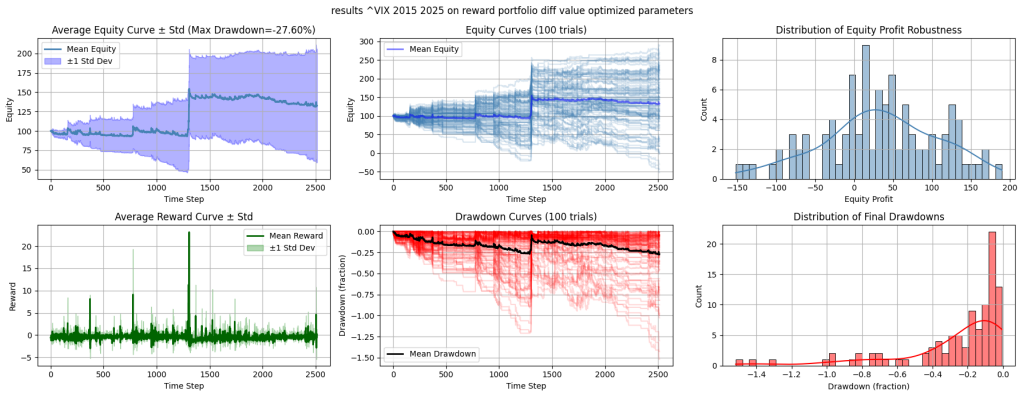


Figure 16.7: Performance distribution of agents trained with the best parameters

From the figure, we observe a stable drawdown distribution around 5% for most trainings, with an average maximum drawdown of −24%. The profit distribution centers around 25%, which is favorable compared to previous market analyses. Importantly, the best trial out of the 100 trained agents achieved nearly 200% profit.

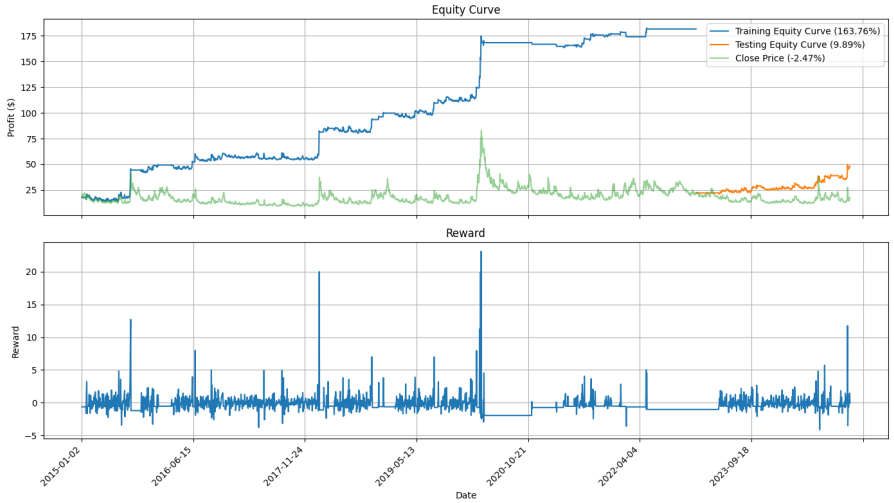


Figure 16.8: Best performing agent from the 100 training trials

The best trial achieved a training profit of 164% over 8 years, corresponding to an annualized return of approximately 20%, and a tested profit of 10% over 2 years. Observations indicate that the agent captures significant profits while avoiding major market downturns. We can now proceed to train a Deep Q-Learning agent, replacing the basic greedy strategy, to further enhance model decision-making.

16.2 Deep Q-Learning Integration

In previous work, we introduced two Deep Q-Learning agents. The first uses a GPT-based transformer model sourced from Hugging Face, while the second leverages custom model layers designed according to research on constructing models specifically tailored to market actions.

The GPT transformer model had already achieved promising results but exhibited unpredictable and unstable behavior. We addressed this issue by refining the environment and preventing illegal actions during training, which significantly improved stability. The "homemade" agent incorporates multi-head attention layers and has been developed, although it has not yet been tested. We were, however, able to evaluate the impact of the new reward function and parameter settings on the GPT transformer agent :

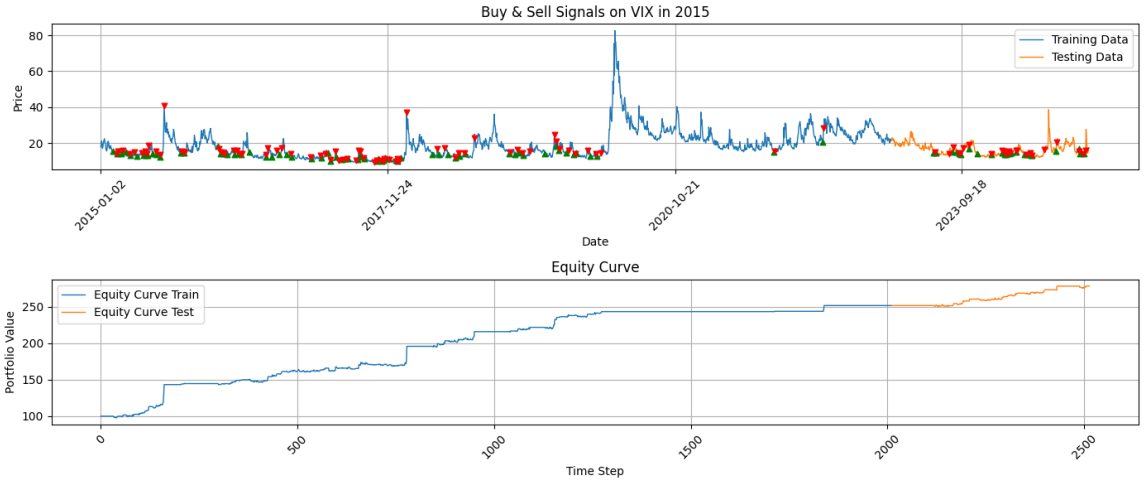


Figure 16.9: Deep Q-Learning Agent (non optimized) results

This agent is limiting the drawdown by itself since it has a very stable uptrend equity curve. We have global profit that is slightly better than the previous basic Q-Learning agent by 1% with a global profit (Training + Testing dataset) of 175%.

	Q-Learning	**Deep Q-Learning (not optimized)**
Agent Training Profit	163%	151%
Agent Testing Profit	10%	24%
Global Profit	173%	175%

Let's noticed that this Deep Q-Learning agent has not been optimized since that for 500 epsiodes training we have to sepdn 40 minutes and the training script needs to be optimized since it is using a lot of RAM. After fixing the RAM issue (if it is possible), we will be able to train multiple agent in the same time to fine-tune the model using optuna. For now the new agent has the best output from every model we used previously, but we suggest that the agent could get better performance after optimization.wallet value.

