
Report on Kalman Filter applied on Neural Network for Finance

August 26, 2025

Mathys VINATIER - [GitHub Project page](#)

Supervisor:

Pr Kim Tae-Wan

Mathys VINATIER

Contents

10 Week 10	1
11 Week 11	17
12 Week 12	24
13 Week 13	30

Chapter 10

Week 10

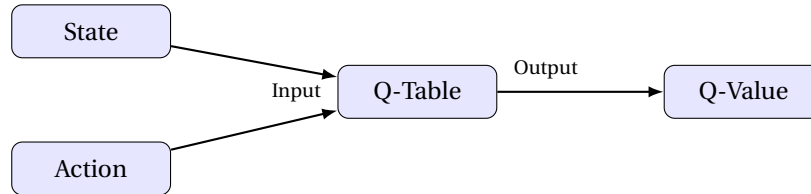
Contents

10.1 Reminder on Greedy Policy and QLearning	2
10.2 Implementation Progress and Experiments	3
10.2.1 Environment Setup	3
10.2.2 Preliminary Results for Q-Learning Greedy Policy	5
10.3 Challenges and Solutions	15
10.4 Next Steps	15

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.

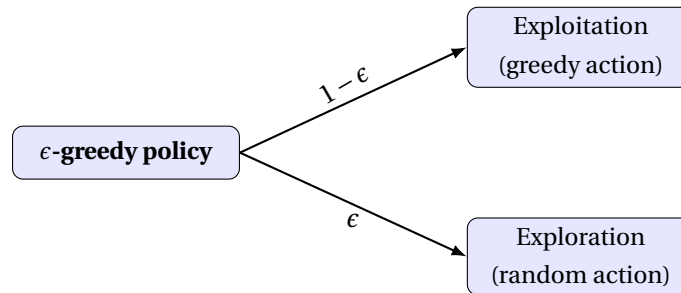
- **Q-table:** Internally, Q-Learning uses a Q-table storing values for each state-action pair. Initially, all values are zero, and the table is updated iteratively during training.



- **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

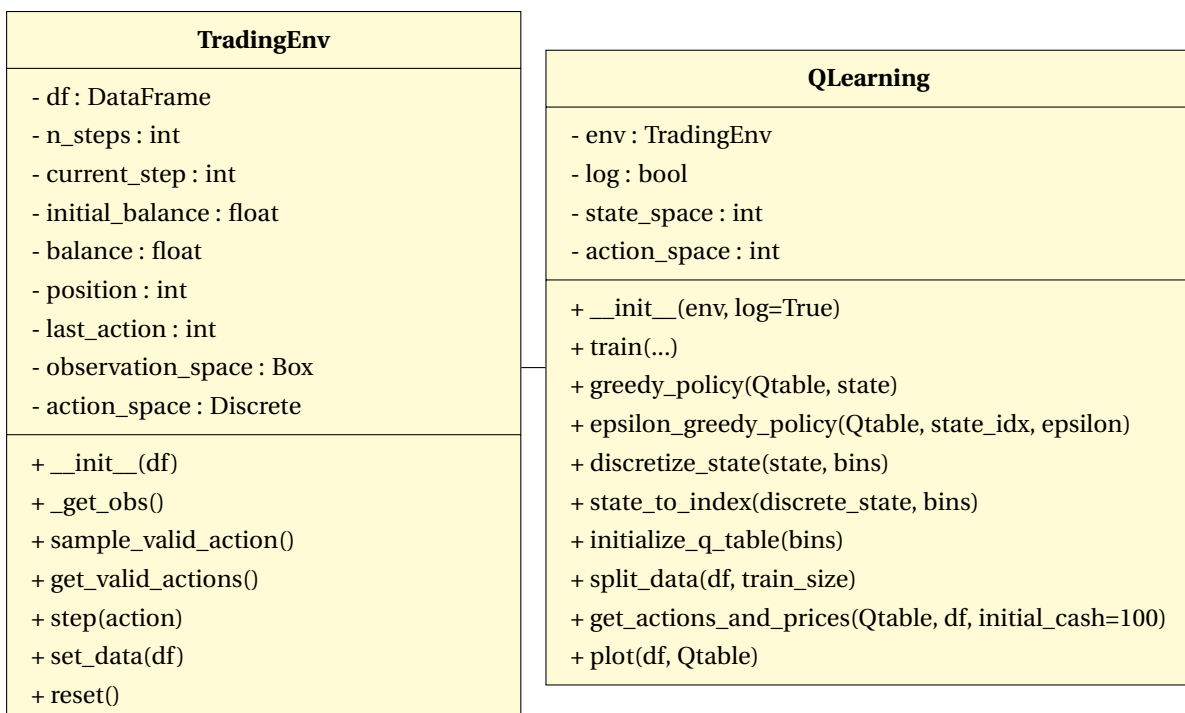
This week was dedicated to the initial implementation and experimentation phase of the project, focusing primarily on the Proximal Policy Optimization (PPO) basis algorithms. Using the Hugging Face classes, we developed a prototype with greedy policy agent and integrated it into a custom trading environment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



Algorithm 1: Trading Environment Behavior**Input:** Environment env , number of episodes N , exploration rate ϵ **Output:** Episode value and rewards

```

1 for  $episode \leftarrow 1$  to  $N$ 
2    $s \leftarrow env.reset()$ ;
3    $total\_reward \leftarrow 0$ ;
4   while  $True$ 
5     Get valid actions  $A_{valid} \leftarrow env.get\_valid\_actions()$ ;
6     Select  $a \leftarrow$  random choice from  $A_{valid}$ ;
7     Execute  $(s', r, done, info) \leftarrow env.step(a)$ ;
8      $total\_reward \leftarrow total\_reward + r$ ;
9     Log( $episode, s, a, r, info["portfolio\_value"]$ );
10     $s \leftarrow s'$ ;
11    if  $done$  then
12      break
13  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

```

1 Split data:  $df_{train} \leftarrow train\_size$  of  $df$ ;
2 Calculate bins for discretization :  $bins \leftarrow$  compute bins from  $df_{train}$ ;
3 Initialize Q-table
4 for  $episode \leftarrow 1$  to  $N$ 
5   Exploration rate :  $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$ ;
6   Reset environment :  $state_{cont} \leftarrow env.reset()$ ;
7   Discretize initial state :  $state_{disc} \leftarrow discretize(state_{cont}, bins)$ ;
8   Convert to index :  $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$ ;
9   for  $step \leftarrow 1$  to  $max\_steps$ 
10    Choose action  $a$  using epsilon-greedy policy :
        
$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1 - \epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

11    Take action :  $(next\_state_{cont}, r, done, info) \leftarrow env.step(a)$ ;
12    Discretize next state :  $next\_state_{disc} \leftarrow discretize(next\_state_{cont}, bins)$ ;
13    Convert to index :  $next\_state_{idx} \leftarrow state\_to\_index(next\_state_{disc}, bins)$ ;
14    Update Q-value
         $:Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha (r + \gamma \max_{a'} Q[next\_state_{idx}, a'] - Q[state_{idx}, a])$ ;
15     $state_{idx} \leftarrow next\_state_{idx}$ ;
16    if  $done$  then
17      break;
18 return  $Q$ 

