Chapter 1: Portfolio Optimization and Reinforcement Learning: An Endgame Perspective

Peng Liu

As the first chapter, we explore the fascinating intersection of portfolio optimization and reinforcement learning from an endgame perspective, presenting you the big picture as well as the necessary know-how and the framework to get things started. Portfolio optimization is a crucial aspect of modern finance, as investors and financial institutions alike strive to maximize returns while minimizing risk. Reinforcement learning, a subfield of machine learning that focuses on agents learning to make decisions in uncertain environments, has recently emerged as a powerful tool to tackle such type of sequential decision-making problems under uncertainty.

The endgame perspective highlights one of the most important considerations in the sequential decision-making process: maximizing the long-term cumulative return. In portfolio management, this means not only aiming for short-term gains but also ensuring long-term utility and adaptability in the face of ever-changing market conditions. Using reinforcement learning as a principled decision engine, we can develop dynamic investment strategies that continuously learn and adapt to the market's complexities. These data-driven policies would help navigate the uncertainty of financial markets.

We will begin by providing a brief overview of portfolio optimization, discussing its key principles and traditional methods such as Modern Portfolio Theory (MPT). We will then delve into reinforcement learning, covering its fundamentals, algorithms, and applications in finance. This background will serve as a foundation for understanding the advantages of combining these two fields to create a more robust and adaptive approach to portfolio management.

Next, we will a practical example on how to use reinforcement learning techniques in dynamic asset allocation. The implementation, which is based on off-the-shelf reinforcement learning algorithms, provides a self-contained framework on developing RL-based policies for portfolio optimization. The coding examples also help enhance our understanding from a pragmatic perspective, demonstrating the real-life snapshot when one develops RL algorithms for automated trading.

This chapter aims to provide a comprehensive understanding of the synergy between portfolio optimization and reinforcement learning, paving the way for more advanced and in-depth discussion in the chapters.

0.1 Introducing Portfolio Optimization

Portfolio optimization is an important topic in modern finance, focusing on the optimal selection (in the form of binary choice) and allocation (in the form of percentages) of assets to construct a well-diversified portfolio that maximizes return while minimizing risk. The foundations of portfolio optimization can be traced back to the seminal work of Harry Markowitz, who introduced the concept of *Modern Portfolio Theory* (MPT) in his 1952 seminal paper (Markowitz 1952). The essence of MPT lies in the quantification of the trade-off between risk and return, both at portfolio level. This leads to the formation of the *efficient frontier*, which represents a set of optimal portfolios offering the highest expected return for a given level of risk, or equivalently, the lowest risk for a target expected return.

Mathematically, portfolio optimization can be framed as an optimization problem, in which the objective is to either maximize the portfolio return for a given level of risk, or minimize the portfolio risk for a given level of return. Denote $\mathbf{w} = [w_1, w_2, \dots, w_n]^T$ as a column vector of portfolio weights, where w_i represents the proportion of capital allocated to asset i, and n is the number of assets under consideration. The expected return and risk of the portfolio are given by the following:

$$E[R_p] = \mathbf{w}^T \mu \tag{1}$$

$$\sigma_p^2 = \mathbf{w}^T \Sigma \mathbf{w}, \tag{2}$$

where $E[R_p]$ is the expected return of the portfolio, σ_p^2 is the portfolio variance (a measure of risk), μ is a column vector of expected returns for each asset, and Σ is the covariance matrix of asset returns.

Let us pause and understand the calculation for the expected returns μ and the covariance matrix Σ when given a matrix of daily asset prices, where the rows indicate the time index and each column represents an asset.

0.1.1 Understanding Asset Return and Covariance Matrix

An asset's return is a measure of its financial performance, often expressed as a percentage change in its value over a particular period, such as a day for daily asset prices. Specifically, given the asset's price at time t, denoted by p_t, the percentage return at time $t+1$, denoted by r_{t+1}, can be calculated as follows:

$$r_{t+1} = \frac{p_{t+1} - p_t}{p_t} = \frac{p_{t+1}}{p_t} - 1.$$

Since calculating the return of the current day requires knowing the previous day's return, the first day will not report a percentage return as there is no previous day. The return series thus starts from the second day onwards, with the first day discarded in later analysis. Alternatively, we can define the logarithmic return (log return):

