Unlocking the Power of Deep Q Networks in Portfolio Optimization

Peng Liu*

April 24, 2025

Abstract

Deep Q Networks (DQNs) represent a significant advancement in reinforcement learning, utilizing neural networks to approximate the optimal Q-value for guiding sequential decision processes. This paper presents a comprehensive introduction to reinforcement learning principles, delves into the mechanics of DQNs, and explores its application in portfolio optimization. By evaluating the performance of DQNs against traditional benchmark portfolios, we demonstrate its potential to enhance investment strategies. Our results underscore the advantages of DQNs in dynamically adjusting asset allocations, offering a robust portfolio management framework.

Key Words

Deep reinforcement learning; deep Q networks; portfolio optimization; multi-period optimization

Key Messages

- Our study shows that DQN-based portfolio optimization strategies, particularly Double DQN, consistently outperform traditional benchmarks such as Equal-Weighted and Global Minimum Variance portfolios in both training and testing periods, highlighting their effectiveness in dynamic and adaptive investment strategies.

*Lee Kong Chian School of Business, Singapore Management University, Singapore. Email: liupeng@smu.edu.sg

- The integration of experience replay and target networks, along with the dueling architecture, significantly enhances the stability and accuracy of DQNs. These techniques mitigate issues like overestimation bias and correlated updates, leading to more reliable and robust portfolio management models.

- DQNs can dynamically adjust asset allocations in response to evolving market conditions by learning optimal actions from a multi-period perspective. This adaptability optimizes returns and effectively manages risks, making DQNs a valuable tool for modern portfolio management in volatile financial markets.

Reinforcement learning (RL) is a branch of machine learning dedicated to determining the optimal actions an agent should take in an environment to maximize long-term cumulative returns. A fundamental technique within RL is Q-learning, which seeks to learn the optimal action-value function, known as the Q-function. This function estimates the expected utility of taking a specific action in a given state and subsequently following a particular policy. Once the Q-values for each state-action pair are known, an optimal policy can be constructed by adopting a greedy approach, where the action with the highest Q-value is selected in each state.

Q-learning operates by iteratively updating Q-values based on the Bellman equation, which decomposes the value of a state-action pair into the immediate reward plus the discounted future reward. This method enables an agent to learn the long-term value of actions through trial and error, progressively refining its policy to converge on an optimal strategy.

However, traditional Q-learning faces significant limitations when dealing with large state-action spaces, which are common in complex environments such as financial markets. For example, both the observed stock price as state and the allocation weights as action are continuous and thus allow for infinite values. The exponential growth of possible states and actions makes it infeasible to store and update Q-values for every possible pair. To address this, Deep Q Networks (DQN) integrate Q-learning with deep neural networks, allowing for the approximation of the Q-function in high-dimensional state spaces. In DQN, a neural network is trained to predict Q-values, enabling the agent to generalize from seen to unseen states. This generalization is crucial in dynamic environments like finance, where the agent must adapt to new market conditions and asset behaviors.

In the realm of finance, portfolio optimization is a critical task involving the allocation of assets to maximize return while minimizing risk. Traditional methods, such as the mean-variance optimization proposed by Markowitz, rely on static assumptions about returns and risks. However, financial markets are dynamic and often exhibit non-stationary behavior, necessitating adaptive strategies that can respond to changing conditions. Applying DQN to portfolio optimization leverages its ability to adaptively learn and adjust asset allocations in response to market dynamics, without the explicit step of forecasting the expected returns and covariance matrix. By modeling portfolio management as a sequential decision-making problem, DQN can optimize asset allocations over time, balancing the trade-off between exploring new strategies and exploiting known profitable ones. This approach allows DQN to continually refine its strategy as new data becomes available, making it highly suited for the unpredictable and ever-changing nature of financial markets. Moreover, DQN's capacity to handle large, complex state spaces makes

it particularly effective in capturing the intricate relationships between different assets, ultimately enhancing the robustness and performance of portfolio management strategies.

This paper explores the application of DQN in portfolio optimization, detailing the underlying principles of reinforcement learning and Q-learning, the architecture and training methodology of DQN, and the evaluation of its performance against traditional benchmark portfolios. Through empirical analysis, we demonstrate the potential of DQN in enhancing investment strategies, providing a robust and adaptive framework for modern portfolio management.

In the following sections, we begin by reviewing the foundations of reinforcement learning, including Markov Decision Processes (MDP) and the concepts of optimal state and value functions. We then introduce DQN and discuss their integration with neural networks to approximate the Q-function in high-dimensional spaces. Following this theoretical groundwork, we present experiments applying DQN to portfolio optimization and compare its performance with traditional benchmark portfolio policies. These comparisons highlight the practical advantages and potential of DQN in financial applications. Additionally, we provide detailed implementation insights and guidelines for practitioners, ensuring that our findings can be effectively replicated and utilized in real-world portfolio management scenarios. See the accompanying notebook for a detailed implementation of our experiments (to be released after publication).