Chapter 17

Week 17

Contents

17.1 Encoder Transformer

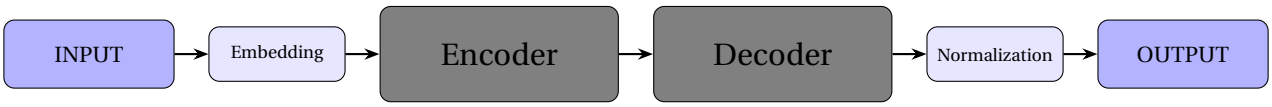


Figure 17.1: Encoder-Decoder Stack of the transformer model

Few weeks ago, we read the paper written in 2023 by Pranav Agarwal (École de Technologie Supérieure/ Mila, Canada), Aamer Abdul Rahman (École de Technologie Supérieure/Mila, Canada), Pierre-Luc St-Charles (Mila, Applied ML Research Team, Canada), Simon J.D. Prince (University of Bath, United Kingdom) and Samira Ebrahimi Kahou (École de Technologie Supérieure/Mila/CIFAR, Canada), provides a comprehensive overview of this rapidly developing research area. The paper reviews how transformers have been adapted for reinforcement learning, highlights emerging architectures and methodologies, and categorizes them according to their applications. It also discusses empirical results, key challenges such as scalability and sample efficiency, and potential future research directions.

By situating transformer-based reinforcement learning within the broader landscape of sequence modeling and decision-making, this survey aims to deepen our understanding of both the opportunities and limitations of applying transformers in RL contexts. It serves as a reference point for researchers and practitioners seeking to build on the foundations of this promising intersection of fields. While transformers have demonstrated remarkable success across natural language processing, computer vision, and sequential prediction tasks, their application to financial trading is particularly compelling, given the complex, dynamic and highly stochastic nature of financial markets.

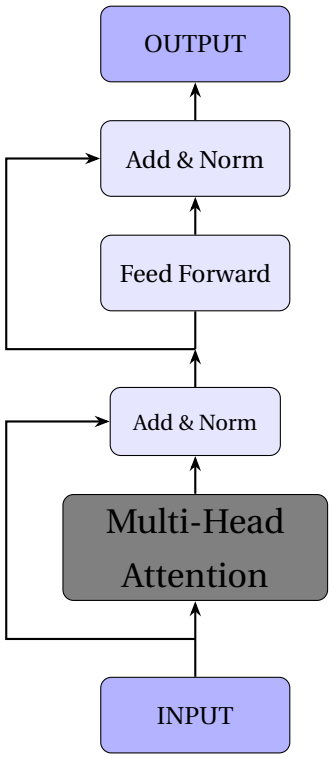


Figure 17.2: Encoder stack

In the context of trading, the model architecture can be divided into two core components. The encoder, built upon a multi-head attention mechanism, is responsible for capturing the long-term dependencies and intricate temporal patterns in financial time-series data. This design enables the model to represent market dynamics more effectively than traditional recurrent architectures, which often struggle with long-horizon dependencies. The decoder, on the other hand, is structured with varying output dimensions tailored to the specific prediction or decision-making task at hand, such as making an action.

By combining these components, the transformer-based RL framework is capable of not only learning sequential representations of market states but also optimizing decision policies that directly impact trading performance. This dual capability provides an advantage over conventional predictive models, which typically focus only on forecasting without explicitly linking predictions to downstream decision-making. Moreover, reinforcement learning introduces a feedback loop that allows the agent to continuously adapt its strategy in response to evolving market conditions, thereby improving its robustness and generalization ability.

17.2 First Encoder Transformer results

We performed 100 trials with Optuna to compare the best-performing agents. These trials optimized the hyperparameters of the reinforcement learning framework, including the learning rate, batch size, discount factor, and replay buffer capacity. By systematically searching through this space, Optuna was able to identify combinations that improved both convergence speed and trading performance.

The training setup, as illustrated in our implementation, involved running the agent for 500 episodes with a training size of 80% of the dataset. A smaller learning rate (1×10^{-4}) was chosen to ensure stability during optimization, while the discount factor ($\gamma = 0.99$) preserved the importance of long-term rewards. To encourage exploration in the early stages, the policy started with a maximum epsilon value of 1.0, gradually decaying to 0.05 with a rate of 0.005. This ensured that the agent initially explored the market environment widely before converging toward exploitation of profitable strategies.

Additionally, a large replay buffer of 100,000 transitions was used to stabilize learning by diversifying the sampled experiences, while a batch size of 128 allowed for more robust gradient updates. The target network was updated every 500 steps, and learning occurred every 4 steps to maintain a balance between computational efficiency and training stability. A warm-up phase of 5,000 random steps was also introduced to populate the replay buffer before policy learning began.

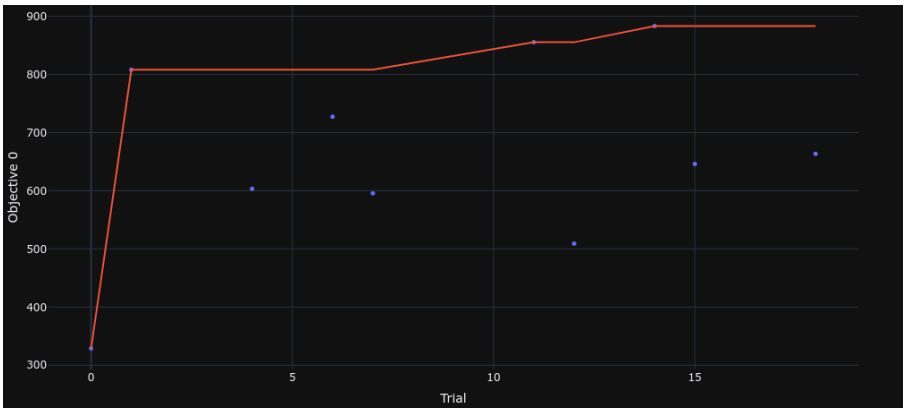


Figure 17.3: Encoder Transformer Best optuna trials

This configuration provided a strong baseline for comparing the agents optimized by Optuna. The results of these experiments highlighted which hyperparameter settings led to consistent profitability, reduced volatility in returns, and improved sample efficiency. The following section presents a detailed analysis of the performance metrics obtained from these trials, including cumulative returns, Sharpe ratio and maximum drawdown.

Finally, we found the best trail to have more than 5% of profit while previously we had a profit of 10%. The basic previous nodel seems to be better. However let's run 100 trials with the optimized hyperparameterization to get a better understanding of the best agent. For example, there could be a drawdown anomaly with more or less frequency of actions taken.