$$r_{\log,t+1} = \log \frac{p_{t+1}}{p_t} = \log(1 + r_{t+1})$$

Let's denote the historical returns of an asset i over a series of T periods by $r_{i,1}, r_{i,2}, \dots, r_{i,T}$. The expected return for asset i, represented by μ_i, is approximated by the empirical mean of its historical returns:

$$\mu_i = \frac{1}{T} \sum_{t=1}^{T} r_{i,t}.$$

The covariance matrix quantifies the direction and magnitude of co-movement between all pairs of assets in the portfolio. To compute the covariance matrix, we first need to calculate the covariance between the returns of each pair of assets. The covariance between the returns of assets i and j can be calculated as:

$$\mathrm{Cov}(r_i, r_j) = \sigma_{ij} = \frac{1}{T-1} \sum_{t=1}^{T} (r_{i,t} - \mu_i)(r_{j,t} - \mu_j),$$

where μ_i and μ_j are the expected returns of assets i and j, respectively. Note that we divide by $T - 1$ instead of T to provide an unbiased estimate of the population covariance based on the sample covariance. As the sample size T becomes large, the difference between dividing by T and dividing by $T - 1$ becomes negligible.

The covariance matrix Σ is a symmetric $n \times n$ matrix, where each element σ_{ij} represents the covariance between the returns of assets i and j. The diagonal elements of the covariance matrix, σ_{ii}, represent the variances of the individual assets. The covariance matrix can be represented as:

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{bmatrix},$$

where $\sigma_{ij} = \mathrm{Cov}(r_i, r_j)$.

Zooming out at the portfolio level for a total of n assets, the expected return and risk of the portfolio can then be calculated using the expected returns and covariance matrix of the individual assets. Given the portfolio weights $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$, the expected return of the portfolio is the weighted sum of the expected returns of the individual assets. The portfolio variance (i.e., risk) is computed as the quadratic product of the portfolio weights and the covariance matrix.

Note that each entry in the covariance matrix also conveys some information about asset-wise correlation. Specifically, the covariance matrix is related to the asset-wise correlation through a scaling operation involving the standard deviations of the individual assets. The correlation between two assets measures the strength and direction of the linear relationship between their returns. It is a scalar value between -1 and 1, with -1 indicating a perfect negative correlation and 1 representing a perfect positive correlation.

The correlation coefficient between assets i and j, denoted as ρ_{ij}, can be computed using their covariance, σ_{ij}, and the individual standard deviations of their returns, σ_i and σ_j, as follows:

$$\rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j},$$

where $\sigma_i = \sqrt{\sigma_{ii}^2}$ and $\sigma_j = \sqrt{\sigma_{jj}^2}$ are the standard deviations of assets i and j, respectively.

The correlation matrix R can be derived from the covariance matrix Σ using the asset-wise correlations:

$$R = \begin{bmatrix} 1 & \rho_{12} & \cdots & \rho_{1n} \\ \rho_{21} & 1 & \cdots & \rho_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n1} & \rho_{n2} & \cdots & 1 \end{bmatrix},$$

where $\rho_{ij} = \frac{\sigma_{ij}}{\sigma_i \sigma_j}$ for $i \neq j$, and $\rho_{ii} = 1$ for all i (since the correlation of an asset with itself is always 1).

Conversely, we can compute the covariance matrix Σ from the correlation matrix R and the individual assets' standard deviations:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \rho_{12}\sigma_1\sigma_2 & \cdots & \rho_{1n}\sigma_1\sigma_n \\ \rho_{21}\sigma_1\sigma_2 & \sigma_2^2 & \cdots & \rho_{2n}\sigma_2\sigma_n \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n1}\sigma_1\sigma_n & \rho_{n2}\sigma_2\sigma_n & \cdots & \sigma_n^2 \end{bmatrix},$$

where $\sigma_{ij} = \rho_{ij}\sigma_i\sigma_j$ for $i \neq j$, and σ_{ii}^2 (or simply σ_i^2) for all i (the variance of asset i).

In summary, the covariance matrix and the correlation matrix are related through a scaling operation involving the standard deviations of the individual assets. The correlation matrix provides a normalized and bounded view of the linear relationship between asset returns, while the covariance matrix captures the joint variability of those returns.