1 Markov Decision Process

Reinforcement learning is a general approach to solving reward-based problems, where an agent learns to make decisions by interacting with the environment. At the core of RL is the Markov Decision Process (MDP), a mathematical framework used to model decision-making problems. This section introduces the key concepts of MDPs and explores their application in portfolio optimization.

1.1 Markov Process and Markov Decision Process

A Markov process describes a sequence of states with the property that the probability of transitioning to the next state depends only on the current state, not the preceding states. Formally, a Markov process is defined as a tuple (S, P), where S is a finite set of states, and P is a state transition probability matrix, where $P_{ss'} = P[S_{t+1} = s'|S_t = s]$, which specifies the

probability of transitioning from the current state s to the next state s'.

To model decision-making problems, we can extend Markov processes to Markov Decision Processes (MDP). An MDP additionally incorporates actions, rewards, and discount factors into a tuple (S, A, P, γ, R), where:

- S is a finite set of states.
- A is a finite set of actions.
- P is the state transition probability matrix, where $P_{ass'} = P[S_{t+1} = s' | S_t = s, A_t = a]$ specifies the probability of transitioning to the next state s' given the current state s and action a.
- $\gamma \in [0, 1]$ is the discount factor.
- $R : S \times A \to \mathbb{R}$ is a reward function that specifies the reward obtained for a given action taken in a given state.

The dynamics of an MDP proceed as follows: Starting in some state s_0, the agent selects an action $a_0 \in A$. As a result, the state transitions to s_1 according to the probability $P_{a_0 s_0 s_1}$, and the agent receives a reward $R(s_0, a_0)$. This process repeats, forming a sequence of states, actions, and rewards.

1.2 Return, Policy, and Value Functions

In reinforcement learning, the goal is to choose actions over time to maximize the expected long-term return G, defined as the total discounted reward from a given time step t onward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{1}$$

A *policy* π is a distribution over actions given states, $\pi(a|s) = P[A_t = a | S_t = s]$. The objective is to find the optimal policy π^* that maximizes the expected return from each state.

There are two crucial value functions that help in determining the optimal policy:

- The *state-value function* $v_\pi(s)$, representing the expected return from

state s under policy π:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \\ &= \sum_{a\in A} \pi(a|s) \sum_{s'\in S} P_{ass'} \left(R_{as} + \gamma v_\pi(s')\right). \end{aligned} \tag{2}$$

- The *action-value function* $q_\pi(s, a)$, representing the expected return from state s taking action a and thereafter following policy π:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \\ &= \sum_{s'\in S} P_{ass'} \left(R_{as} + \gamma \sum_{a'\in A} \pi(a'|s') q_\pi(s', a')\right) \\ &= \sum_{s'\in S} P_{ass'} \left(R_{as} + \gamma v_\pi(s')\right). \end{aligned} \tag{3}$$

The relationship between these functions is given by:

$$v_\pi(s) = \sum_{a\in A} \pi(a|s) q_\pi(s, a). \tag{4}$$

1.3 Bellman Equations

The Bellman equations provide recursive decompositions for value functions, offering a way to express the value of a state or state-action pair in terms of immediate rewards and the value of successor states. These equations form the foundation for many reinforcement learning algorithms, including policy evaluation and improvement.

The *state-value function* $v_\pi(s)$ is given by:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\ &= \sum_{a\in A} \pi(a|s) \sum_{s'\in S} P_{ass'} \left(R_{as} + \gamma v_\pi(s')\right). \end{aligned} \tag{5}$$

The *action-value function* $q_\pi(s, a)$ is given by:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma \sum_{a' \in A} \pi(a'|s') q_\pi(s', a') \right) \\ &= \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma v_\pi(s') \right). \end{aligned} \tag{6}$$

These recursive equations can be used in an iterative manner to evaluate a given policy. The iterative policy evaluation method involves repeatedly updating the value functions using the Bellman equations until they converge to the true value functions. The process is as follows:

- Initialize $v_\pi(s)$ arbitrarily (e.g., $v_\pi(s) = 0$ for all s).
- Repeat until v_π converges:

$$v_\pi^{(k+1)}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma v_\pi^{(k)}(s') \right). \tag{7}$$

Similarly, for the action-value function $q_\pi(s, a)$:

- Initialize $q_\pi(s, a)$ arbitrarily.
- Repeat until q_π converges:

$$q_\pi^{(k+1)}(s,a) = \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma \sum_{a' \in A} \pi(a'|s') q_\pi^{(k)}(s', a') \right). \tag{8}$$

When the state and action spaces are small, we can solve for the closed-form solution for a finite MDP with a given policy π by solving the Bellman equations as a system of linear equations. The closed-form Bellman equation for the state-value function is:

$$v_\pi = (I - \gamma P_\pi)^{-1} R_\pi, \tag{9}$$

where:

- I is the identity matrix.
- P_π is the state transition matrix under policy π.
- R_π is the reward vector under policy π.

The action-value function q_π can be expressed in closed form using the state-value function v_π. The equation is:

$$q_\pi = R + \gamma P_\pi v_\pi = R + \gamma P_\pi (I - \gamma P_\pi)^{-1} R_\pi \tag{10}$$

This closed-form solution indicates that the action-value function can be derived directly from the immediate rewards and the discounted state-value function. Essentially, it combines the immediate reward of taking a specific action in a given state with the expected future rewards, discounted by γ.