17.2.1 Getting the best DeepQLearning agent

Out of a 100 trials, we found the best agent to make 5% of profit, this is two times less than the previous basic QLearning agent. However, the drawdis especially low with a Maximum Drawdown of 3% which is very low. Also, the drawdown distribution is even lower with an average distribution lower than 1.5%.

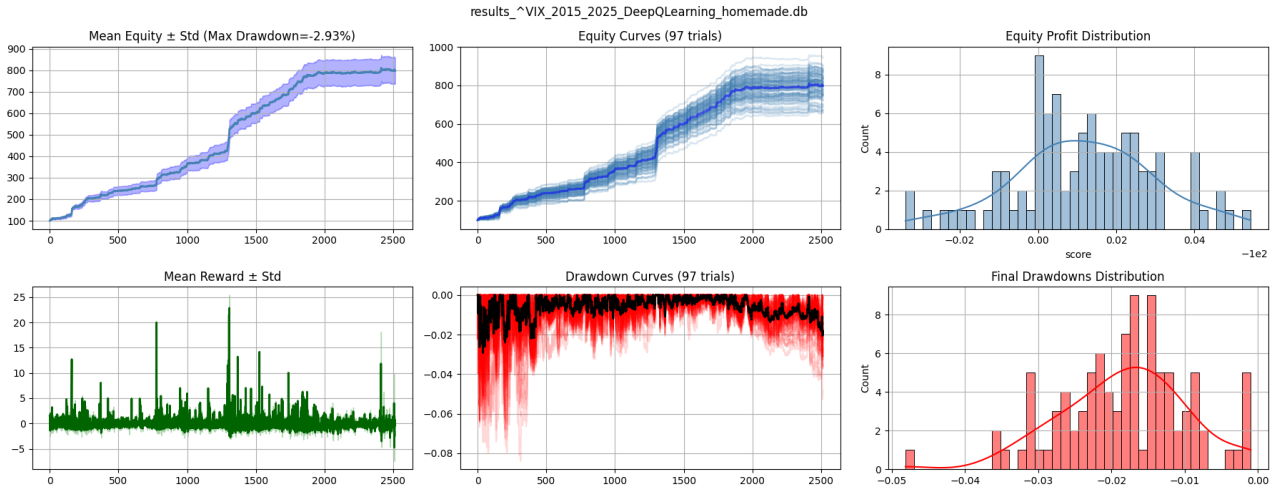


Figure 17.4: Bests Encoder Transformer

As we can see, the agent appears to capture the underlying market dynamics effectively during the training phase. When evaluated on the test set, the model demonstrates an ability to identify and exploit the main market peaks, thereby securing the majority of potential profits. At the same time, it maintains a relatively low drawdown, which is a critical factor in assessing the reliability of trading strategies. This balance between profit generation and risk management suggests that the agent is not only capable of learning profitable patterns but also of adopting a conservative behavior that mitigates excessive losses.

While this conservative approach may limit the overall profit potential compared to highly aggressive strategies, it contributes to the robustness and stability of the trading policy. In practice, such behavior is often more desirable, as it prioritizes capital preservation and consistency over occasional large gains accompanied by higher risk. These results therefore highlight the suitability of transformer-based reinforcement learning agents for financial applications where reliability and risk control are of paramount importance.

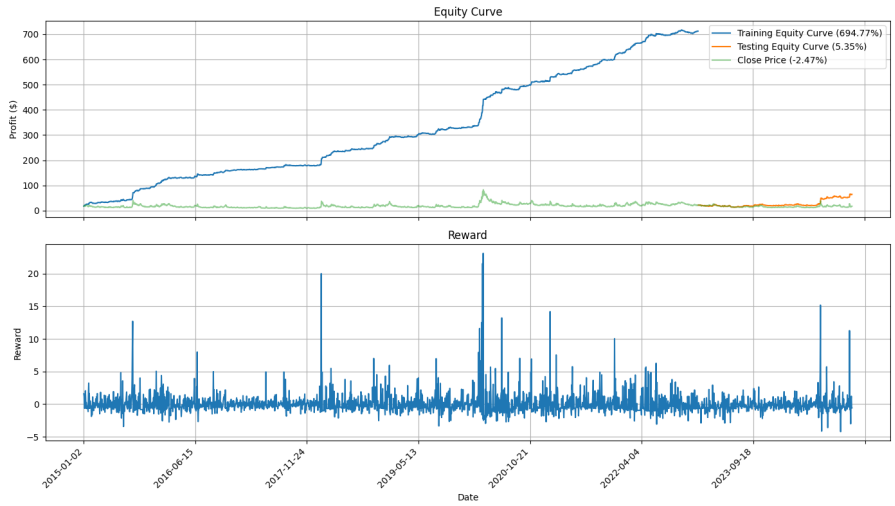


Figure 17.5: Encoder Transformer Results

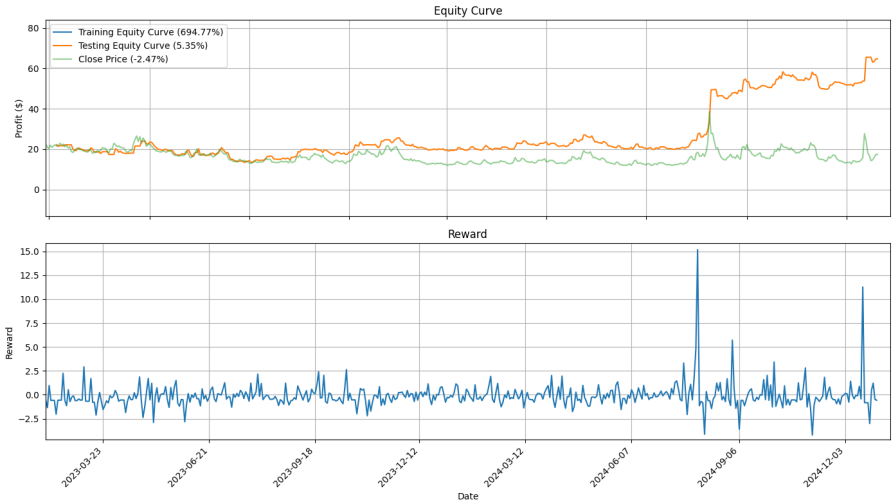


Figure 17.6: Encoder Transformer Results (test focus)

17.2.2 Getting the best GPT DeepQLearning agent

To establish a baseline for comparison, we employed a GPT-2 based transformer model specifically adapted for deep reinforcement learning. This model, known as the *edbeeching Decision Transformer*, was originally developed for training decision transformers on the Gym Hopper environment. Since the model was not designed to operate on time-series data, additional fine-tuning was required to enable it to capture market dynamics and generate appropriate trading actions. Through this adaptation, the model was aligned with the requirements of financial datasets, allowing us to evaluate its effectiveness in sequential decision-making tasks within the trading domain.