0.1.2 More on the Portfolio Variance

The portfolio variance, denoted as $\mathbf{w}^T\Sigma\mathbf{w}$, is a positive scalar value that measures the magnitude of the portfolio's risk. A single portfolio variance requires three components: $\mathbf{w}^T$, Σ, and $\mathbf{w}$. Let us look at these components in more detail.

We start with $\mathbf{w}^T$, the transpose of the portfolio weights vector $\mathbf{w}$. $\mathbf{w}^T$ has dimensions $1 \times n$, where n is the number of assets in the portfolio. Now we come to Σ, the covariance matrix of the assets' returns, with dimensions $n \times n$. And finally, we have $\mathbf{w}$, the portfolio weights vector, with dimensions $n \times 1$.

The computation then follows via a series of matrix multiplications, starting with $\mathbf{w}^T\Sigma$. The step gives a $1 \times n$ row vector, where each element of this vector is the sum of the product of the corresponding row elements of $\mathbf{w}^T$ and the column elements of Σ. Next, we calculate $(\mathbf{w}^T\Sigma)\mathbf{w}$, which gives a scalar result computed as the sum of the product of the corresponding elements of the row vector $\mathbf{w}^T\Sigma$ and the column vector $\mathbf{w}$.

In more detail, the portfolio variance can be expressed as:

$$\mathbf{w}^T\Sigma\mathbf{w} = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \sigma_{ij},$$

where w_i and w_j are the weights of assets i and j, respectively, and σ_{ij} is the covariance between the returns of assets i and j. The portfolio variance is thus a scalar value that represents the portfolio's risk.

0.1.3 Mean Variance Optimization

The classical mean-variance optimization problem, as proposed by Markowitz, can be formulated as:

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize}} \quad & \mathbf{w}^T\Sigma\mathbf{w} \\ \text{subject to} \quad & \mathbf{w}^T\mu = \mu_0, \\ & \mathbf{w}^T 1 = 1, \end{aligned}$$

where μ_0 is the user-defined target expected return, and 1 is a column vector of ones. The optimization problem aims to find the portfolio weights $\mathbf{w}$ that minimize portfolio variance, subject to the constraints of achieving the target expected return and fully investing the available capital (i.e., the sum of the weights equals 1).

Note that we often model the first constraint as $\mathbf{w}^T\mu \geq \mu_0$, which requires that the expected portfolio return is no less than the target return.

We can also reformulate the mean-variance optimization problem as maximizing the expected return by adjusting the objective function and constraints. The optimization problem is now formulated as:

$$\begin{aligned} \underset{\mathbf{w}}{\text{maximize}} \quad & \mathbf{w}^T\mu \\ \text{subject to} \quad & \mathbf{w}^T\Sigma\mathbf{w} \leq \sigma_p^2, \\ & \mathbf{w}^T 1 = 1, \end{aligned}$$

In this formulation, the objective is to maximize the expected return of the portfolio, subject to the constraint of not exceeding a specified level of risk (portfolio variance) and fully investing the available capital (the sum of the weights equals 1).

A common approach to solving the mean-variance optimization problem is to reformulate it as a Lagrangian form. In particular, we can introduce Lagrange multipliers λ_1 and λ_2 for the constraints, and construct the Lagrangian function $\mathcal{L}(\mathbf{w}, \lambda_1, \lambda_2)$:

$$\mathcal{L}(\mathbf{w}, \lambda_1, \lambda_2) = \mathbf{w}^T\Sigma\mathbf{w} - \lambda_1(\mathbf{w}^T\mu - \mu_0) - \lambda_2(\mathbf{w}^T 1 - 1).$$

To find the optimal portfolio weights $\mathbf{w}^*$, we need to minimize the Lagrangian function with respect to the portfolio weights $\mathbf{w}$ and solve the following system of first-order conditions:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 2\Sigma\mathbf{w} - \lambda_1\mu - \lambda_2 1 = 0.$$

From this equation, we can express the optimal portfolio weights as a linear combination of the expected return vector μ and the constant vector 1:

$$\mathbf{w}^* = \frac{1}{2}\lambda_1\Sigma^{-1}\mu + \frac{1}{2}\lambda_2\Sigma^{-1}1.$$

Now, we can substitute the optimal weights $\mathbf{w}^*$ back into the constraint equations:

$$\begin{aligned} \mathbf{w}^{*T}\mu &= \mu_0, \\ \mathbf{w}^{*T}1 &= 1. \end{aligned}$$

Solving this linear system for λ_1 and λ_2, we obtain the values for the Lagrange multipliers. Then, we can substitute these values back into the expression for $\mathbf{w}^*$ to find the optimal portfolio weights.