While these closed-form solutions are exact, they are typically impractical for large state spaces due to the computational complexity of inverting matrices. See appendix for the derivation of these closed-form solutions.

1.4 Optimal Value Functions and Policies

The optimal state-value function $v^*(s)$ is the maximum value function over all policies:

$$v^*(s) = \max_\pi v_\pi(s). \tag{11}$$

The optimal action-value function $q^*(s, a)$ is the maximum action-value function over all policies:

$$q^*(s, a) = \max_\pi q_\pi(s, a). \tag{12}$$

The Bellman optimality equations are:

$$v^*(s) = \max_{a \in A} \mathbb{E}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a], \tag{13}$$

$$q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a]. \tag{14}$$

1.5 Q-learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function, $q^*(s, a)$, which provides the maximum expected utility for each state-action pair (s, a). The core idea of Q-learning is to start from a random set of initial values and iteratively update the Q-values using the Bellman optimality equation. This allows the agent to learn the long-term value of actions through direct interaction with the environment, without requiring a model of the environment's dynamics.

The Q-learning algorithm involves the following steps:

1. Initialize the Q-values arbitrarily, $Q(s, a) \forall s \in S, a \in A$. Common practice is to initialize all Q-values to zero.

2. Observe the current state s.

3. Select an action a based on the current Q-values, typically using an ϵ-greedy policy:

   - With probability ϵ, select a random action (exploration).
   - With probability $1 - \epsilon$, select the action with the highest Q-value (exploitation): $a = \arg\max_a Q(s, a)$.

4. Execute the action a and observe the reward r and the next state s'.

5. Update the Q-value for the state-action pair (s, a) using the Q-learning update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{15}$$

   where α is the learning rate and γ is the discount factor.

6. Set the current state s to the next state s'.

7. Repeat steps 3-6 until a terminal state is reached.

The Q-learning algorithm can converge to the optimal action-value function $q^*(s, a)$ under certain technical conditions[1]. In practice, the -greedy policy balances exploration and exploitation, ensuring that the agent explores the state-action space sufficiently while gradually shifting toward exploiting the learned Q-values.

However, while effective in many environments, traditional Q-learning introduced above faces significant challenges when applied to high-dimensional or continuous state-action spaces. For example, the size of the Q-table used to store each state-action pair grows exponentially with the number of states and actions, making it infeasible to store and update Q-values for every possible state-action pair in large or continuous spaces.

To address these limitations, researchers have developed extensions and variants of Q-learning that incorporate function approximation techniques, such as deep neural networks, to handle large and continuous state-action spaces. One of the most prominent and successful extensions is the Deep Q-Network (DQN), which integrates Q-learning with deep learning to enable efficient learning and generalization in high-dimensional environments.

[1]These include: the learning rate α is sufficiently small but does not decay too quickly; that is, $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$; the environment is stationary, meaning the transition probabilities and reward distributions do not change over time.

1.6 Application of RL to Portfolio Optimization

Recent research has extensively explored the application of Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) in portfolio optimization, highlighting its potential to outperform traditional methods. Studies such as Fernando Acero and Veloso (2024) provide comprehensive comparisons between DRL and MVO, emphasizing the advantages of DRL in handling complex financial environments and incorporating ESG objectives. The work by Wang and Liu (2023) introduces the industry-aware application of DRL. Additionally, Sen (2023) discusses the integration of financial indicators and hierarchical risk parity approaches to enhance portfolio management. These studies collectively underscore the transformative potential of RL and DRL in modern portfolio optimization, offering insights into their practical implementations and comparative performance against traditional optimization techniques.

The objective of portfolio optimization is to maximize the cumulative return of a portfolio over a given time horizon. This problem can be formulated as a dynamic control optimization problem where the agent dynamically adjusts the portfolio weights in response to market conditions. Formally, we define the problem as follows:

$$\begin{aligned} \text{Maximize} \quad & \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R_t\right] \\ \text{subject to} \quad & a_t = \pi(s_t), \end{aligned}$$

where R_t is the portfolio return at time t, γ is the discount factor, s_t is the state representing the market conditions and portfolio composition at time t, and a_t is the action representing the adjustment to the portfolio weights. The policy $\pi(s_t)$ maps states to actions and is typically parameterized by a deep neural network.

In this setup, the expected return $\mathbb{E}[R_t]$ is maximized over a series of time steps, capturing the sequential nature of portfolio management. The use of deep networks for the policy function π allows for capturing complex relationships between market conditions and optimal portfolio adjustments.

When using a neural network to approximate the Q-value function, we can learn optimal policies in high-dimensional state spaces. The optimization problem can now be expressed as:

$$\text{Maximize} \quad \mathbb{E}\left[\prod_{t=1}^{T} (1 + \langle \hat{a}_t, \vec{r_t} \rangle)\right]$$

subject to $a_t = \pi_\theta(s_t)$, where $\hat{a_t}$ denotes the portfolio weights at time t and $\vec{r_t}$ represents the returns vector of the assets. The policy π_θ is parameterized by the network weights θ.