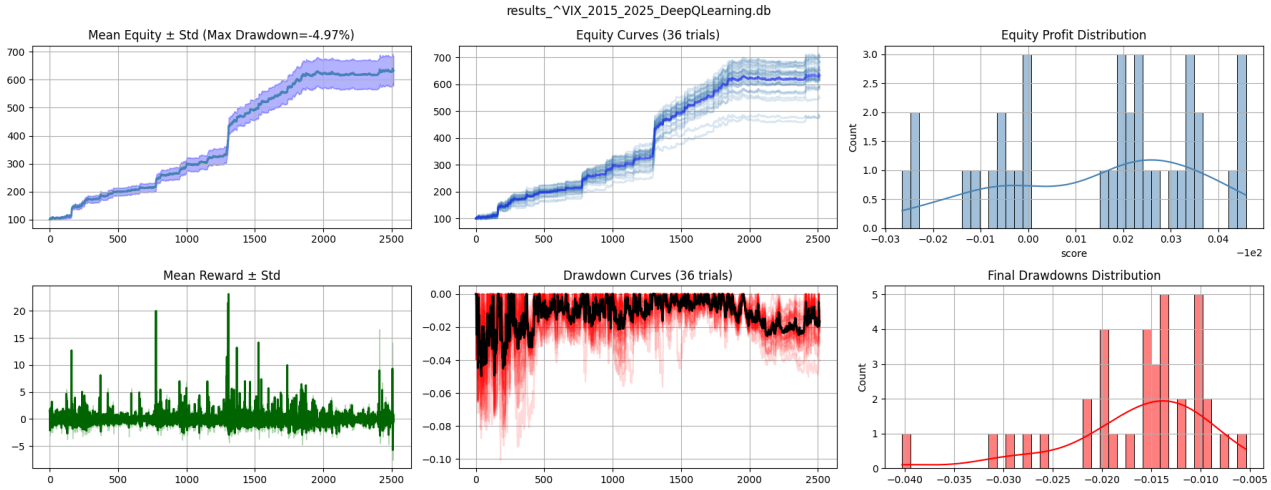


Figure 17.7: GPT Transformer Agents

We observe that this model performs worse than our previously proposed architecture. In particular, the maximum drawdown reaches approximately 5%, while the best trial achieves a test profit of only 4.5%. These results indicate weaker performance compared to our custom encoder-based transformer, which was able to achieve higher profitability while maintaining a lower level of risk. This suggests that the adapted Decision Transformer, although effective in some reinforcement learning benchmarks, struggles to fully capture the complexities of financial time-series data when applied directly to trading tasks.

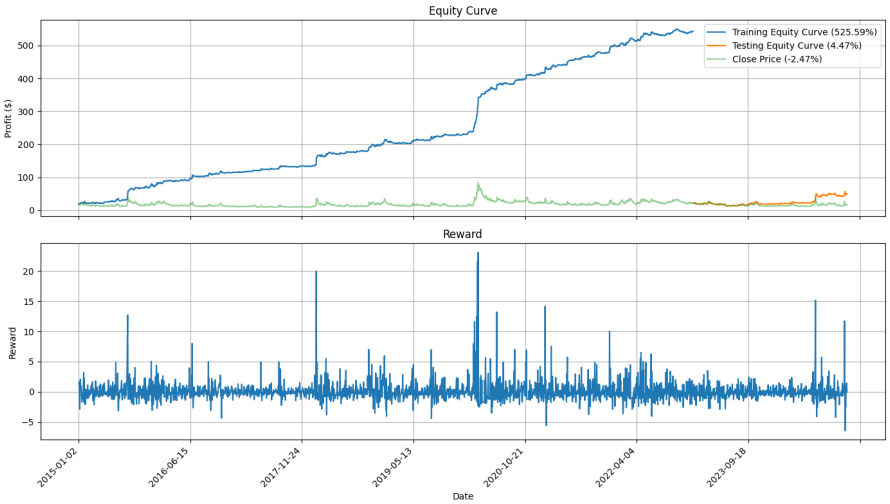


Figure 17.8: Best GPT Transformer Agents

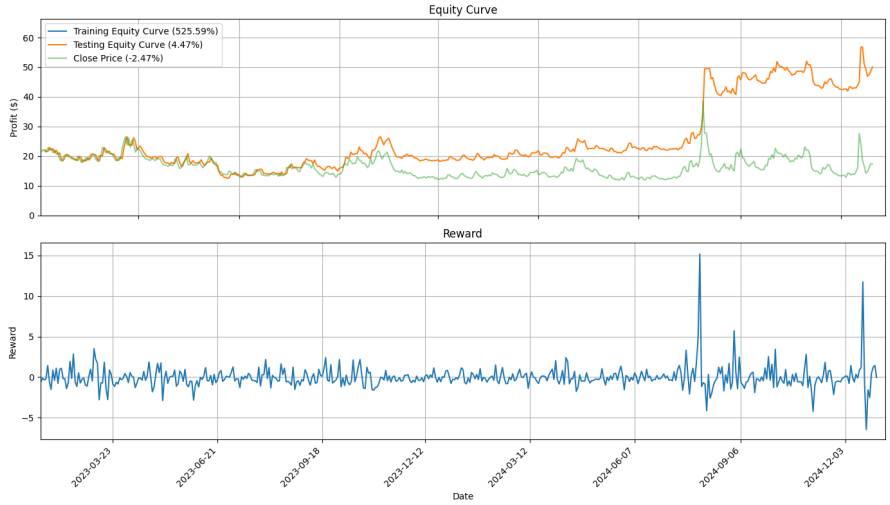


Figure 17.9: Best GPT Transformer Agents (test focus)

17.2.3 DeepQLearning agent comparison

To summarize the experimental results, the Q-Learning model achieved the strongest performance in our tests. However, its evaluation was limited to a relatively short time period, which raises concerns about its ability to capture the dynamics of long-term market behavior. The GPT-based transformer, by contrast, produced weaker results compared to the custom encoder transformer. The encoder transformer emerges as a promising option, as it maintains an exceptionally low drawdown, making it suitable for risk-averse strategies, while still delivering a modest but consistent profit.