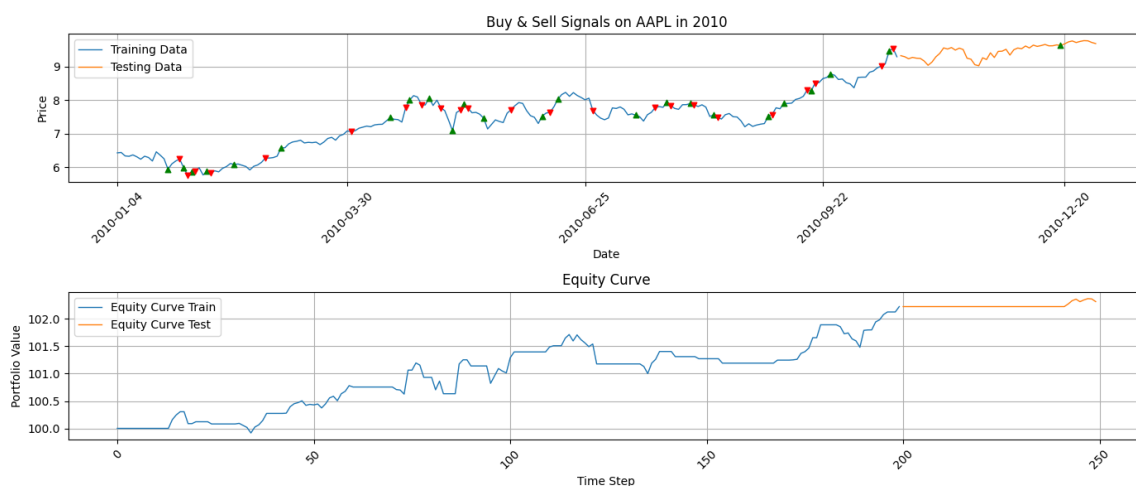
```

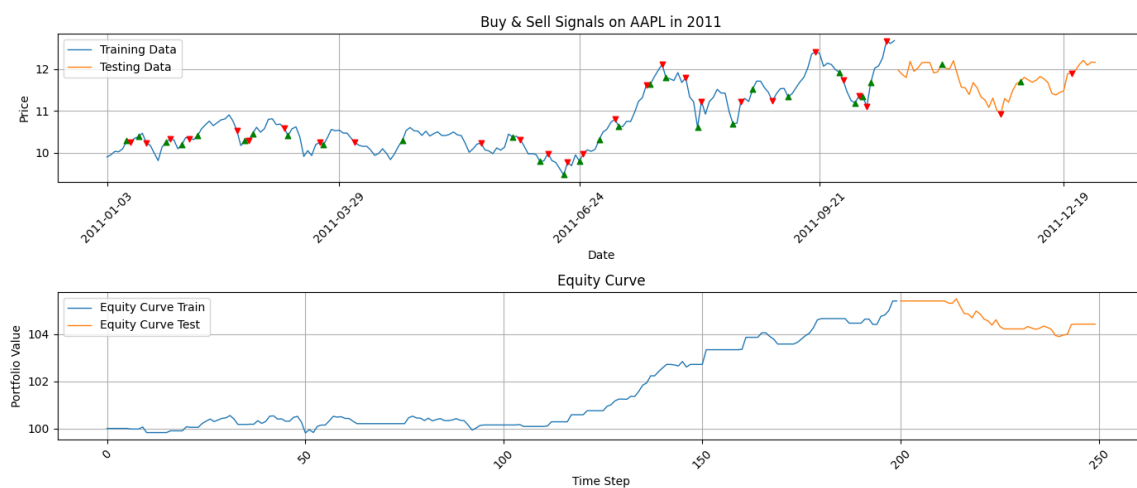
In the context of Q-learning for trading environments, bins are used to discretize the continuous state space into a finite number of discrete states. This discretization is essential because the Q-table requires a finite and manageable number of states to store and update the expected rewards for each 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.

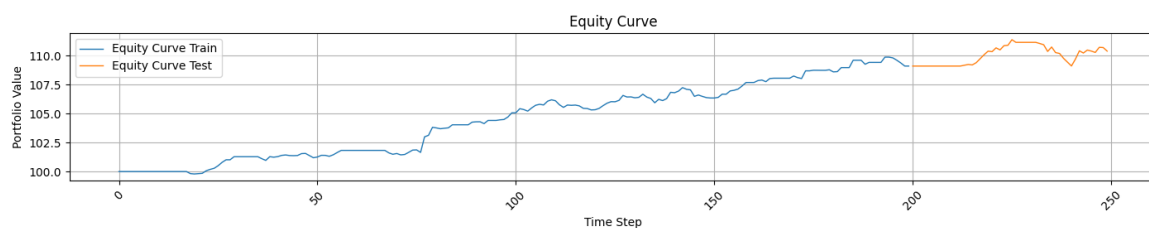
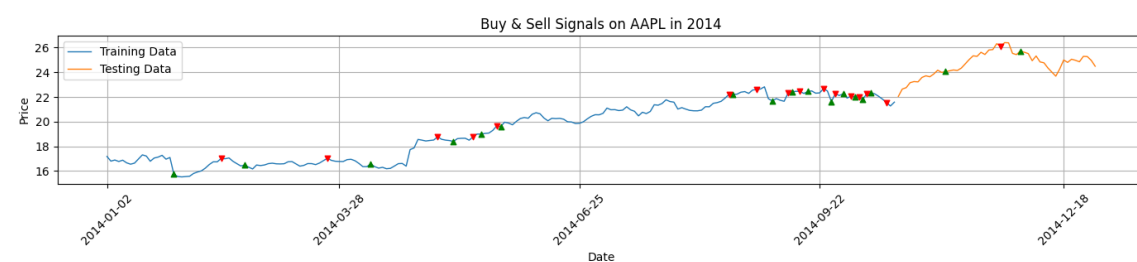
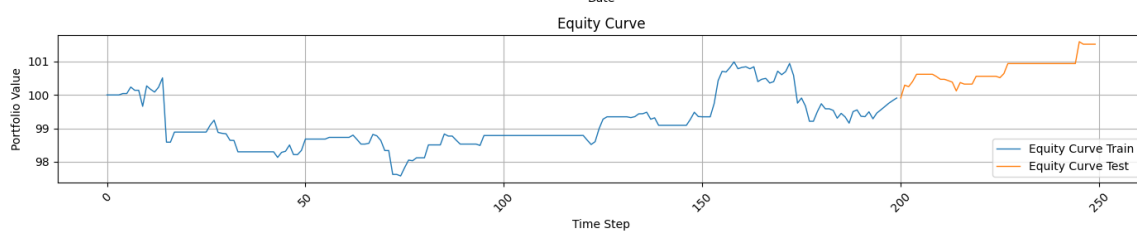
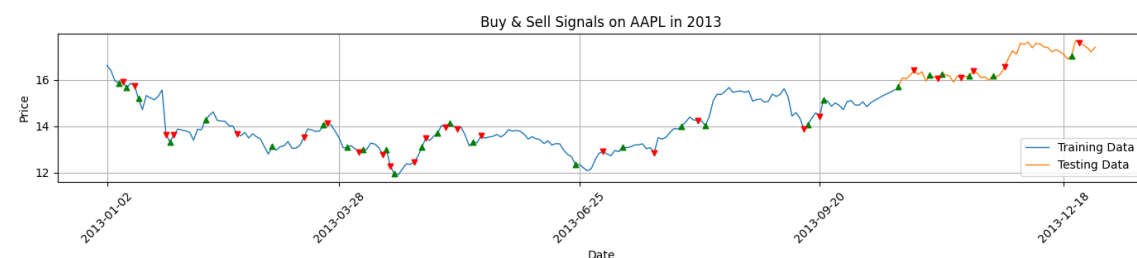
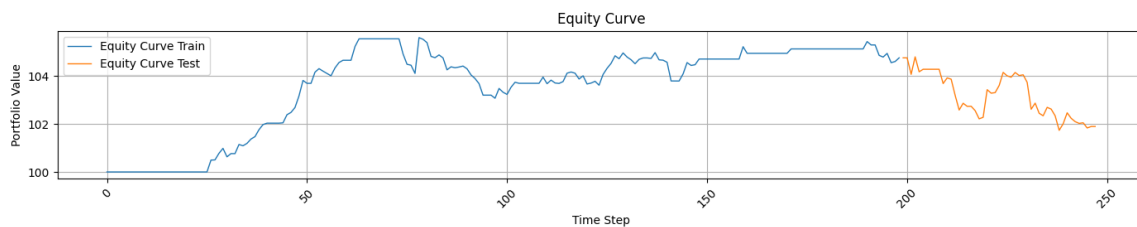
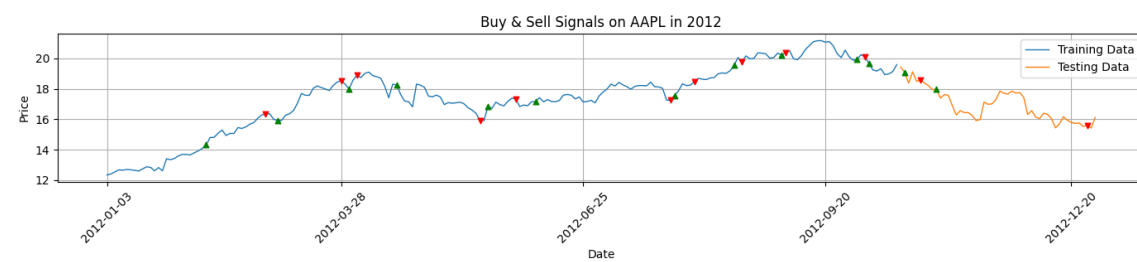
10.2.2 Preliminary Results for Q-Learning Greedy Policy

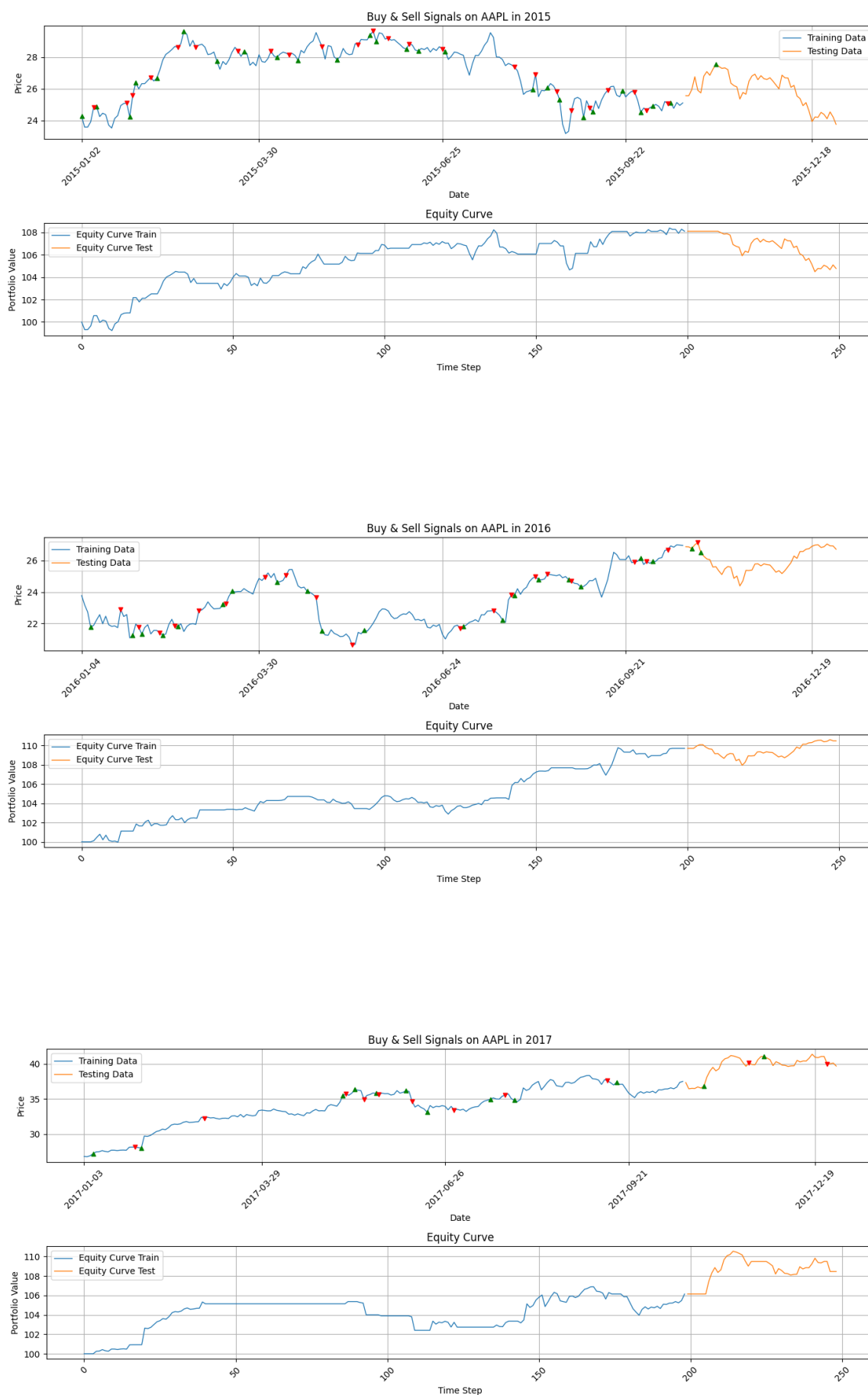
Initial experiments aimed to evaluate the training behavior and reward progression of the Q-Learning agent following a greedy policy. Results were compared over two time horizons : a shorter training period of 100 episodes and an extended training of 1000 episodes.

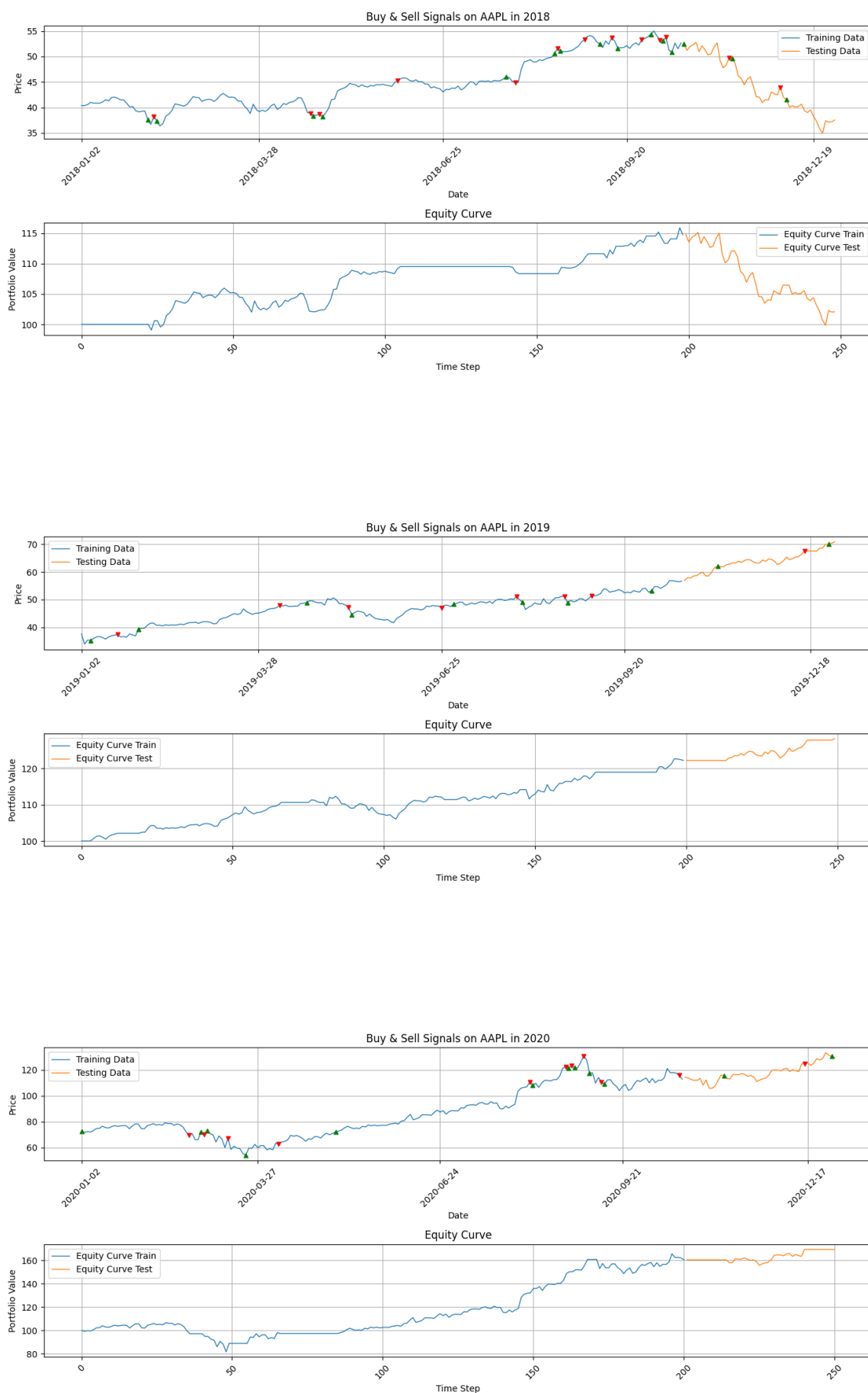
100 episodes

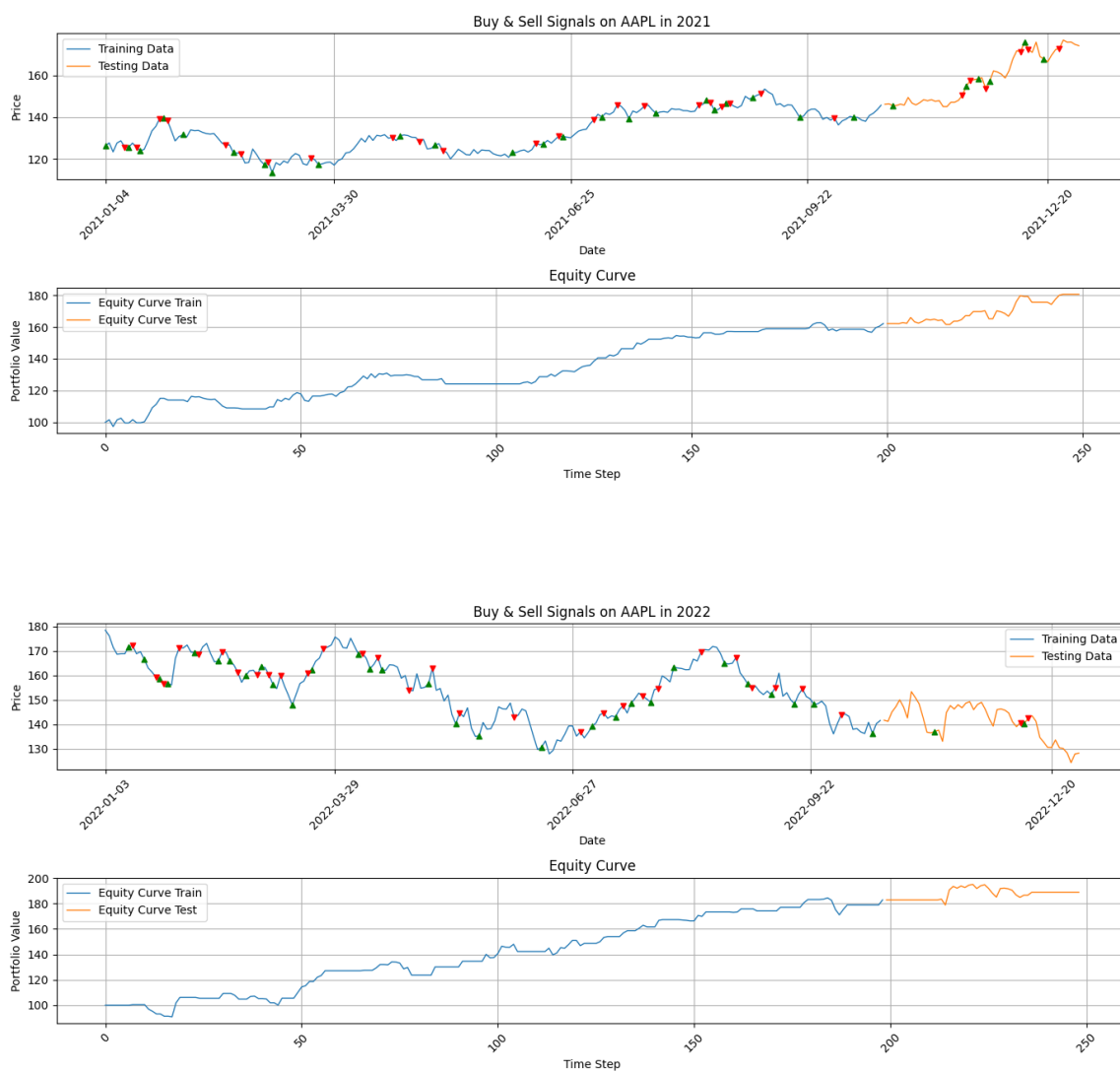




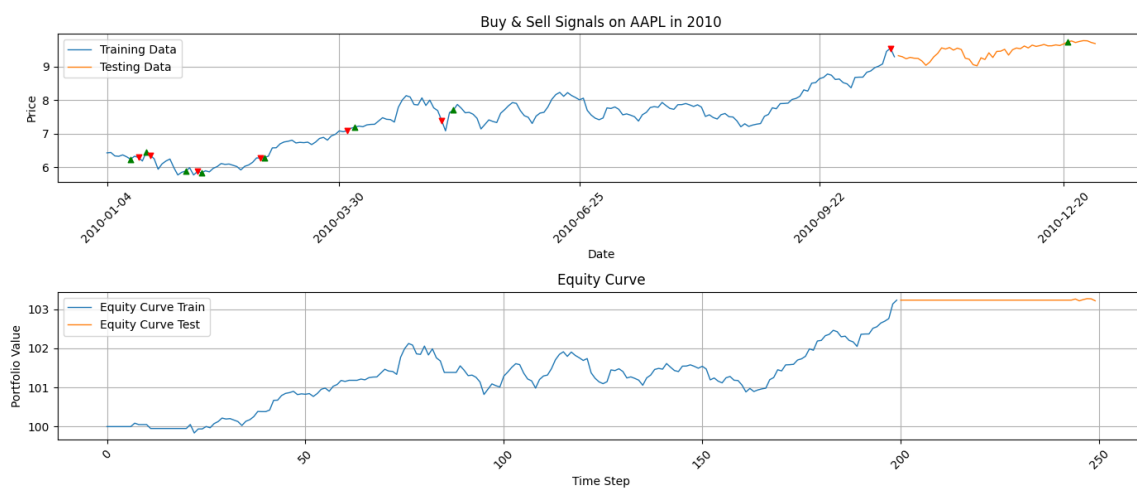


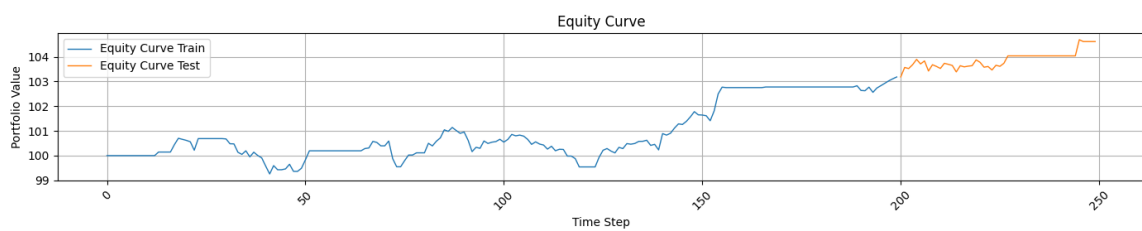
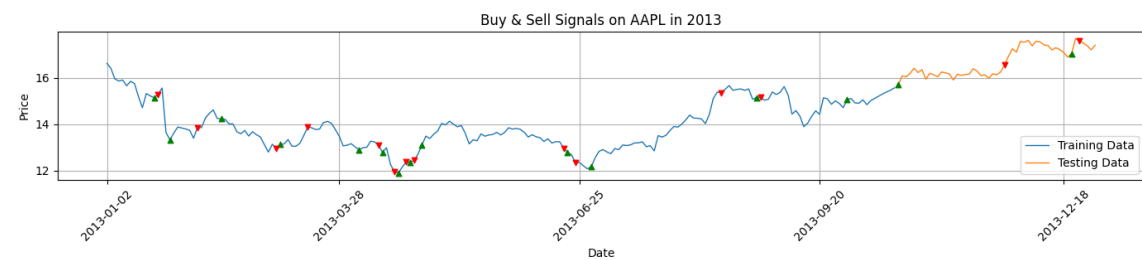
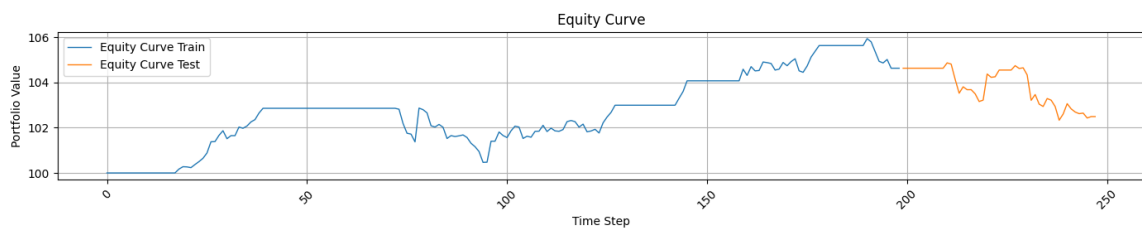
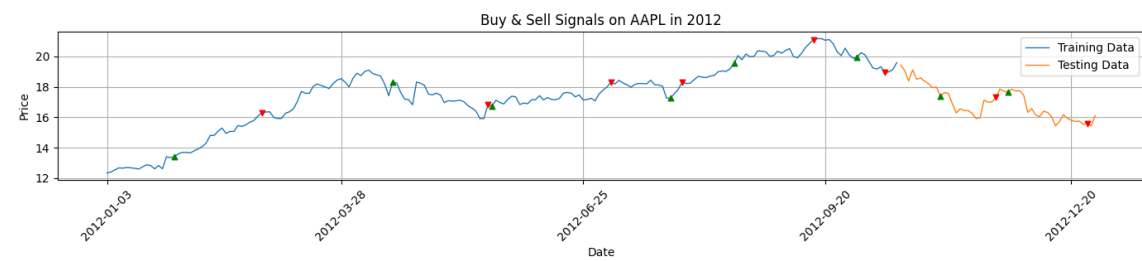