Compared to traditional portfolio optimization methods, RL-based approach offers several advantages:

1. **Optimization of Function** π: Traditional methods often optimize portfolio weights directly, while DQN optimizes a policy function π that dynamically adjusts weights based on state observations. This shift from static to dynamic optimization allows the policy to adapt to changing market conditions.

2. **Multi-step Optimization**: DQN performs multi-step optimization over the entire investment horizon, rather than solving a series of single-step optimization problems. This holistic approach better captures the temporal dependencies and compounding effects of investment decisions.

3. **Handling of Large State Spaces**: The use of deep networks enables handling large and complex state spaces, capturing intricate relationships between assets and market conditions that traditional methods might miss.

In portfolio optimization, we can model the problem as a Markov Decision Process (MDP) where:

- *States* represent the current portfolio composition and market conditions. This includes historical asset prices, technical indicators, macroeconomic variables, and other relevant financial data.

- *Actions* are the possible adjustments to the portfolio. These can involve trading strategies such as buying, selling, or holding assets, or adjusting the allocation weights across different assets in the portfolio.

- *Rewards* are the returns from the portfolio. The reward function can be defined in various ways, such as raw returns, risk-adjusted returns like the Sharpe ratio, or other performance metrics that account for both returns and risk.

- *Transition probabilities* capture the market dynamics and the resulting changes in portfolio value due to the actions taken. These probabilities are generally unknown in real-world markets and must be estimated or learned from historical data.

By defining the portfolio optimization problem as an MDP, we enable the RL agent to interact with the market environment and learn an optimal policy for managing the portfolio. The agent will then aim to maximize cumulative rewards by continuously adjusting the portfolio in response to changing market conditions.

Given the infinite-dimensional nature of the state space, it is impractical to store and learn optimal action-value functions using traditional algorithms like Q-learning. A suitable learning mechanism is required to generalize the learned action-value function to unseen states. In the following section, we introduce the Deep Q-Network (DQN) algorithm, which combines Q-learning with deep neural networks to address this challenge.

2 Deep Q Networks

Deep Q Networks (DQNs) represent a significant advancement in reinforcement learning by combining the principles of Q-learning with the function approximation capabilities of deep neural networks. This approach facilitates effective learning and generalization in high-dimensional state spaces, which is particularly crucial in complex environments such as financial markets. In the following sections, we will first introduce the architecture of DQNs, as initially proposed by (Mnih et al., 2015). Subsequently, we will highlight two key advancements: experience replay and target networks. Additionally, we will discuss the dueling architecture, an important enhancement to the DQN framework introduced by (Wang et al., 2016).

2.1 Architecture of DQNs

The core component of a DQN is a deep neural network, typically a feed-forward neural network, that takes a state s as input and outputs Q-values (state-action values) for all possible actions in that state. The architecture can vary, but a common choice is a multi-layer perceptron (MLP) with several hidden layers.

- **Input Layer:** The input layer consists of neurons corresponding to the features representing the state. In the context of portfolio optimization, these features could include asset prices, technical indicators, macroeconomic variables, and other relevant financial data.
- **Hidden Layers:** The hidden layers are fully connected layers with activation functions such as ReLU (Rectified Linear Unit). These layers

enable the network to learn complex, non-linear representations of the state-action value function and act as hidden feature extractors.

- **Output Layer:** The output layer consists of neurons corresponding to the Q-values for each possible action. In a discrete action space, each action will have a corresponding Q-value, representing the expected return of taking that action in the given state.

Training a DQN involves iteratively updating the network's weights to minimize the difference between the predicted Q-values and the target Q-values. The target Q-value is computed using the Bellman optimality equation, which incorporates the immediate reward and the maximum Q-value of the next state:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

Training a DQN involves two main steps:

1. **Training Data Collection:** Initialize the Q-network $Q(s, a; \theta)$ with random weights θ and enter a finite loop to interact with the environment following an exploration-exploitation strategy (e.g., -greedy policy), producing experience tuples (s_t, a_t, r_t, s_{t+1}).

2. **Model Training:**

   - Compute the target Q-value based on the Bellman optimality equation at time t:

   $$y_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) \tag{16}$$

   - Compute the mean squared error (MSE) $L(\theta)$ between the predicted Q-value and the target Q-value:

   $$L(\theta) = \mathbb{E}\left[(y_t - Q(s_t, a_t; \theta))^2\right]$$

   - Update the network weights using gradient descent to minimize the training loss:

   $$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

   where α is the learning rate, and $\nabla_\theta L(\theta)$ is the gradient of the loss with respect to the network weights:

   $$\nabla_\theta L(\theta) = (y_t - Q(s_t, a_t; \theta))\, \nabla_\theta Q(s_t, a_t; \theta)$$

- In other words, the network weights are updated via:

$$\theta \leftarrow \theta + \alpha \left(y_t - Q(s_t, a_t; \theta) \right) \nabla_\theta Q(s_t, a_t; \theta) \tag{17}$$

Note that the DQN agent updates the weights and, therefore, the Q-values using experiences gathered in a sequential manner. This can lead to highly correlated updates, which may cause the learning algorithm to diverge or converge prematurely. We introduce experience replay as a remedy to correlated training samples in the next section.

