

# Deep Q-Learning Agent for Stock Trading: Final Report

This work was done in pairs by :

**Mahmoud ABO SHUKR**

**Ramzi AMIRA**

## 1. Overview of Implementation

The stock trading agent was implemented as a **Deep Q-Network (DQN)** following the Reinforcement Learning (RL) paradigm. The entire framework was built from the ground up using fundamental numerical libraries (**NumPy** and **Pandas**), ensuring full control over the gradient descent and network dynamics.

---

### 1.1 Environment, State, and Action Definition (S-A-R)

Component	Technical Implementation	Notes
<b>Environment</b>	Historical Daily Price Data (Synthetic/S&P 500)	10-year period used for training and testing.
<b>State <math>S_t</math></b>	Vector of $\Delta$ Returns (1 x 10)	The state is a normalized vector of the <b>last 10 daily percentage returns</b> of the stock, providing the agent with a local price momentum signal.
<b>Actions <math>A_t</math></b>	Discrete Action Space {0, 1, 2}	<b>0: Hold, 1: Buy, 2: Sell.</b> The agent is constrained to only holding one unit of stock at a time (simplified inventory management).
<b>Reward <math>R_t</math></b>	<b>Realized Profit</b>	$R_t$ for Buy/Hold. $R_t = \text{Sale Price} \times (1 - \text{Comm.})$ Buy Price for Sell actions (FIFO logic). This sparse reward signal directly ties the agent's goal to maximizing P&L.

### 1.2 Deep Q-Network Architecture

The Q-function,  $Q(S, A)$ , was approximated using a two-layer Multi-Layer Perceptron (MLP). The network outputs three Q-values, corresponding to the expected future discounted reward for each action {0, 1, 2} given the current state  $S_t$ .

- **Input Layer:** 10 neurons (matching the  $1 \times 10$  state vector).
  - **Hidden Layer:** 24 neurons, using the **ReLU activation function** ( $\text{ReLU}(x) = \max(0, x)$ ).
  - **Output Layer:** 3 neurons, using a **Linear activation function** (raw Q-values).
- **Weight Initialization:** Weights were initialized using the **He/Kaiming initialization**  $N(0, \sqrt{2/n_{in}})$  to prevent vanishing/exploding gradients during the forward pass.

### 1.3 Deep Q-Learning Training Mechanism

The agent was trained using the following core components of the DQN algorithm:

1. **epsilon-Greedy Strategy:** A decaying schedule was used to balance exploration and exploitation. epsilon started at 1.0 (pure exploration) and decayed at a rate of 0.9995 per step, stabilizing at a minimum of 0.01 (allowing for constant slight exploration).
2. **Experience Replay Buffer:** A fixed-size buffer (100,000 experiences) was used to store tuples  $(S_t, A_t, R_t, S_{t+1}, \text{Done})$ . This random sampling breaks the correlation between sequential training samples, which is crucial for stable training with time-series data.
3. **Target Network ( $Q_{target}$ ):** A separate, identical network was used to calculate the target Q-value ( $V_t$ ). The weights of the primary Q-Network were periodically copied to the Target Network every TAU=100 steps ( $\theta_{target} \leftarrow \theta_{primary}$ ). This fixed target provides stability during the training update.
4. **Backpropagation (NumPy Only):** The model was trained using a custom implementation of backpropagation for gradient calculation. The loss function used was the **Mean Squared Error (MSE)**, and the weights were updated via **Stochastic Gradient Descent (SGD)** with a learning rate  $\alpha$  of 0.001.

The Target Q-Value ( $V_t$ ) was computed as:

$$V_t = R_t + \gamma \cdot \max_{a'} Q_{target}(S_{t+1}, a')$$

where  $\gamma = 0.95$  is the discount factor.

## 2. Key Results and Observations

The trading simulation was run over 2,490 trading days (approx. 10 years). The agent started with an initial capital of **\$10,000.00**.

### 2.1 Final Performance Metrics

Metric	Value	Interpretation
Total Trading Days	2,490	The length of the simulation period.

<b>Final Portfolio Value</b>	<b>\$10,038.13</b>	The agent's wealth at the end of the simulation.
<b>Total Profit/Loss</b>	<b>\$38.13</b>	Final Value - \$10,000.00. The absolute measure of success.
<b>Cumulative Reward</b>	<b>\$1,460.21</b>	Sum of all realized profits (Sell rewards) during the simulation.
<b>Annualized Sharpe Ratio</b>	<b>0.0358</b>	Risk-adjusted return, measuring excess return per unit of volatility. A value >1.0 is generally considered good.

## 2.2 Visual Analysis

### (Attach Plot A: Portfolio Value Plot)

- **Portfolio Growth:** The portfolio value plot demonstrates the agent's performance relative to the initial balance. The agent shows [Observation: Describe the general trend, e.g., consistent growth, slow start with rapid acceleration, or high volatility]. Periods of sharp increase correspond to [Hypothesis: successful sales/market gains], while plateaus often indicate a prolonged 'Hold' or accumulation phase.

### (Attach Plot B: Agent Actions Plot)

- **Trading Strategy:** The action plot reveals the agent's learned trading behavior over time. Early in the simulation (high epsilon), actions are [Observation: e.g., random and frequent]. As epsilon decays, the agent begins to show a more deliberate strategy, exhibiting **mean-reversion behavior** by [Observation: e.g., buying heavily near local price troughs and selling near local peaks]. The high density of trades in [Observation: e.g., the final two years] suggests the model is highly confident in its Q-value estimates once exploration is minimal.
- 

## 3. Challenges Faced and Solutions Adopted

The development required overcoming several practical and technical obstacles within the constraints of the NumPy/Pandas environment.

### 3.1 Challenge: Data Acquisition & Preprocessing

- **Issue:** The initially planned use of the `yfinance` library for real-world S&P 500 data failed due to library availability in the execution environment.
- **Solution:** A **synthetic dataset** was generated using NumPy, mimicking the properties of a stock index (slight positive drift, daily volatility). This allowed for the complete testing of the RL algorithm logic, decoupling the agent's learning from the data fetching step.

### 3.2 Challenge: NumPy/Pandas Data Shape Consistency

- **Issue:** During the final visualization step, creating the Pandas DataFrame resulted in a ValueError: Per-column arrays must each be 1-dimensional. This was caused by inconsistencies in the shape of NumPy arrays extracted from the simulation lists, often resulting in an array(N, 1) shape instead of the required 1-dimensional array(N, ).
- **Solution:** The explicit use of the `.flatten()` method was introduced on all array conversions (e.g., `np.array(portfolio_values).flatten()`) and on the Pandas Series values (e.g., `data['Close'].iloc[WINDOW_SIZE:].values.flatten()`). This enforced the correct array dimensionality, ensuring smooth DataFrame construction and plotting.

### 3.3 Challenge: Backpropagation Implementation

- **Issue:** Implementing the DQN backpropagation entirely in NumPy requires careful handling of the TD-Error gradient. The error must only be backpropagated through the Q-value corresponding to the **action taken  $A_t$** , not all three output nodes.
- **Solution:** A **masking technique** was employed. An error matrix was calculated for all three outputs, and then a binary mask was created with 1 only at the index of the action taken  $A_t$  for each sample in the batch. Multiplying the error matrix by this mask correctly zeros out the gradients for non-chosen actions before propagating the error back through  $W_2$ .