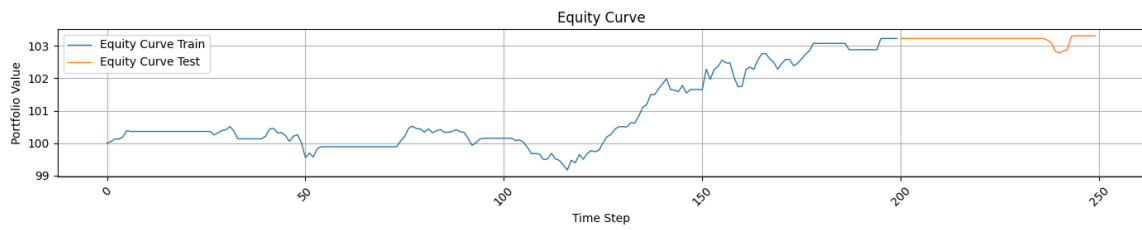
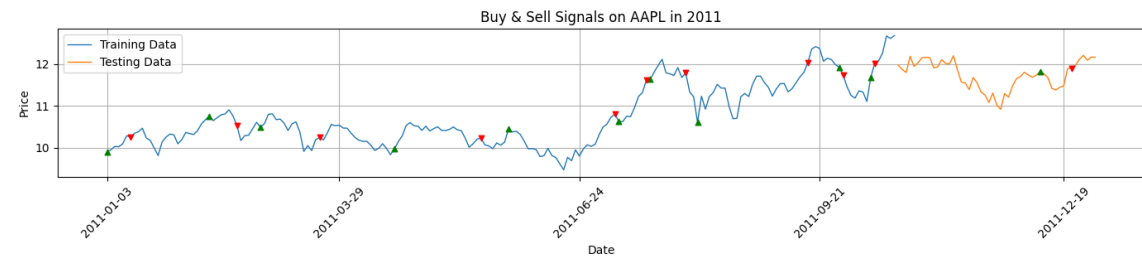


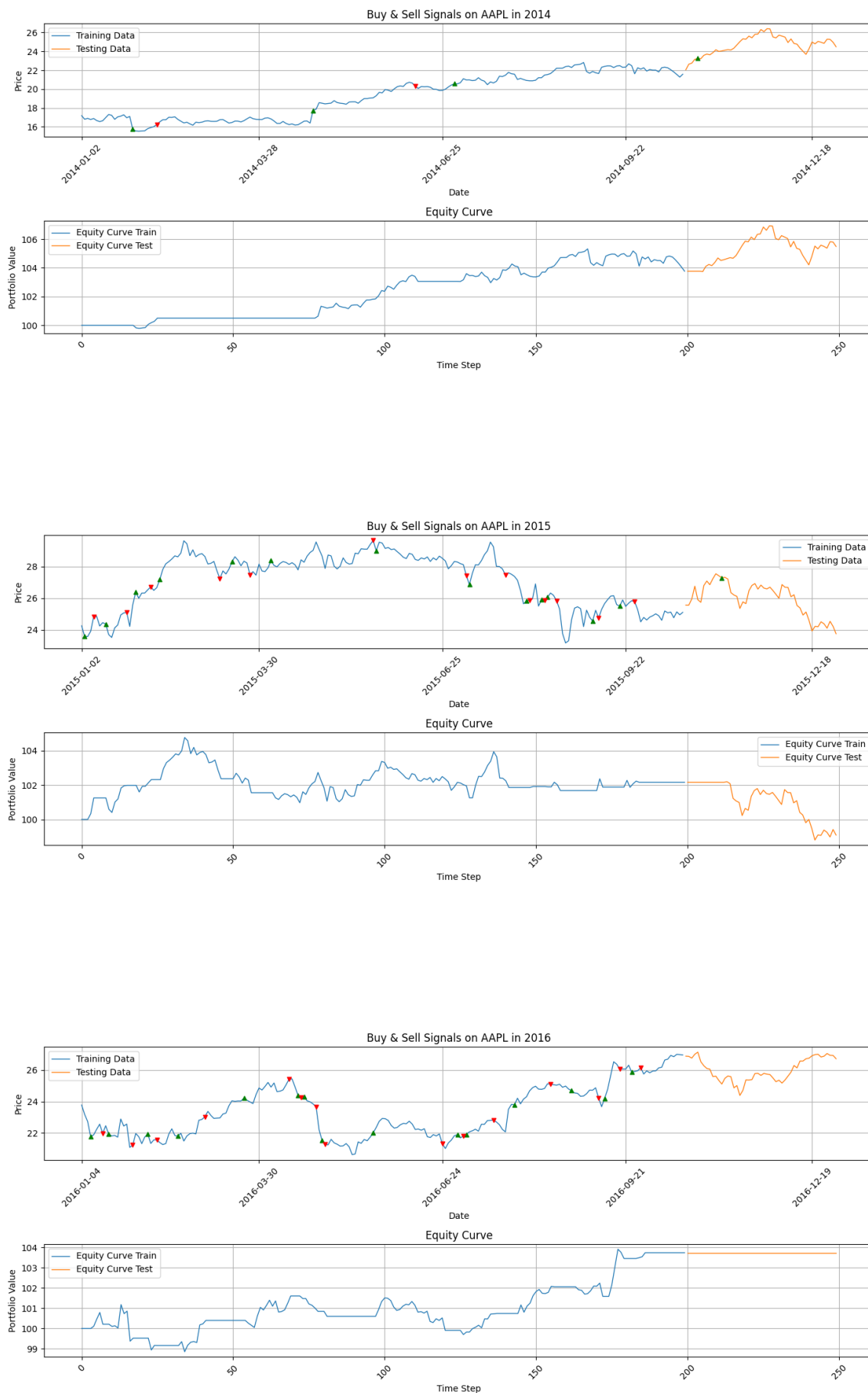


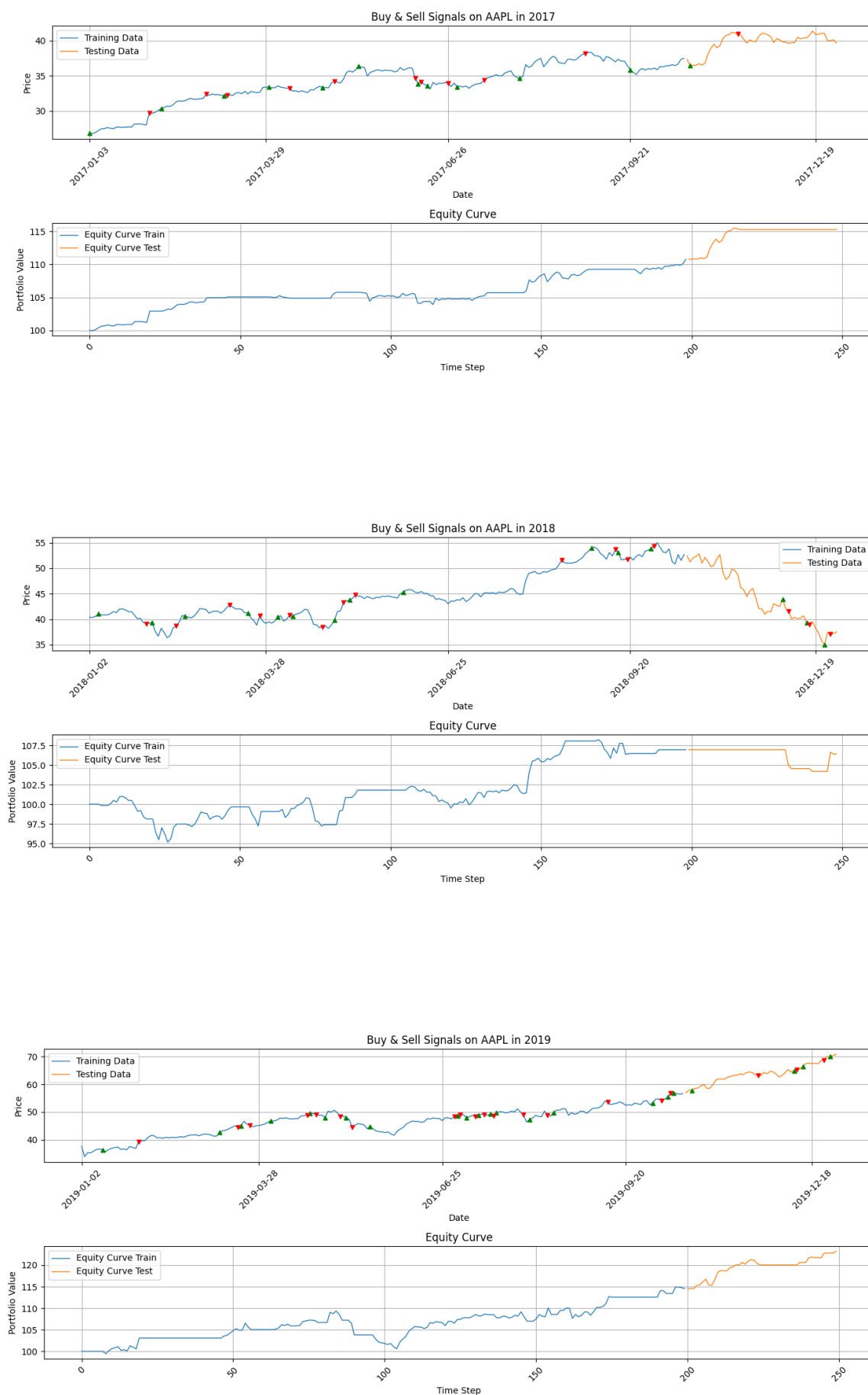


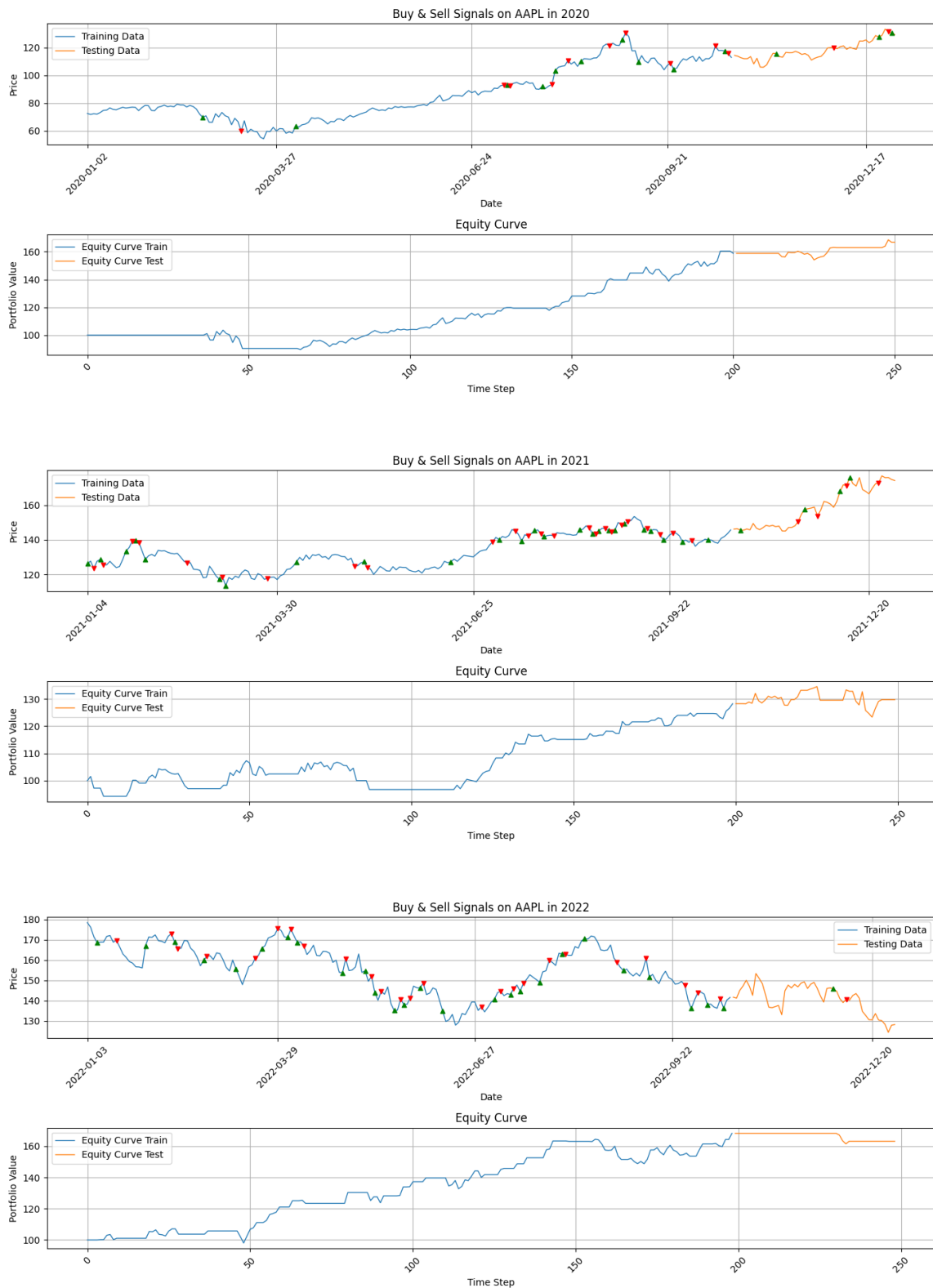
1000 episodes











- 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 should 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	18
11.1.1 Problem Formulation	18
11.1.2 Deep Q-Network (DQN)	18
11.1.3 Exploration and Constraints	19
11.1.4 Training Algorithm	19
11.1.5 Practical Considerations for Time-Series	19
11.1.6 Evaluation Metrics	20
11.1.7 Model Variants (Optional)	20
11.1.8 Reference Hyperparameters (Typical Ranges)	20
11.1.9 Limitations	20
11.2 Transformer-Based Deep Q-Learning for Time-Series Trading	21
11.2.1 Problem Formulation	21
11.2.2 Transformer Q-Network	21
11.2.3 Training with Double DQN Targets	22
11.2.4 Exploration and Constraints	22
11.2.5 Practical Considerations	22
11.2.6 Evaluation Metrics	22
11.2.7 Remarks	22

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 $s_t \in \mathcal{S}$ (Price, Close, High, Low, Open and Volume), takes an action $a_t \in \mathcal{A}$ (Buy, Old or Sell), receives a reward $r_t \in \mathbb{R}$.

State. To incorporate temporal context, we define

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

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:

$$\mathcal{A} = \{-K, \dots, -1, 0, 1, \dots, K\},$$