2.2 Experience Replay

Experience replay is used in training DQNs to improve learning stability (via random sampling) and efficiency (experience reuse across multiple updates). The core idea is to store the agent's experiences at each time step in a replay buffer and then sample a random mini-batch of experiences from this buffer to update the neural network. This approach helps to break the correlation between consecutive experiences, which can destabilize the learning process if they are highly correlated.

Specifically, during the data collection step, a replay buffer $\mathcal{D}$ is initialized to store experience tuples (s_t, a_t, r_t, s_{t+1}). The buffer can hold a fixed number of experiences and operates on a first-in, first-out (FIFO) basis when it reaches capacity. At each learning step, a mini-batch B of N experiences $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{N}$ is randomly sampled from the replay buffer $\mathcal{D}$. This random sampling helps to break the temporal correlations between consecutive experiences. For each sampled experience (s_i, a_i, r_i, s_{i+1}) in B, the target Q-value is computed using the same Bellman optimality equation in Equation 16. Finally, we update the network weights to minimize the empirical MSE for this mini-batch:

$$L(\theta) = \frac{1}{|B|} \sum_{(s_i, a_i, r_i, s_{i+1}) \in B} \left(y_i - Q(s_i, a_i; \theta) \right)^2 \tag{18}$$

The next section introduces another key innovation in DQN: target network and the double DQN architecture.

2.3 Target Networks and Double DQN

In the original DQN setup, the Q-values are updated using the Bellman equation, which incorporates the immediate reward and the maximum Q-value of the next state. However, using the same network to compute both the current and target Q-values can lead to instability. Small changes in the

Q-network can lead to large changes in the target values, causing oscillations and divergence in the training process. To further stabilize the training of DQNs, a target network can be used, as proposed by Van Hasselt et al. (2016), which is a separate network with the same architecture as the main Q-network, but its weights are updated less frequently.

By introducing a target network, we can mitigate this issue by providing more stable target values. The periodic update in the target network's weights ensures that the target values change more slowly and smoothly, leading to more stable and reliable updates. This helps to reduce the oscillations and divergence that can occur during training. Specifically, a target network $Q'(s, a; \theta^-)$ is introduced, where θ^- represents the weights of the target network. These weights are periodically updated to match the weights of the main Q-network θ. For each sampled experience (s_i, a_i, r_i, s_{i+1}) in B, the target Q-value is now computed using the target network:

$$y_i = r_i + \gamma \max_{a'} Q'(s_{i+1}, a'; \theta^-)$$

Double DQN further improves upon this by addressing the overestimation bias inherent in the standard DQN algorithm. In Double DQN, the action that maximizes the Q-value is selected using the main network, but the Q-value of this action is evaluated using the target network. This decouples the selection and evaluation of actions, thereby reducing overestimation bias and leading to more accurate Q-value estimates.

The target Q-value in Double DQN is computed as follows:

$$y_i = r_i + \gamma Q'(s_{i+1}, \arg\max_{a'} Q(s_{i+1}, a'; \theta); \theta^-)$$

This modification ensures that the action selection uses the most recent Q-values from the main network while the value estimation uses the more stable Q-values from the target network. The weights of the target network θ^- are then periodically updated to match the weights of the main Q-network θ:

$$\theta^- \leftarrow \theta$$

These changes introduced by the target network and Double DQN help to stabilize training by providing consistent targets for the Q-value updates, reducing the risk of divergence, and improving the overall learning process.

The next section introduces another improvement called the dueling DQN.

2.4 Dueling DQN

Dueling DQN is an enhancement to the standard DQN architecture that aims to improve learning efficiency by more accurately estimating the value of each action. Introduced by (Wang et al., 2016), the dueling architecture separates the estimation of the state value and the advantage for each action. This decomposition allows the network to learn which states are valuable independently of the chosen action, which is particularly useful in environments where some actions do not significantly affect the outcome.

In many reinforcement learning problems, especially those with large action spaces, knowing which states are valuable, irrespective of the action taken, is beneficial. The standard DQN architecture conflates the value of a state with the value of the state-action pair, which can lead to inefficient learning. By separating these two components, the dueling architecture can provide a more robust estimate of the state value and the advantages of different actions, leading to improved performance.

The key innovation in dueling DQN is that it is split into two separate streams within the network architecture: one for estimating the state value and one for estimating the advantage of each action. These two streams are then combined to produce the final Q-values.