To proceed, we obtain the following equations after the substitution:

$$\begin{aligned}\mu_0 = \mathbf{w}^{*T}\mu &= \frac{1}{2}\lambda_1(\mu^T\Sigma^{-1}\mu) + \frac{1}{2}\lambda_2(\mu^T\Sigma^{-1}1),\\ 1 = \mathbf{w}^{*T}1 &= \frac{1}{2}\lambda_1(1^T\Sigma^{-1}\mu) + \frac{1}{2}\lambda_2(1^T\Sigma^{-1}1).\end{aligned}$$

We can write the above system of linear equations in matrix form:

$$\begin{bmatrix}\mu^T\Sigma^{-1}\mu & \mu^T\Sigma^{-1}1\\ 1^T\Sigma^{-1}\mu & 1^T\Sigma^{-1}1\end{bmatrix}\begin{bmatrix}\lambda_1\\ \lambda_2\end{bmatrix} = \begin{bmatrix}2\mu_0\\ 2\end{bmatrix}.$$

Let's denote the matrix on the left-hand side as A and the vectors on the right-hand side as b:

$$A = \begin{bmatrix}\mu^T\Sigma^{-1}\mu & \mu^T\Sigma^{-1}1\\ 1^T\Sigma^{-1}\mu & 1^T\Sigma^{-1}1\end{bmatrix}, \quad b = \begin{bmatrix}2\mu_0\\ 2\end{bmatrix}.$$

To find the values of λ_1 and λ_2, we need to solve the linear system $A\lambda = b$, where $\lambda = \begin{bmatrix}\lambda_1\\ \lambda_2\end{bmatrix}$. We can solve this system by calculating the inverse of matrix A and then multiplying it by vector b:

$$\lambda = A^{-1}b.$$

To facilitate the derivation, let's denote $A = \mu^T\Sigma^{-1}\mu$, $B = \mathbf{1}^T\Sigma^{-1}\mu = \mu^T\Sigma^{-1}\mathbf{1}$ (by symmetry of Σ^{-1}), and $C = \mathbf{1}^T\Sigma^{-1}\mathbf{1}$, where $\mathbf{1}$ is the vector of ones. These scalars represent the inverse-variance weighted sums of expected returns and unity, respectively, facilitating a simplified representation of the system. Substituting these definitions into the system yields:

$$\mu_0 = \frac{1}{2}\lambda_1 A + \frac{1}{2}\lambda_2 B,$$

$$1 = \frac{1}{2}\lambda_1 B + \frac{1}{2}\lambda_2 C.$$

To solve for λ_1 and λ_2, we find:

$$\lambda_1^* = \frac{-2B + 2C\mu_0}{AC - B^2},$$

$$\lambda_2^* = \frac{2A - 2B\mu_0}{AC - B^2}.$$

Once we have the optimal values λ_1^* and λ_2^*, we can replace them back into the expression for the optimal portfolio weights $\mathbf{w}^*$, giving:

$$\mathbf{w}^* = \frac{1}{2}\left(\frac{-2B + 2C\mu_0}{AC - B^2}\right)\Sigma^{-1}\mu + \frac{1}{2}\left(\frac{2A - 2B\mu_0}{AC - B^2}\right)\Sigma^{-1}\mathbf{1},$$

Simplifying the expression by distributing $\Sigma^{-1}\mu$ and $\Sigma^{-1}\mathbf{1}$ and combining terms gives us the closed-form solution for the optimal weights:

$$\mathbf{w}^* = \left(\frac{C\mu_0 - B}{AC - B^2}\right)\Sigma^{-1}\mu + \left(\frac{A - B\mu_0}{AC - B^2}\right)\Sigma^{-1}\mathbf{1},$$

This solution precisely characterizes the distribution of portfolio weights across the risky assets and the risk-free asset that minimizes the portfolio variance for a given target return μ_0, subject to the constraints that the expected portfolio return meets the target and the sum of the weights equals one.