where a_t represents the target position (short to long) in normalized units subject to a position limit $|a_t| \leq K$. An alternative is the set {SHORT, FLAT, 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

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

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 DD_{max} .

11.1.2 Deep Q-Network (DQN)

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

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

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

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

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:

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

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 $|\mathcal{A}|$ 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 ϵ_{\max} to ϵ_{\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

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

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 DD_{\max} .

11.1.6 Evaluation Metrics

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

- Annualized Sharpe: $SR = \frac{\sqrt{A} \mathbb{E}[R_t]}{\text{Std}[R_t]}$ 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 $Q_\theta(s, a) = V_\theta(s) + A_\theta(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A_\theta(s, a')$ to stabilize value estimation.
- **N-step Returns:** Replace y with n -step target $r_t + \gamma r_{t+1} + \dots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_{a'} Q_{\hat{\theta}}(s_{t+n}, a')$.
- **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 $ \mathcal{D} $	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^* 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 $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$, receives reward r_t , and transitions to s_{t+1} .

State. The state is a sequence of past observations:

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

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:

$$\mathcal{A} = \{-K, \dots, -1, 0, 1, \dots, K\},$$

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

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

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

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 $|\mathcal{A}|$ Q-values.

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

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

11.2.3 Training with Double DQN Targets

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

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

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 \mathcal{D} :

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

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