The architecture of a dueling DQN consists of the following components:

- **Shared Feature Extractor:** The network's initial layers are shared, and features are extracted from the input states. These layers are typically fully connected and have activation functions such as ReLU.
- **Value Stream:** One stream estimates the state value $V(s;\theta,\beta)$, where β represents the parameters of the value stream.
- **Advantage Stream:** The other stream estimates the advantage of each action $A(s,a;\theta,\kappa)$, where κ represents the parameters of the advantage stream. Note that the value stream $V(s;\theta,\beta)$ outputs a single scalar value, while the advantage stream $A(s,a;\theta,\kappa)$ outputs a vector of advantage values, one for each action.
- **Combining Layer:** The state value and advantage streams are combined to produce the final Q-values. The typical way to combine them is:

$$Q(s,a;\theta,\beta,\kappa) = V(s;\theta,\beta) + \left(A(s,a;\theta,\kappa) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\kappa) \right)$$

  The de-meaning operation in the formulation ensures that the advantage function has zero mean, which helps to stabilize learning.

The loss function remains the same MSE between the predicted Q-values and the target Q-values, but now the Q-values are computed using the dueling architecture:

$$L(\theta, \alpha, \beta) = \frac{1}{|B|} \sum_{(s_i, a_i, r_i, s_{i+1}) \in B} (y_i - Q(s_i, a_i; \theta, \beta, \kappa))^2$$

The network weights θ, β, κ are updated by minimizing the loss function using gradient descent:

$$\theta, \beta, \kappa \leftarrow \theta, \beta, \kappa - \alpha \nabla_{\theta,\beta,\kappa} L(\theta, \beta, \kappa)$$

where α is the learning rate, and $\nabla_{\theta,\beta,\kappa} L(\theta, \beta, \kappa)$ is the gradient of the loss with respect to the network weights.

The dueling DQN architecture provides more accurate and stable value estimates, especially in environments where certain actions have little impact on the outcome. By decomposing the Q-values into value and advantage components, the network can better capture the underlying structure of the value function, leading to improved performance and faster convergence.

The following section introduces the experiment setup and results using various DQN architectures compared to common benchmark strategies.

3 Experiments and Results

To evaluate the empirical performance of the various DQN architectures discussed, we focused on their application to a selection of leading stocks and compared their performance against common benchmark portfolio strategies, including Equal Weighted (EW) and Global Minimum Variance (GMV) portfolios, as well as a random policy. Our dataset consists of daily adjusted closing prices for four major stocks: 'GOOG', 'MSFT', 'AAPL', and 'TSLA', covering the period from '2014-01-01' to '2024-01-01'. We used the data from the last three years as the test set, while the earlier data was used for training. Exhibit 1 shows the daily cumulative returns for all the stocks, where 'TSLA' exhibits the highest growth but also comes with the greatest volatility.

To set up the custom environment for RL and to enhance the feature set for the RL agent, we calculated a variety of common daily technical indicators, including Simple Moving Average (SMA), Relative Strength Index (RSI), Exponential Moving Average (EMA), Moving Average Convergence Divergence (MACD), Bollinger Bands, Average True Range (ATR), and Stochastic Oscillator. We then resample the data to monthly frequency, computing the cumulative monthly return and volatility and aggregating the

daily indicators to the end of the previous month. The processed dataset provides a comprehensive set of features for the RL environment, ensuring that the agent can learn from a rich array of market signals.

The custom gym environment was configured as follows:

- **Action Space:** The action space is defined as a multi-discrete space representing the allocation weights for each stock. Each weight can take one of a finite number of discrete values, which are then normalized via softmax transformation in the final layer to sum to one.
- **Observation Space:** The observation space is continuous and includes all technical indicators for the current month and the portfolio's current allocation weights.
- **Reward Function:** The reward is calculated as the ratio of the portfolio's monthly return to its volatility, incentivizing higher returns and lower risk.

We used several key metrics to assess the performance of the trained models, including cumulative annual growth rate (CAGR), annual Sharpe ratio, max drawdown, and final net worth. These metrics provide a comprehensive evaluation of the portfolio's performance, balancing return and risk considerations.

In the experiments, we set an initial wealth of 1000, train all RL agents for a total of 300 epochs, and use a total of 10 discretized bins for the action space for each stock. We set a learning rate of 0.005 and a dropout rate of 0.2. We consider a total of six strategies, including three DQN-based strategies (vanilla DQN, Double DQN, and Dueling DQN) and three benchmark strategies (Equal Weighted, GMV, and random policy).

Exhibit 2 shows the cumulative wealth curve of all policies during the training period. DQN and Double DQN performed best, followed by Dueling DQN and equal weighted portfolio. GMVP and random policy performed worst.

For test set performance, as shown in Exhibit 3, Double DQN displays the best generalization performance in the test set, which shows the performance improvement using this specific type of architecture. Note that the equal-weighted portfolio also performed reasonably well compared with other strategies. The random strategy performed the worst in both cases.