	Q-Learning	**Encoder Transformer**	**GPT Transformer**
Agent Training Profit	163.76%	697.77%	525.59%
Agent Testing Profit	9.89%	5.35%	4.47%
Max Drawdown	27.60%	2.93%	4.97%

Figure 17.10: Performance comparison

Chapter 18

Week 18

Contents

18.1 PPO Agent

The PPO and TRPO principles has been studied few weeks ago, now we have a functional Deep Learning agent for market actions, the next goal is to use these models to implement it into the PPO agent. To do so we will breifly introduce the different model layers and how it has been implemented based on the work of the engineer from *Neuralnet.ai*.

Policy gradient and actor-critic methods are the primary tools for reinforcement learning in continuous action spaces, since value-based methods like DQN are unsuitable. However, actor-critic algorithms often suffer from instability: small changes in network parameters can cause large, unexpected shifts in the underlying policy, leading to sudden drops in performance.

To address this, Trust Region Policy Optimization (TRPO) constrains policy updates by limiting the KL divergence between old and new policies, ensuring that improvements are made without deviating too far from previously stable regions of parameter space. Despite its effectiveness, TRPO has limitations, particularly when actor and critic share parameters.

Proximal Policy Optimization (PPO), introduced by OpenAI in 2017, improves on TRPO by using a clipped surrogate objective. This constrains policy changes (within $\sim 20\%$) while maximizing advantageous actions, balancing stability and performance. PPO is simpler to implement, works for both discrete and continuous action spaces, and scales efficiently to parallelized training. It has since become the industry standard for modern reinforcement learning tasks.

18.1.1 Model Explanation

To introduce our model, we present the two main components of the PPO agent: the **Actor** and the **Critic**. It is important to separate these networks so that the critic can effectively evaluate (or "criticize") the predictions of the actor, rather than sharing parameters in a way that could bias the learning process.

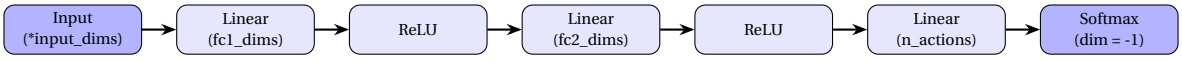


Figure 18.1: Actor network architecture (Softmax output)

The actor network is a multilayer perceptron (MLP) with two hidden layers and ReLU activations. Its final layer uses a Softmax function to produce a probability distribution over the available actions. This distribution represents the policy $\pi_\theta(a \mid s)$ used to sample actions during training and execution.

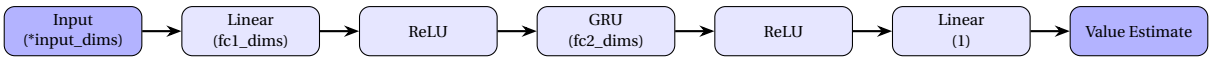


Figure 18.2: Critic network architecture (GRU layout)

The critic network follows a similar structure, but with two key differences:

- It includes a **GRU layer**, which enables it to retain temporal dependencies and capture information from past experiences when evaluating states.
- Its output is a single scalar value, representing the estimated value function $V(s)$. Unlike the actor, the critic does not use a Softmax activation, as it directly predicts state values rather than action probabilities.

Together, the actor and critic form the foundation of the PPO algorithm : the actor proposes actions, while the critic evaluates them to guide more stable learning.

Generalized Advantage Estimation

This method is applied to compute the advantage for each timestep :

$$A_t = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k}, \quad \delta_t = r_t + \gamma V(s_{t+1})(1-d_t) - V(s_t),$$

where γ is the discount factor, λ is the GAE coefficient, and d_t indicates episode termination. The advantages are normalized to reduce variance :

$$\hat{A} = \frac{A - \mu(A)}{\sigma(A) + 10^{-8}}.$$

Policy Update (Actor Loss)

The policy ratio is defined as :

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}.$$

PPO constrains updates by maximizing the clipped surrogate objective:

$$L^{\text{actor}}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] + \beta\, H[\pi_\theta],$$

where ϵ is the clipping parameter and $H[\pi_\theta]$ is the entropy bonus that encourages exploration.

Value Function Update (Critic Loss)

The critic is optimized by minimizing the squared error between predicted and empirical returns :

$$L^{\text{critic}}(\theta) = \mathbb{E}_t \left[\left((\hat{A}_t + V(s_t)) - V_\theta(s_t) \right)^2 \right].$$

Combined Objective The final loss combines both actor and critic objectives :

$$L(\theta) = -L^{\text{actor}}(\theta) + \frac{1}{2} L^{\text{critic}}(\theta).$$

Optimization and Memory Reset

Gradients are backpropagated and used to update both actor and critic networks. Once the updates are complete, the memory buffer is cleared in preparation for the next rollout.

18.1.2 Implementation

In this part we will show how the implementation of the training part of the PPO agent. The following code shows the main training loop of the PPO agent. This function orchestrates interaction with the environment, collects experiences, and triggers learning updates periodically.

Algorithm 5: PPO Agent Training Loop

Input: env, n_games, N, $batch\_size$, α, n_epochs

Output: Trained PPO agent, episode rewards

```
Initialize PPO_agent;
for i ← 1 to n_games
    s ← env.reset();
    done ← False;
    score ← 0;
    while done = False
        A_valid ← env.get_valid_actions();
        (a, probs, v) ← agent.choose_action(s, A_valid);
        (s', r, done, info) ← env.step(a);
        score ← score + r;
        n_steps ← n_steps + 1;
        agent.remember(s, a, probs, v, r, done);
        if n_steps mod N = 0 then
            agent.learn();
            learn_iters ← learn_iters + 1;
return trained agent;
```

Principle of the Training Loop :

1. **Agent Initialization :** The PPO agent is instantiated with the environment's action and state dimensions, learning rate α, batch size, and number of epochs for updates.
2. **Game Episodes :** The agent interacts with the environment for n_games episodes. At each step, it chooses an action based on its current policy, executes it, and observes the next state and reward.
3. **Experience Collection :** After each action, the transition (state, action, probability, value, reward, done) is stored in memory for later learning.
4. **Periodic Learning :** Every N time steps, the agent triggers the `learn()` function, which updates both actor and critic networks based on collected experiences.
5. **Score Tracking :** The cumulative rewards per episode are recorded, and the average score is monitored. The agent's models are saved whenever a new best average score is achieved.
6. **Visualization :** After training, a learning curve can be plotted to visualize the agent's performance over episodes.

This loop ensures that the agent continuously improves its policy by alternating between interaction with

the environment and policy updates, following the principles of PPO : stable policy improvement through clipped objective functions and advantage estimation.