```

1 Split data:  $df_{train} \leftarrow train\_size$  of  $df$ ;
2 Calculate bins for discretization :  $bins \leftarrow$  compute bins from  $df_{train}$ ;
3 Initialize Q-table
4 for  $episode \leftarrow 1$  to  $N$ 
5   Exploration rate :  $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$ ;
6   Reset environment :  $state_{cont} \leftarrow env.reset()$ ;
7   Discretize initial state :  $state_{disc} \leftarrow discretize(state_{cont}, bins)$ ;
8   Convert to index :  $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$ ;
9   for  $step \leftarrow 1$  to  $max\_steps$ 
10    Choose action  $a$  using epsilon-greedy policy :
        
$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1 - \epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

11    Take action :  $(next\_state_{cont}, r, done, info) \leftarrow env.step(a)$ ;
12    Discretize next state :  $next\_state_{disc} \leftarrow discretize(next\_state_{cont}, bins)$ ;
13    Convert to index :  $next\_state_{idx} \leftarrow state\_to\_index(next\_state_{disc}, bins)$ ;
14    Update Q-value
        :  $Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha (r + \gamma \max_{a'} Q[next\_state_{idx}, a'] - Q[state_{idx}, a])$ ;
15     $state_{idx} \leftarrow next\_state_{idx}$ ;
16    if  $done$  then
17      break;
18 return  $Q$ 

```

Chapter 12

Week 12

Contents

12.1 Deep Q-Learning	25
12.1.1 Deep Q-Learning introduction	25
12.1.2 Deep Q-Learning Process	26
12.1.3 MLP layer	26
12.1.4 Decision Transformer Layer	28

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

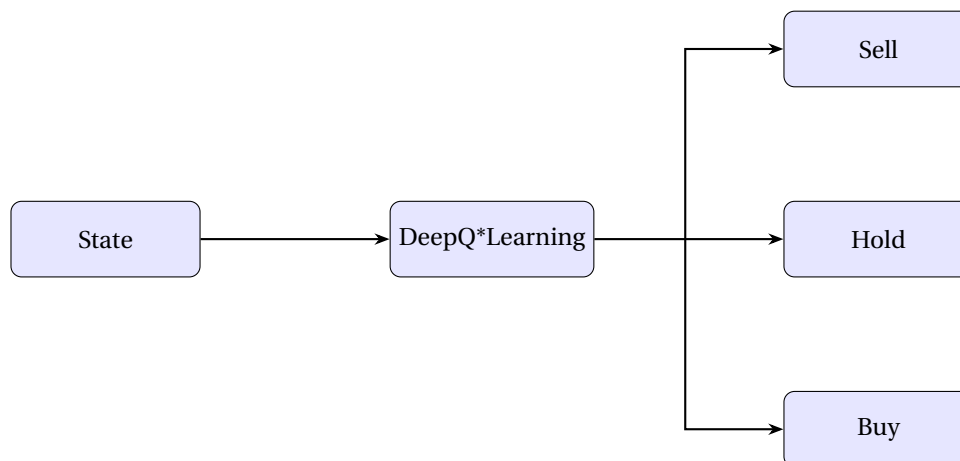
```

1 Split data:  $df_{train} \leftarrow train\_size$  of  $df$ ;
2 Calculate bins for discretization :  $bins \leftarrow$  compute bins from  $df_{train}$ ;
3 Initialize Q-table
4 for  $episode \leftarrow 1$  to  $N$ 
5   Exploration rate :  $\epsilon \leftarrow \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot \exp(-decay\_rate \times episode)$ ;
6   Reset environment :  $state_{cont} \leftarrow env.reset()$ ;
7   Discretize initial state :  $state_{disc} \leftarrow discretize(state_{cont}, bins)$ ;
8   Convert to index :  $state_{idx} \leftarrow state\_to\_index(state_{disc}, bins)$ ;
9   for  $step \leftarrow 1$  to  $max\_steps$ 
10    Choose action  $a$  using epsilon-greedy policy :
        
$$a \leftarrow \begin{cases} \arg\max_{a'} Q[state_{idx}, a'] & \text{with probability } 1 - \epsilon \\ \text{random valid action} & \text{with probability } \epsilon \end{cases}$$

11    Take action :  $(next\_state_{cont}, r, done, info) \leftarrow env.step(a)$ ;
12    Discretize next state :  $next\_state_{disc} \leftarrow discretize(next\_state_{cont}, bins)$ ;
13    Convert to index :  $next\_state_{idx} \leftarrow state\_to\_index(next\_state_{disc}, bins)$ ;
14    Update Q-value
        :  $Q[state_{idx}, a] \leftarrow Q[state_{idx}, a] + \alpha (r + \gamma \max_{a'} Q[next\_state_{idx}, a'] - Q[state_{idx}, a])$ ;
15     $state_{idx} \leftarrow next\_state_{idx}$ ;
16    if  $done$  then
17      break;
18 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
<code>nn.Linear</code>	Linear Layer
<code>nn.ReLU</code>	Activation Function
<code>nn.Sequential</code>	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 :

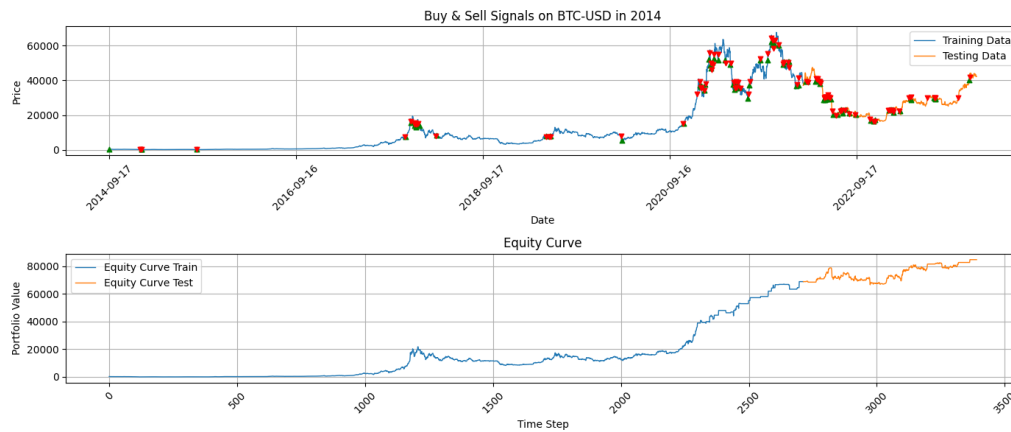


Figure 12.1: Performance of the Q-Learning Model on a BTC-USD test set

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 :

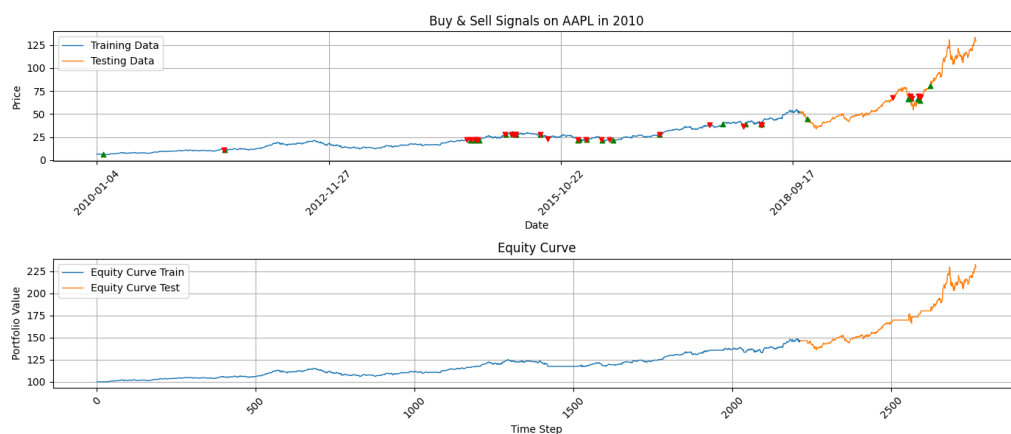
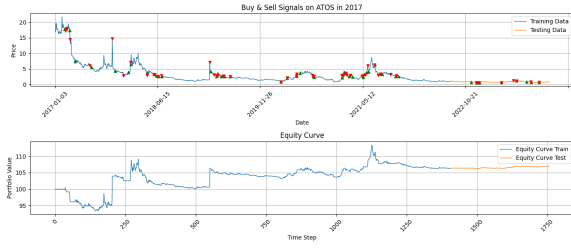
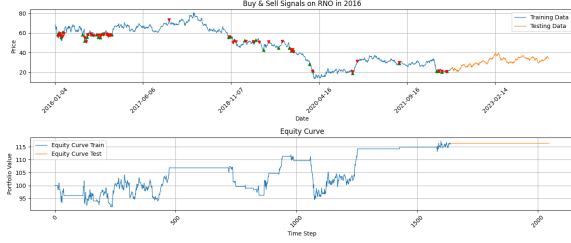


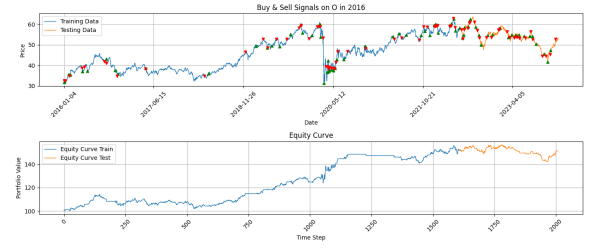
Figure 12.2: Performance on Apple (AAPL) data from 2010.



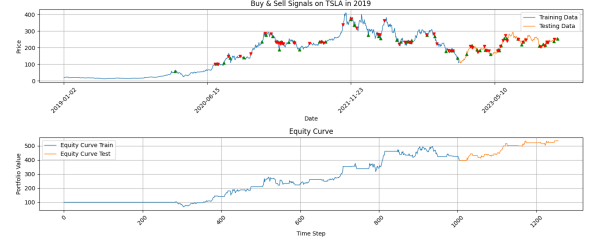
(a) Performance on ATOS data from 2017.