To better examine the overall risk-and-return tradeoff, we compile all metrics in Exhibit 4. We note that the GMV portfolio achieves the highest Sharpe ratio and the lowest maximum drawdown during the training period. However, its test performance is suboptimal due to its excessive emphasis on

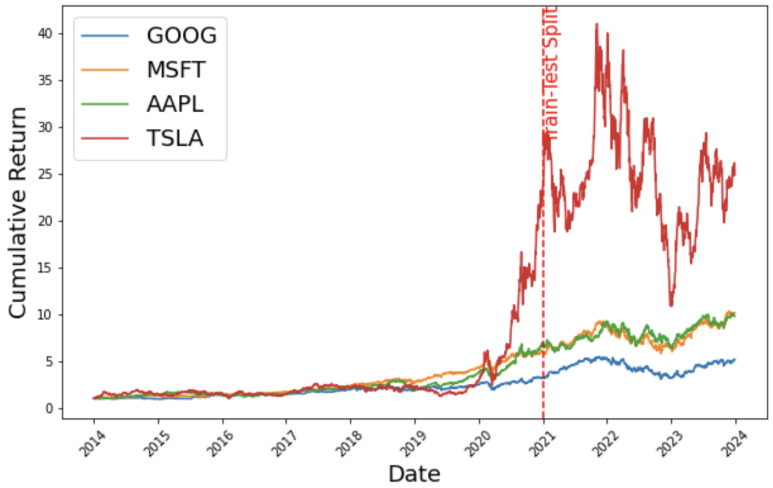

Exhibit 1: Daily cumulative returns of GOOG, MSFT, AAPL, and TSLA.

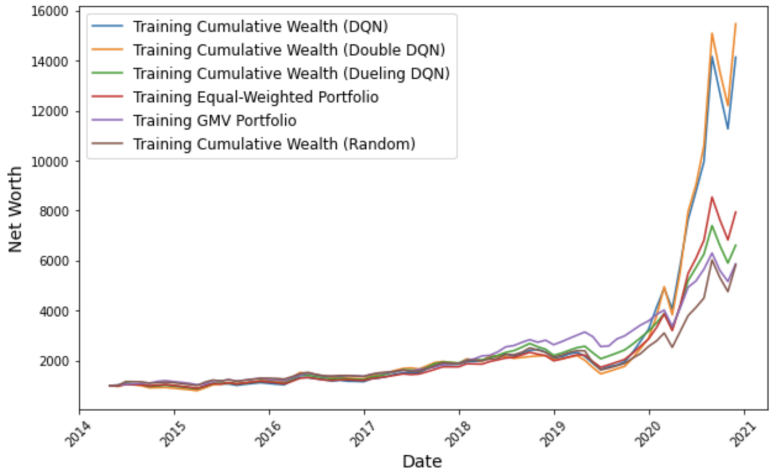

Exhibit 2: Cumulative wealth over time during the training period for all portfolio strategies. DQN and Double DQN performed best, followed by Dueling DQN and equal weighted portfolio. GMVP and random policy performed worst.

risk minimization. On the other hand, DQN-based portfolios, especially Double DQN, perform well in both the training and testing periods. By learning the optimal action from a multi-period perspective based on the dynamic nature of the market, Double DQN is able to obtain the best performance in all metrics during the testing period.

4 Conclusion

In this paper, we demonstrate the potential of DQN-based reinforcement learning policies in enhancing portfolio optimization strategies. By leveraging the dynamic and adaptive nature of reinforcement learning, particularly through the application of DQN and its variants such as Double DQN and Dueling DQN, we have shown that these models can significantly outperform traditional benchmark portfolios under both training and testing conditions.

Our empirical results indicate that while the GMV portfolio exhibits the highest Sharpe ratio and the lowest maximum drawdown during the training period, its performance diminishes during testing due to its overemphasis on risk minimization. Conversely, the DQN-based portfolios, especially the Double DQN, consistently perform well across all metrics during both training and testing periods. This superior performance is attributed to the model's ability to learn optimal actions from a multi-period perspective and adapt to the dynamic nature of financial markets.

The findings from this research underscore the advantages of using DQN in portfolio management. Specifically, Double DQN emerges as the best-performing strategy in the test set, achieving the highest cumulative return, Sharpe ratio, and final net worth while maintaining a relatively low maximum drawdown. These results highlight the efficacy of DQN in capturing the complexities of market conditions and making informed asset allocation decisions.

Furthermore, the integration of experience replay and target networks has been pivotal in stabilizing the training process, reducing overestimation bias, and improving the reliability of the learned policies. The dueling architecture further enhances the learning efficiency by accurately estimating state values and action advantages, making it particularly effective in environments with large action spaces.

In summary, the application of DQNs to portfolio optimization offers a robust and adaptive framework that can dynamically adjust asset allocations in response to evolving market conditions. This approach not only optimizes returns but also manages risks effectively, making it a valuable tool for modern portfolio management. Future research could explore the integra-