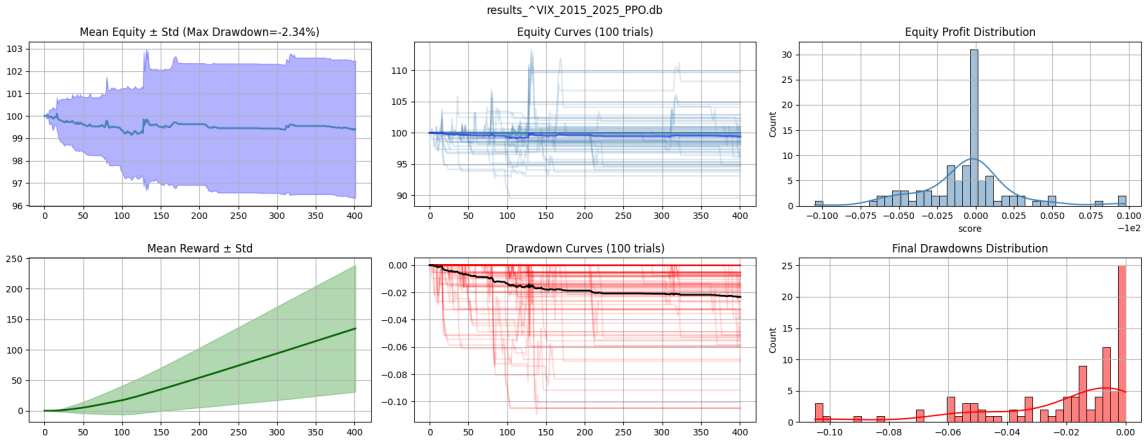


Figure 18.3: First ouput for PPO agent

18.2 Trade Station DataBase

To obtain real-time updates from Trade Station, we are designing a dedicated database server to collect market data from the VIX index at a specified frequency. This service will be deployed using Docker and connected to a database that continuously updates the current VIX bar in real time.

In the future, we plan to extend this server with a function to retrieve historical data from the Trade Station API. This will allow us to test our models in both historical and live settings, ensuring that the same infrastructure can support real-time execution and backtesting simultaneously.

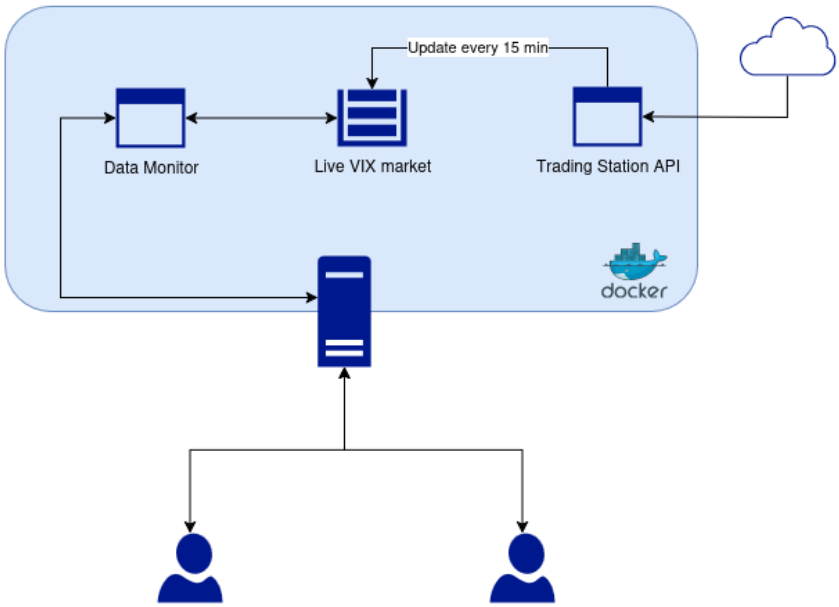


Figure 18.4: Architecture of the Service

18.2.1 Environment

The architecture of our system is designed so that the client can interact directly with the Docker service to access a live copy of the current VIX market data. This setup abstracts away the underlying database and server operations, providing a clean and simple interface for the client to retrieve real-time market information for analysis and model evaluation.

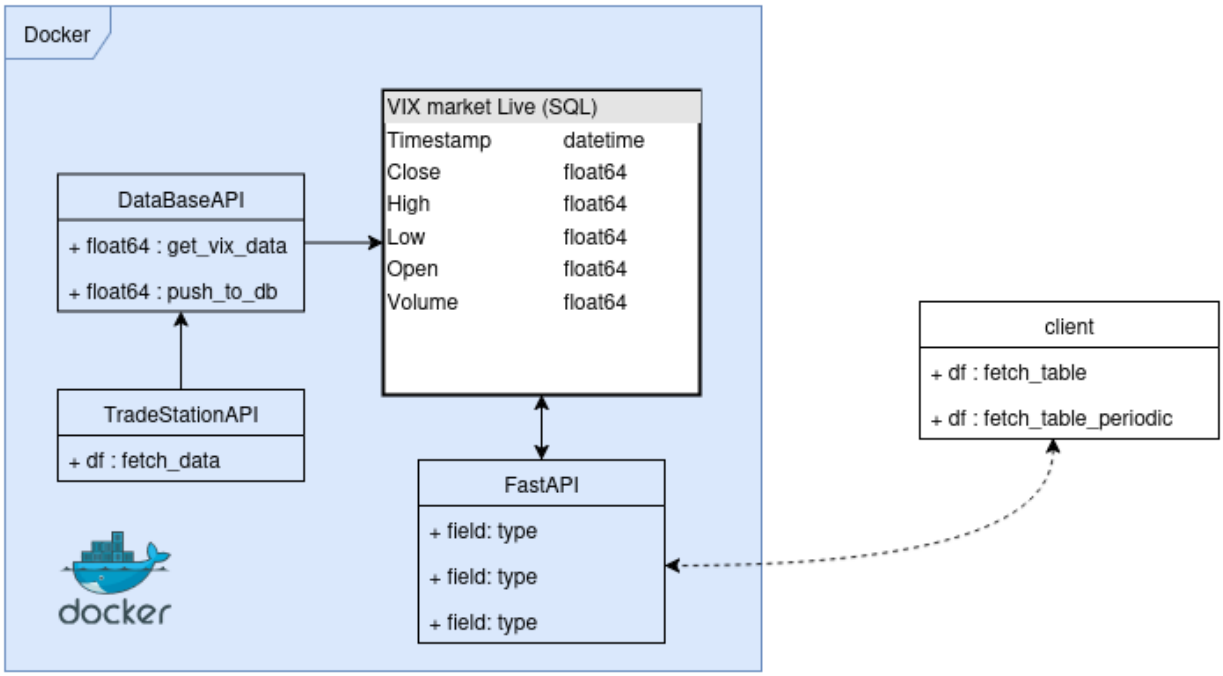


Figure 18.5: UML of the Service

Chapter 20

Week 20

Contents

20.1 The PPO agent Strategy

The PPO (Proximal Policy Optimization) agent strategy has been refined to enable a more robust analysis based on clearer evaluation criteria. This section presents the progress achieved during the week, focusing on a detailed analytical report that examines key aspects of the agent's training performance and optimization process.