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



(b) Performance on Realty Income (O) 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:

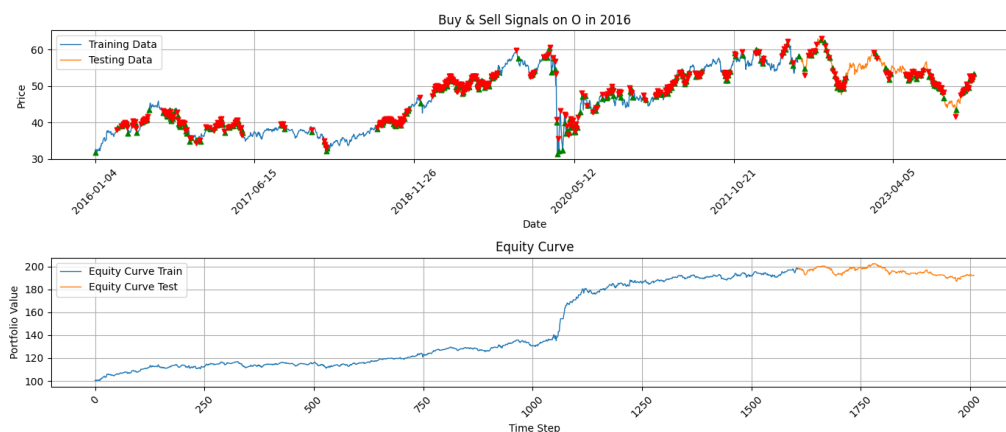


Figure 12.4: Performance of the Deep Q-Learning Model on Realty Income (O) data from 2016.

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	31
13.1.1 Decision Transformer Layer	31
13.1.2 GPT2 Transformer description	31
13.1.3 Result for QLearning (with commission)	34
13.1.4 Result for DeepQLearning - GPT Transformer (with commission)	35
13.2 Paper : Transformers in Reinforcement Learning : A survey	36
13.2.1 Transformer RL in Trading	36

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.

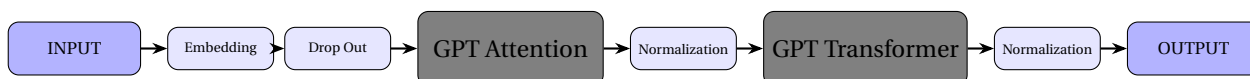
Before the input is fed into the transformer, it passes through a '`self.input_proj`' linear layer. This layer's purpose is to project the raw input data, which has a dimensionality of `input_dim`, into an embedding space that matches the transformer's hidden size. This ensures the input is correctly 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
<code>DecisionTransformerGPT2Model</code>	Transformer Backbone
<code>nn.Linear (Input)</code>	Linear Layer
<code>nn.Linear (Q-Head)</code>	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.

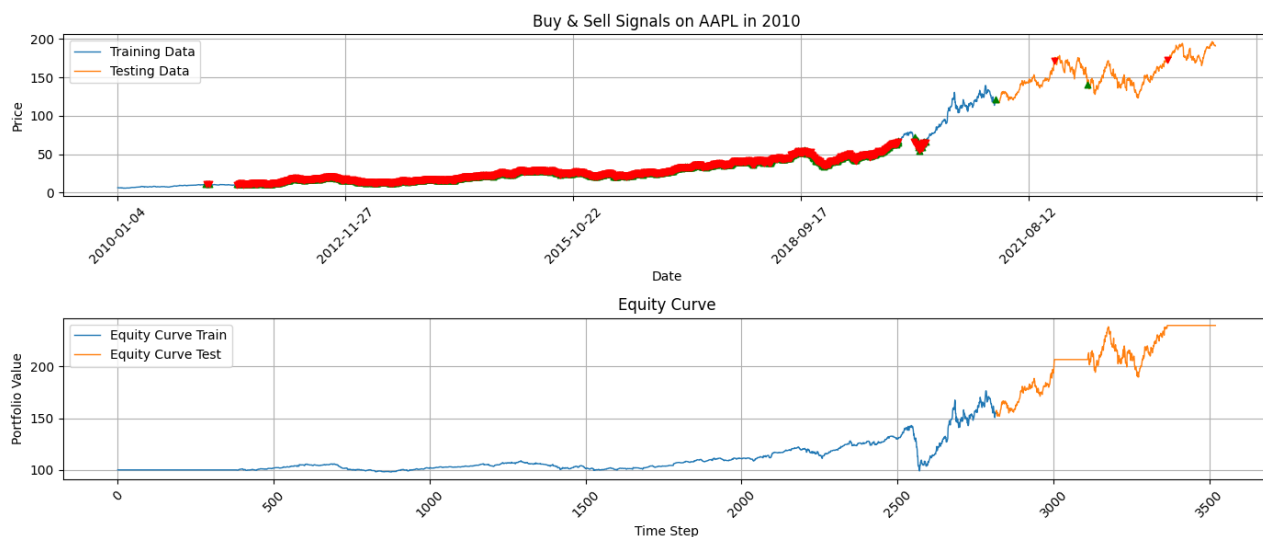


Figure 13.1: Performance of the Transofrmer Model on Apple data from 2010.

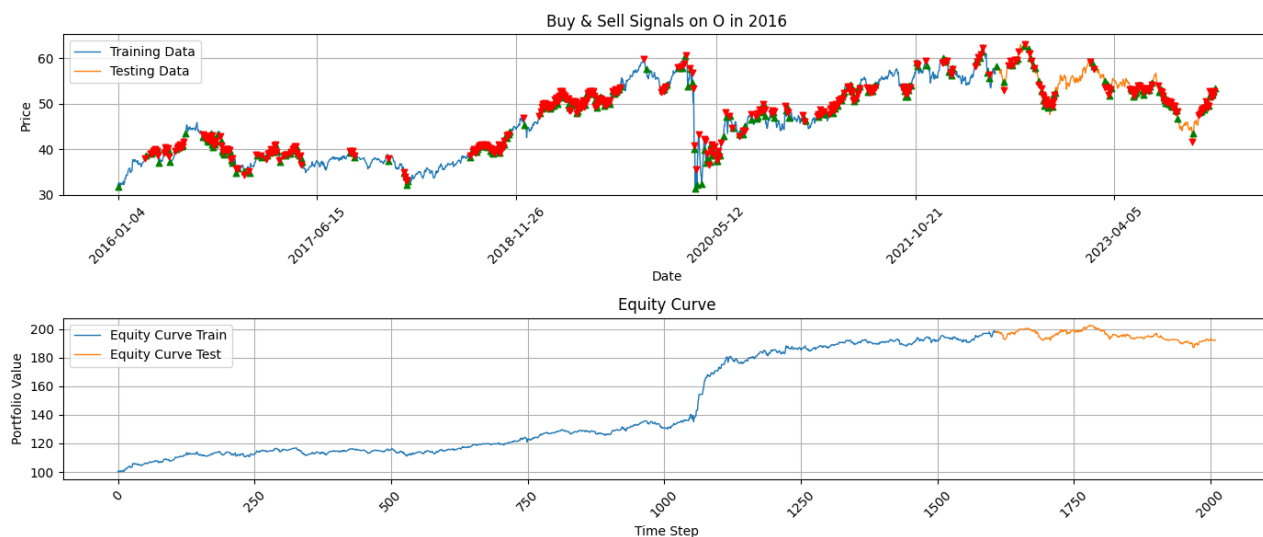


Figure 13.2: Performance of the Transofrmer Model on Realty income data from 2016.

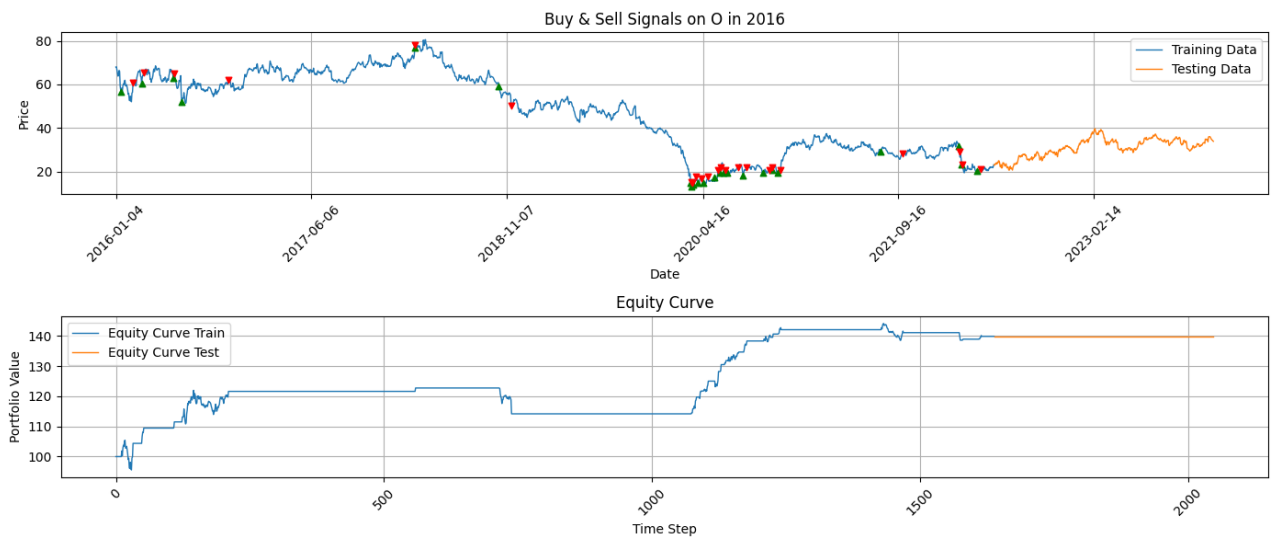


Figure 13.3: Performance of the Transofrmer Model on Renault data from 2016.