Note that the matrix A is invertible if it is nonsingular, which means that its determinant is nonzero. Let us calculate the determinant of matrix A:

$$\det(A) = \left(\mu^T\Sigma^{-1}\mu\right)\left(1^T\Sigma^{-1}1\right) - \left(\mu^T\Sigma^{-1}1\right)^2.$$

The determinant can be simplified as:

$$\det(A) = \mu^T\Sigma^{-1}\mu \cdot 1^T\Sigma^{-1}1 - \left(\mu^T\Sigma^{-1}1\right)^2.$$

Thus, matrix A is invertible if $\det(A) \neq 0$. In practice, matrix A is usually invertible for typical financial data as long as there is sufficient variation in expected returns and covariances of the assets. If $\det(A) = 0$, it would indicate that there is a linear dependency between the expected returns and the constant vector 1, which is an unusual situation. In such a case, alternative optimization methods or regularization techniques might be needed to solve the portfolio optimization problem.

The Lagrangian formulation allows us to incorporate the trade-off between risk and return explicitly in the optimization problem. By adjusting the target expected return μ_0, we can find different optimal portfolios on the efficient frontier. A higher target return will generally result in a higher portfolio risk, while a lower target return will lead to a lower portfolio risk. This trade-off between risk and return is a key aspect of portfolio optimization in the context of Modern Portfolio Theory.

Now let us look at a brief overview of environment setup used in deep reinforcement learning.

0.1.4 More on the Portfolio Variance

The portfolio variance, denoted as $\mathbf{w}^T\Sigma\mathbf{w}$, is a positive scalar value that measures the magnitude of the portfolio's risk. A single portfolio variance requires three components: $\mathbf{w}^T$, Σ, and $\mathbf{w}$. Let us look at these components in more detail.

We start with $\mathbf{w}^T$, the transpose of the portfolio weights vector $\mathbf{w}$. $\mathbf{w}^T$ has dimensions $1 \times n$, where n is the number of assets in the portfolio. Now we come to Σ, the covariance matrix of the assets' returns, with dimensions $n \times n$. And finally, we have $\mathbf{w}$, the portfolio weights vector, with dimensions $n \times 1$.

The computation then follows via a series of matrix multiplications, starting with $\mathbf{w}^T\Sigma$. The step gives a $1 \times n$ row vector, where each element of this vector is the sum of the product of the corresponding row elements of $\mathbf{w}^T$ and the column elements of Σ. Next, we calculate $(\mathbf{w}^T\Sigma)\mathbf{w}$, which gives a scalar result computed as the sum of the product of the corresponding elements of the row vector $\mathbf{w}^T\Sigma$ and the column vector $\mathbf{w}$.

In more detail, the portfolio variance can be expressed as:

$$\mathbf{w}^T\Sigma\mathbf{w} = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \sigma_{ij},$$

where w_i and w_j are the weights of assets i and j, respectively, and σ_{ij} is the covariance between the returns of assets i and j. The portfolio variance is thus a scalar value that represents the portfolio's risk.

0.1.5 Mean Variance Optimization

The classical mean-variance optimization problem, as proposed by Markowitz, can be formulated as:

$$\begin{aligned} \underset{\mathbf{w}}{\text{minimize}} \quad & \mathbf{w}^T\Sigma\mathbf{w} \\ \text{subject to} \quad & \mathbf{w}^T\mu = \mu_0, \\ & \mathbf{w}^T 1 = 1, \end{aligned}$$

where μ_0 is the user-defined target expected return, and 1 is a column vector of ones. The optimization problem aims to find the portfolio weights $\mathbf{w}$ that minimize portfolio variance, subject to the constraints of achieving the target expected return and fully investing the available capital (i.e., the sum of the weights equals 1).

Note that we often model the first constraint as $\mathbf{w}^T\mu \geq \mu_0$, which requires that the expected portfolio return is no less than the target return.