20.1.1 The input Data

One crucial aspect of developing our market trading agent is the selection of input data. In particular, the choice of the time interval (or tick size) plays a significant role in shaping the agent's behavior and learning outcomes. After consideration, we identify four possible tick intervals to analyze :

	Signal Capture	Noise Level	Transaction Cost	Computation	Recommended
1-second					not
1-minute					2nd choice
5-minutes					1st choice
15-minutes					not

The most suitable choice appears to be training the agent on 5-minute tick data. This interval maintains the market's dynamic behavior while avoiding excessive computational costs and unnecessary transaction expenses.

Furthermore, since the VIX market dynamics are closely correlated with those of the S&P 500, it would be valuable to incorporate S&P 500 features into the agent's training data. Doing so could enhance the model's understanding of market interactions and improve overall performance. However, this addition would increase the input dimensionality to approximately ten features. Therefore, a comparative experiment should be conducted to evaluate the impact of including these additional variables.

20.1.2 Model Layouts

The proposed model consists of two deep learning components : the **Actor**, which determines the trading positions taken by the agent, and the **Critic**, which evaluates these actions by estimating the expected market outcome and computing a corresponding loss value. Their interaction forms the core of the PPO (Proximal Policy Optimization) framework.

The **Actor** model is based on a transformer architecture with a SoftMax activation function in the output layer, which produces a probability distribution over possible actions. The agent then selects an action if its associated probability exceeds a predefined confidence threshold. In our case, it is recommended to act when the probability is higher than 65%.

The **Critic** model, on the other hand, outputs an estimated closing price of the market at time step $n+1$. This model employs an LSTM (Long Short-Term Memory) layer to capture the temporal dependencies and dynamic behavior of the market. Unlike the Actor, the Critic does not take direct actions; instead, it learns to evaluate the quality of the Actor's decisions based on the predicted market trajectory.

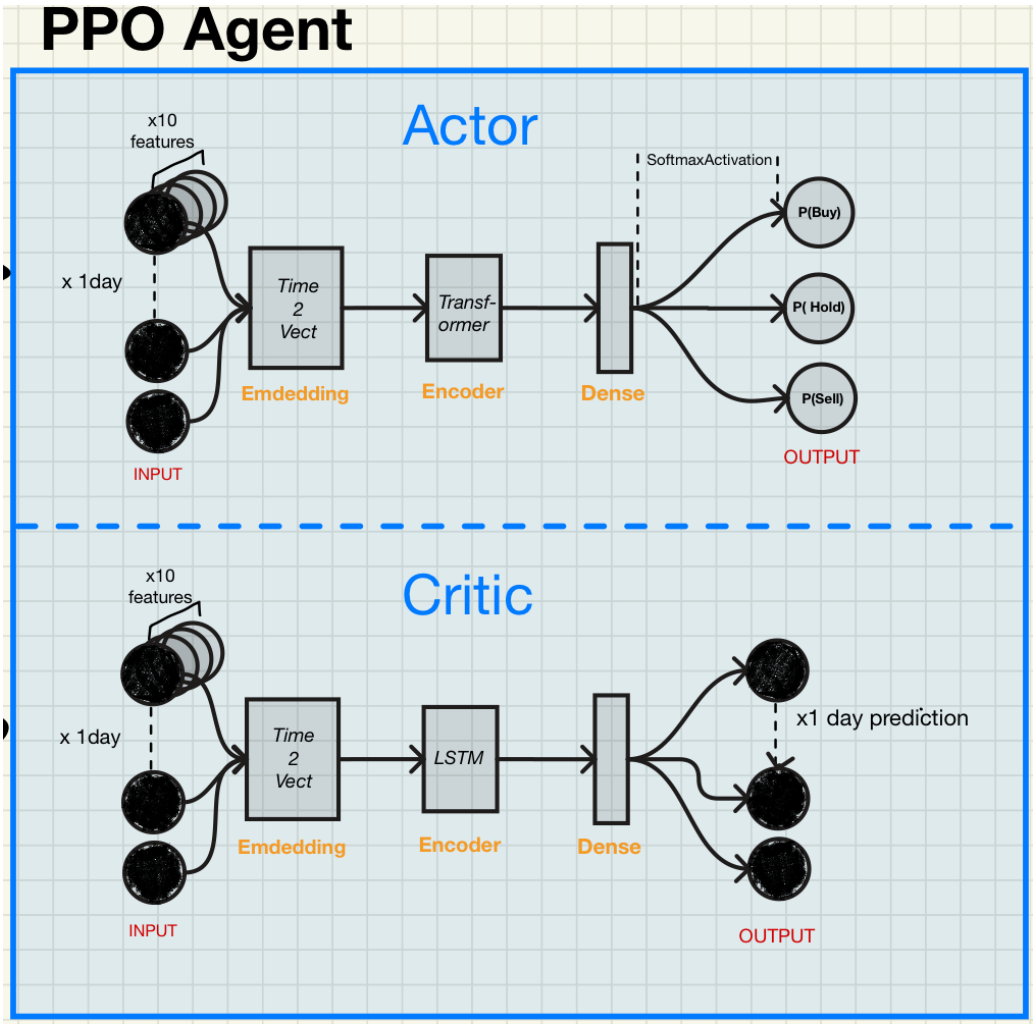


Figure 20.1: PPO agent architecture for the market environment

20.1.3 The performance analysis

After training, a comprehensive performance analysis is required to identify the elements necessary for fine-tuning the agent. To achieve this, several evaluation metrics are recommended:

- **Maximum Drawdown** - measures the largest observed loss from a peak to a trough before a new peak is achieved.
- **Calmar Ratio** - the ratio of annualized return to maximum drawdown, providing a risk-adjusted performance measure.
- **Profit (Equity Curve)** - the cumulative profit evolution over time, representing the model's overall growth performance.
- **Compound Annual Growth Rate (CAGR)** - the mean annual growth rate of the portfolio over a specified period, assuming profits are reinvested.