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 :

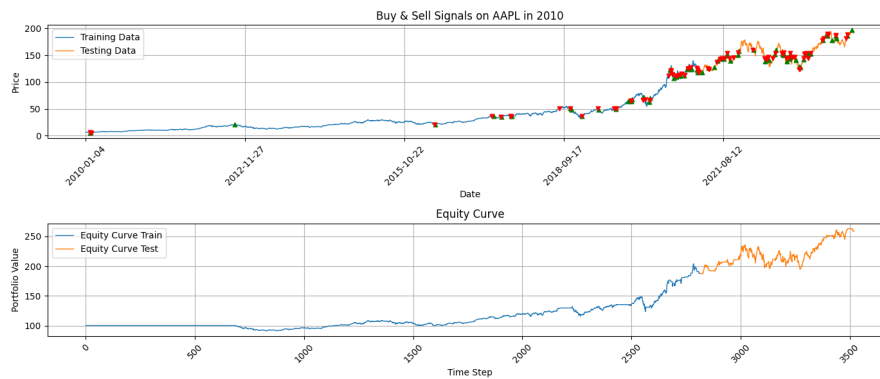


Figure 13.5: Performance of the QLearning Model on Apple data from 2010 with commission

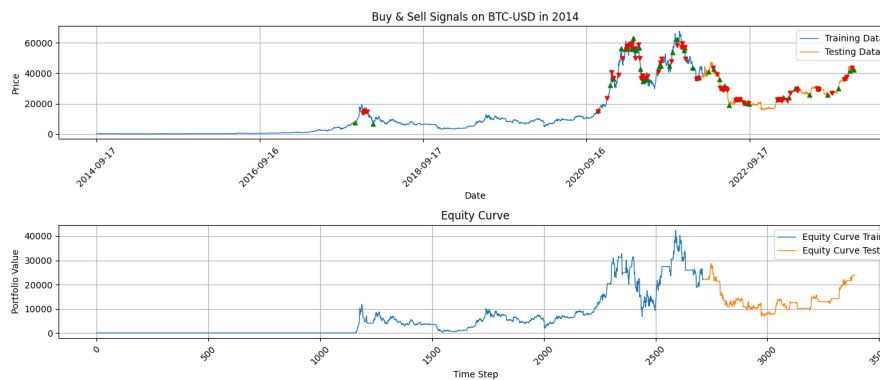


Figure 13.6: Performance of the QLearning Model on Bitcoin data from 2014 with commission



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 :

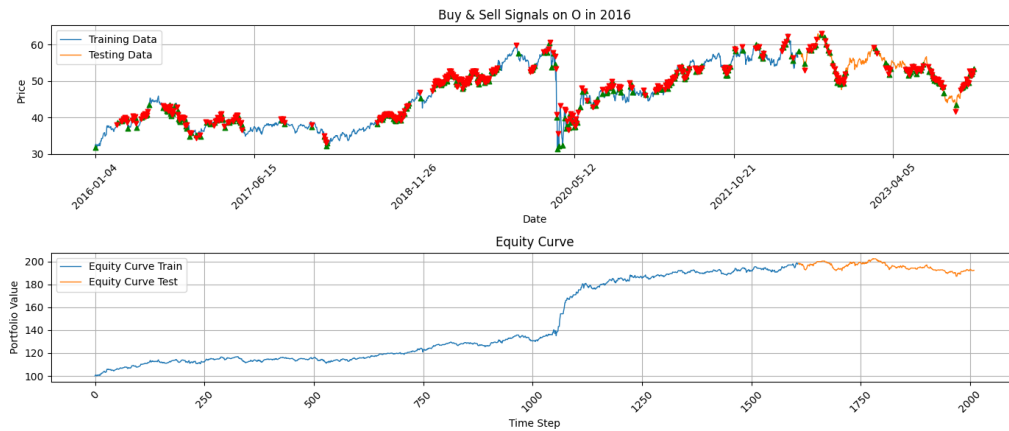


Figure 13.8: Performance of the Transformer Model on Real income data from 2016 with commission

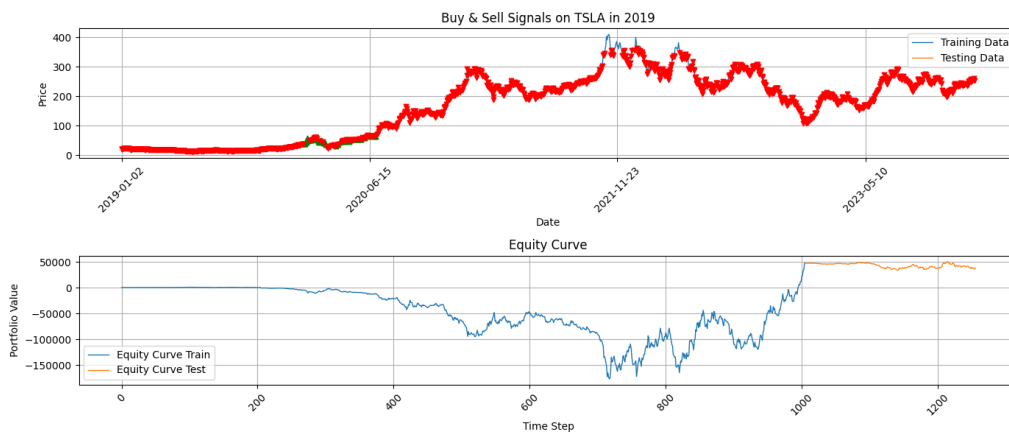


Figure 13.9: Performance of the Transformer Model on Tesla data from 2019 with commission

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 environnement 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 be described as an encoder-decoder transformer like so :

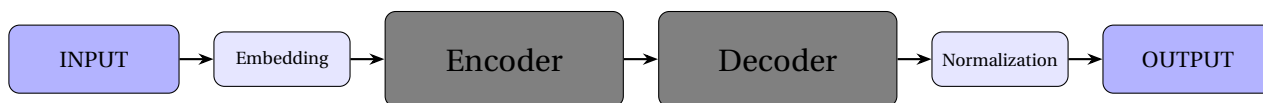


Figure 13.10: Encoder-Decoder Stack of the transformer model

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.

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 :

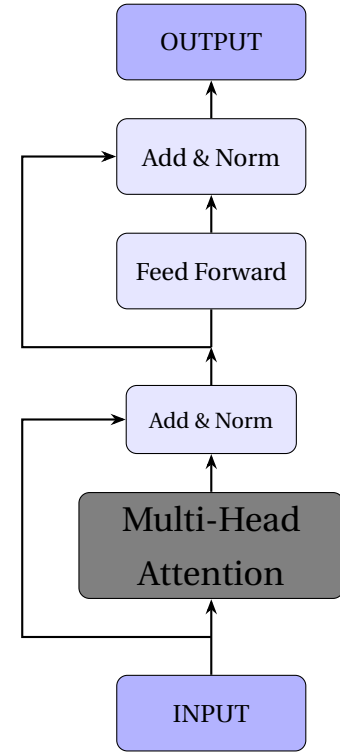


Figure 13.11: Encoder stack

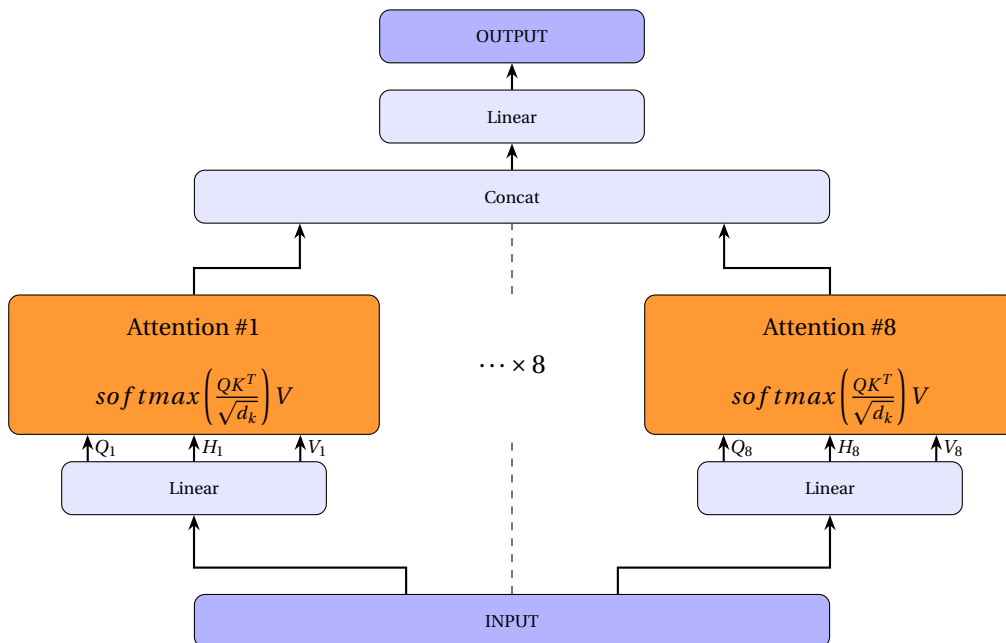


Figure 13.12: Multi-Head Attention stack (x8 heads)

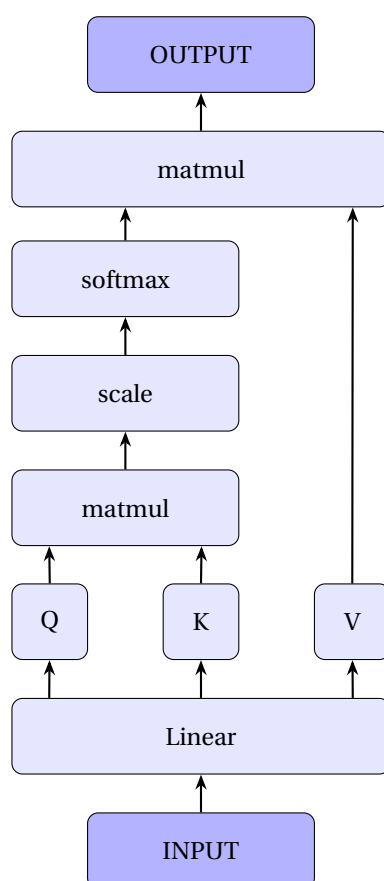


Figure 13.13: Attention stack