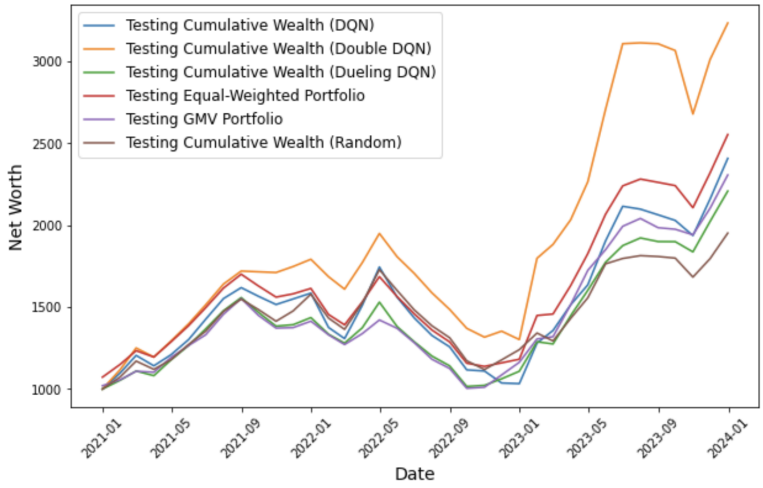

Exhibit 3: Cumulative wealth over time during the test period for all portfolio strategies. Double DQN displays the best generalization performance in the test set, followed by the equal weighted portfolio.

Exhibit 4: Performance metrics for different policies during training and testing periods. The Double DQN policy performs best in the test set across all metrics. Although the GMV portfolio gives the best Sharpe ratio during the training period, its test performance is suboptimal due to excessive emphasis on risk minimization.

Dataset	Policy	CAGR	Annualized Sharpe Ratio	Max Drawdown	Final Net Worth
Training	DQN	0.4953	1.3115	0.2047	14133.21
	Double DQN	**0.5159**	1.2154	0.1874	**15468.63**
	Dueling DQN	0.3323	1.2789	0.2025	6610.03
	Equal-Weighted	0.3645	1.2664	0.2005	7939.56
	GMV	0.3040	**1.4120**	**0.1806**	5869.18
	Random	0.3066	1.0470	0.2100	5814.85
Testing	DQN	0.3405	1.0908	0.2958	2408.94
	Double DQN	**0.4790**	**1.4295**	**0.2003**	**3235.33**
	Dueling DQN	0.3024	1.2354	0.2457	2209.16
	Equal-Weighted	0.3555	1.3527	0.2205	2554.48
	GMV	0.3115	1.3963	0.2376	2307.58
	Random	0.2499	1.0206	0.3123	1952.54

tion of more sophisticated market indicators and the application of advanced reinforcement learning algorithms to further enhance the performance and applicability of DQNs in financial markets.

References

Fernando Acero, Parisa Zehtabi, N. M. M. C. D. M. and Veloso, M. (2024). Deep reinforcement learning and mean-variance strategies for responsible portfolio optimization. *arXiv*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Sen, J. (2023). Portfolio optimization using reinforcement learning and hierarchical risk parity approach. In Rivera, G., Cruz-Reyes, L., Dorronsoro, B., and Rosete, A., editors, *Data Analytics and Computational Intelligence: Novel Models, Algorithms and Applications*, volume 132 of *Studies in Big Data*, pages 509–554. Springer, Cham.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Wang, Y. and Liu, P. (2023). A&I Trader: Integrating Industry Effect into Reinforcement Learning for Balanced Portfolio Management. SSRN Working Paper.

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1995–2003.

5 Appendix

5.1 Deriving the Closed-form Solution of Value Functions

Recall the matrix form of the Bellmen equation for the state value function:

$$v_\pi = R_\pi + \gamma P_\pi v_\pi$$

Here, - v_π is an $|S| \times 1$ vector (where $|S|$ is the number of states), - R_π is an $|S| \times 1$ vector where each element is the expected reward for being in state s under policy π, - P_π is an $|S| \times |S|$ matrix where each element $P_\pi(s, s')$ represents the probability of transitioning from state s to state s' under policy π, - γ is the discount factor.

By isolating v_π on one side, we have:

$$v_\pi - \gamma P_\pi v_\pi = R_\pi$$

Factoring out v_π:

$$(I - \gamma P_\pi)v_\pi = R_\pi$$

Here, I is the identity matrix of size $|S| \times |S|$.

No we can solve for v_π:

$$v_\pi = (I - \gamma P_\pi)^{-1} R_\pi$$

For the action value function, recall that

$$q_\pi(s, a) = \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma \sum_{a' \in A} \pi(a'|s') q_\pi(s', a') \right)$$

Since the state-value function $v_\pi(s)$ is given by:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s, a)$$

Using this relationship, substitute $v_\pi(s')$ into the Bellman equation for $q_\pi(s, a)$:

$$q_\pi(s, a) = \sum_{s' \in S} P_{ass'} \left(R_{as} + \gamma v_\pi(s') \right)$$

Recall that the closed-form solution for the state-value function is:

$$v_\pi = (I - \gamma P_\pi)^{-1} R_\pi$$

Let q_π be the vector of action-values, R be the reward matrix where each entry $R^a_{ss'}$ represents the expected reward for taking action a in state s and transitioning to state s', and P_π be the state transition probability matrix under policy π. The equation for q_π becomes:

$$q_\pi = R + \gamma P_\pi v_\pi$$

Using the closed-form solution for v_π, we have:

$$q_\pi = R + \gamma P_\pi (I - \gamma P_\pi)^{-1} R_\pi$$