As input, we will generate a comparative table to evaluate the different fine-tuned models based on these metrics, as shown below :

Total Return	cumulative return over a period
Annual Return	average annual return
Annual Volatility	annual standard deviation on daily returns
Sharpe Ratio	(annual return – risk-free rate)/annual volatility
Sortino Ratio	Sharpe ratio using downside volatility as the risk
Max Drawdown	biggest percentage drop in portfolio
Calmar Ratio	annual return divided by max drawdown
Win/Loss Ratio	percentage of profitable trades
Average Profit	average profit divided by average loss per trade

20.1.4 Environment Fine-Tuning

The environment is implemented using the `Gym` library to simulate real-time market conditions. It retrieves live data from the VIX market and assigns rewards based on the agent's chosen actions. The following sequence diagram illustrates the behavior of the environment during interaction and training:

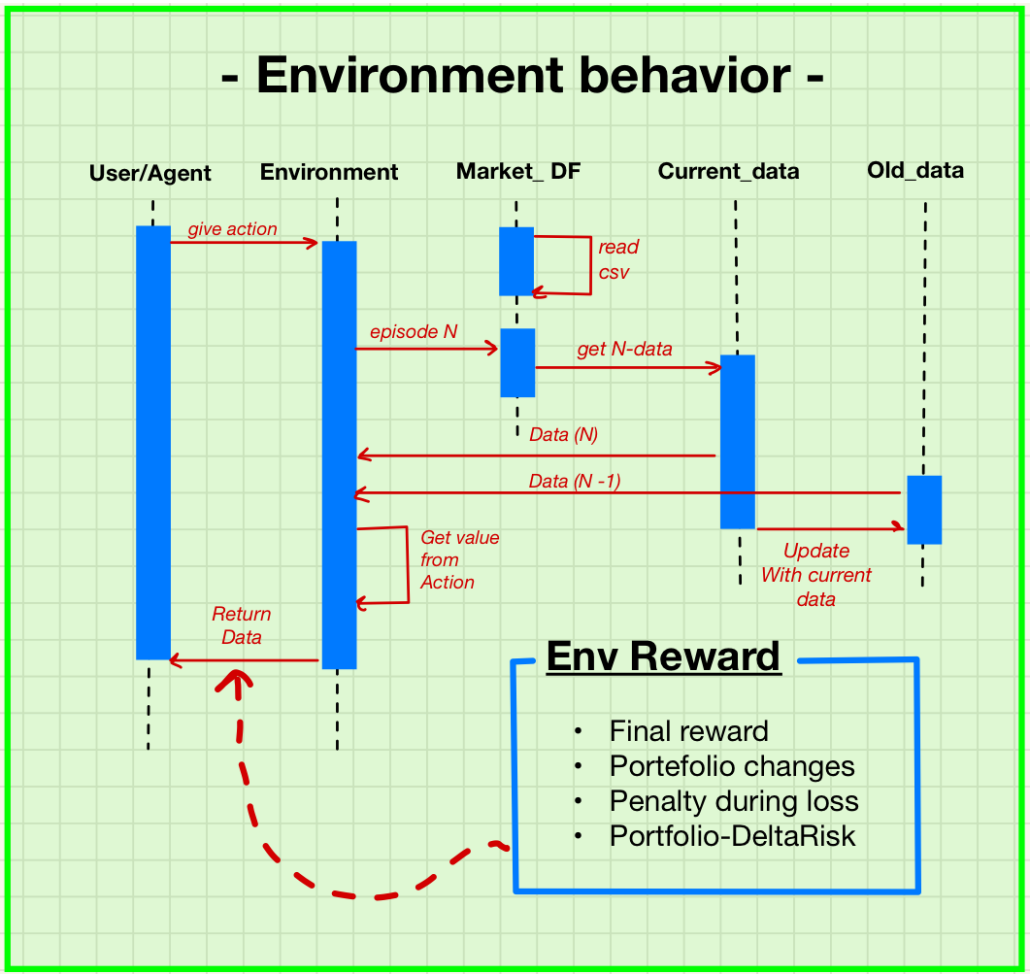


Figure 20.2: Sequence diagram of the environment's behavior.

To stabilize the training process and improve policy convergence, the reward function within the environment can be fine-tuned according to several criteria:

- **Final Reward :** the total profit achieved over a given trading period.
- **Portfolio Change :** the difference between the portfolio value before and after an action is taken.
- **Penalty Loss :** a negative reward applied whenever the agent executes a losing trade.
- **Portfolio Delta Risk :** a risk-adjusted reward component based on the volatility or exposure associated with the selected action.

20.1.5 Risk Management

Risk can be controlled during the testing phase of the agent—prior to deployment—by implementing a set of predefined safety rules. The first measure involves introducing a **stop-loss mechanism**, defined by a user-specified loss threshold, to automatically close a position when losses exceed acceptable limits. The second measure consists of enforcing a **Maximum Daily Drawdown** constraint to prevent the agent from taking excessive risks during periods of strong market decline.

20.1.6 Suggestions and Future Work

The following list outlines the next steps for improving and extending the PPO agent framework:

1. Incorporate S&P 500 features into the environment (using 5-minute tick data).
2. Refine the **Actor** deep learning architecture:
    - Integrate a `Time2Vec` layer to encode temporal patterns.
    - Modify the output layer to provide probability scores for each possible action.
3. Develop the **Critic** deep learning architecture:
    - Include a `Time2Vec` layer.
    - Employ an `LSTM` layer to capture temporal dependencies.
    - Produce an output representing the predicted closing price.
4. Implement comprehensive performance analysis tools for post-training evaluation.
5. Use the performance metrics to fine-tune the environment's reward structure.
6. Integrate additional risk management features (e.g., stop-loss and maximum drawdown limits).

20.2 Real-Time Service

In parallel with the PPO agent development, a real-time data service has been implemented to collect VIX market data across multiple computers.

20.2.1 Service Description

The service leverages the `TradeStation API` to retrieve live market data. On the laboratory servers, a background process continuously runs to collect and update the latest VIX market features in real time. The data are stored in a single-row database that is refreshed every 15 minutes. However, as discussed previously, a 5-minute update interval would provide better responsiveness to market dynamics, and this adjustment will be implemented in future versions.

When the database is updated, the server exposes an `HTTP` endpoint accessible via the laboratory's local network (LAN), allowing clients to query the latest data. In addition, the server hosts a lightweight debugging interface that acts as the main access point for clients to retrieve system information and monitor status.

The client component has been developed as a Python library. At this stage, it can retrieve the most recent VIX data and locally store all database updates, thereby maintaining a continuous historical record on the client machine.

20.2.2 Future Updates

Future improvements will focus on extending the system's capabilities beyond merely retrieving the latest data. For instance, one key enhancement will enable the server to use the `TradeStation API` to independently fetch historical data ranges upon client request. To achieve this, a dedicated cache database will be introduced to temporarily store the requested data before transmission to the client via `HTTP`.

The main objectives for the next iteration are as follows:

- Reduce the update interval from 15 minutes to a 5-minute basis.
- Add a dedicated database for range-based data requests.
- Implement functionality to retrieve and transmit a range of data (both server- and client-side).