We can also reformulate the mean-variance optimization problem as maximizing the expected return by adjusting the objective function and constraints. The optimization problem is now formulated as:

$$\begin{aligned} \underset{\mathbf{w}}{\text{maximize}} \quad & \mathbf{w}^T\mu \\ \text{subject to} \quad & \mathbf{w}^T\Sigma\mathbf{w} \leq \sigma_p^2, \\ & \mathbf{w}^T 1 = 1, \end{aligned}$$

In this formulation, the objective is to maximize the expected return of the portfolio, subject to the constraint of not exceeding a specified level of risk (portfolio variance) and fully investing the available capital (the sum of the weights equals 1).

A common approach to solving the mean-variance optimization problem is to reformulate it as a Lagrangian form. In particular, we can introduce Lagrange multipliers λ_1 and λ_2 for the constraints, and construct the Lagrangian function $\mathcal{L}(\mathbf{w}, \lambda_1, \lambda_2)$:

$$\mathcal{L}(\mathbf{w}, \lambda_1, \lambda_2) = \mathbf{w}^T\Sigma\mathbf{w} - \lambda_1(\mathbf{w}^T\mu - \mu_0) - \lambda_2(\mathbf{w}^T 1 - 1).$$

To find the optimal portfolio weights $\mathbf{w}^*$, we need to minimize the Lagrangian function with respect to the portfolio weights $\mathbf{w}$ and solve the following system of first-order conditions:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 2\Sigma\mathbf{w} - \lambda_1\mu - \lambda_2 1 = 0.$$

From this equation, we can express the optimal portfolio weights as a linear combination of the expected return vector μ and the constant vector 1:

$$\mathbf{w}^* = \frac{1}{2}\lambda_1\Sigma^{-1}\mu + \frac{1}{2}\lambda_2\Sigma^{-1}1,$$

Now, we can substitute the optimal weights $\mathbf{w}^*$ back into the constraint equations:

$$\mathbf{w}^{*T}\mu = \mu_0,$$

$$\mathbf{w}^{*T}1 = 1.$$

Solving this linear system for λ_1 and λ_2, we obtain the values for the Lagrange multipliers. Then, we can substitute these values back into the expression for $\mathbf{w}^*$ to find the optimal portfolio weights.

To proceed, we obtain the following equations after the substitution:

$$\mu_0 = \mathbf{w}^{*T}\mu = \frac{1}{2}\lambda_1(\mu^T\Sigma^{-1}\mu) + \frac{1}{2}\lambda_2(\mu^T\Sigma^{-1}1),$$

$$1 = \mathbf{w}^{*T}1 = \frac{1}{2}\lambda_1(1^T\Sigma^{-1}\mu) + \frac{1}{2}\lambda_2(1^T\Sigma^{-1}1).$$

We can write the above system of linear equations in matrix form:

$$\begin{bmatrix} \mu^T\Sigma^{-1}\mu & \mu^T\Sigma^{-1}1 \\ 1^T\Sigma^{-1}\mu & 1^T\Sigma^{-1}1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2\mu_0 \\ 2 \end{bmatrix}.$$

Let's denote the matrix on the left-hand side as A and the vectors on the right-hand side as b:

$$A = \begin{bmatrix} \mu^T\Sigma^{-1}\mu & \mu^T\Sigma^{-1}1 \\ 1^T\Sigma^{-1}\mu & 1^T\Sigma^{-1}1 \end{bmatrix}, \quad b = \begin{bmatrix} 2\mu_0 \\ 2 \end{bmatrix}.$$

To find the values of λ_1 and λ_2, we need to solve the linear system $A\lambda = b$, where $\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}$. We can solve this system by calculating the inverse of matrix A and then multiplying it by vector b:

$$\lambda = A^{-1}b.$$

To facilitate the derivation, let's denote $A = \mu^T\Sigma^{-1}\mu$, $B = \mathbf{1}^T\Sigma^{-1}\mu = \mu^T\Sigma^{-1}\mathbf{1}$ (by symmetry of Σ^{-1}), and $C = \mathbf{1}^T\Sigma^{-1}\mathbf{1}$, where $\mathbf{1}$ is the vector of ones. These scalars represent the inverse-variance weighted sums of expected returns and unity, respectively, facilitating a simplified representation of the system. Substituting these definitions into the system yields:

$$\mu_0 = \frac{1}{2}\lambda_1 A + \frac{1}{2}\lambda_2 B,$$

$$1 = \frac{1}{2}\lambda_1 B + \frac{1}{2}\lambda_2 C.$$

To solve for λ_1 and λ_2, we find:

$$\lambda_1^* = \frac{-2B + 2C\mu_0}{AC - B^2},$$

$$\lambda_2^* = \frac{2A - 2B\mu_0}{AC - B^2}.$$

Once we have the optimal values λ_1^* and λ_2^*, we can replace them back into the expression for the optimal portfolio weights $\mathbf{w}^*$, giving:

$$\mathbf{w}^* = \frac{1}{2}\left(\frac{-2B + 2C\mu_0}{AC - B^2}\right)\Sigma^{-1}\mu + \frac{1}{2}\left(\frac{2A - 2B\mu_0}{AC - B^2}\right)\Sigma^{-1}\mathbf{1},$$

Simplifying the expression by distributing $\Sigma^{-1}\mu$ and $\Sigma^{-1}\mathbf{1}$ and combining terms gives us the closed-form solution for the optimal weights:

$$\mathbf{w}^* = \left(\frac{C\mu_0 - B}{AC - B^2}\right)\Sigma^{-1}\mu + \left(\frac{A - B\mu_0}{AC - B^2}\right)\Sigma^{-1}\mathbf{1},$$

This solution precisely characterizes the distribution of portfolio weights across the risky assets and the risk-free asset that minimizes the portfolio variance for a given target return μ_0, subject to the constraints that the expected portfolio return meets the target and the sum of the weights equals one.

Note that the matrix A is invertible if it is nonsingular, which means that its determinant is nonzero. Let us calculate the determinant of matrix A:

$$\det(A) = \left(\mu^T\Sigma^{-1}\mu\right)\left(1^T\Sigma^{-1}1\right) - \left(\mu^T\Sigma^{-1}1\right)^2.$$

The determinant can be simplified as:

$$\det(A) = \mu^T\Sigma^{-1}\mu \cdot 1^T\Sigma^{-1}1 - \left(\mu^T\Sigma^{-1}1\right)^2.$$

Thus, matrix A is invertible if $\det(A) \neq 0$. In practice, matrix A is usually invertible for typical financial data as long as there is sufficient variation in expected returns and covariances of the assets. If $\det(A) = 0$, it would indicate that there is a linear dependency between the expected returns and the constant vector 1, which is an unusual situation. In such a case, alternative optimization methods or regularization techniques might be needed to solve the portfolio optimization problem.

The Lagrangian formulation allows us to incorporate the trade-off between risk and return explicitly in the optimization problem. By adjusting the target expected return μ_0, we can find different optimal portfolios on the efficient frontier. A higher target return will generally result in a higher portfolio risk, while a lower target return will lead to a lower portfolio risk. This trade-off between risk and return is a key aspect of portfolio optimization in the context of Modern Portfolio Theory.

Now let us look at a brief overview of environment setup used in deep reinforcement learning.

0.2 Introducing Deep Reinforcement Learning

Deep reinforcement learning (DRL) is an advanced branch of machine learning that combines deep learning and reinforcement learning techniques. It leverages the power of deep neural networks to approximate complex functions, such as value functions or policy mappings, in high-dimensional or even continuous state spaces. DRL has demonstrated impressive results in various fields, such as playing games, robotics, and natural language processing. One of the key advantages of DRL is its ability to learn complex decision-making strategies directly from raw data provided by the learning environment, making it well-suited for tackling challenging problems such as portfolio optimization in finance.

As discussed in the previous section, portfolio optimization aims to select the best combination of assets to maximize expected return while minimizing risk. The optimal portfolio weights, as shown earlier based on the Modern Portfolio Theory (MPT), relies on historical data to estimate future expected returns (μ) and risk (Σ). However, financial markets are complex, dynamic, and influenced by numerous factors that make it difficult to model and predict. This is where deep reinforcement learning can offer a significant advantage.

In portfolio optimization, a DRL agent learns to make decisions on asset allocation based on the current market state, aiming to maximize the long-term cumulative returns over a given investment horizon. The agent interacts with a simulated or real trading environment. Each action receives a feedback in the form of rewards or penalties. By automatically exploring different trading strategies and learning from its experiences (past state, action, and reward), the DRL agent can adapt to market changes and discover effective allocation strategies.

Using DRL for portfolio optimization offers several benefits:

- **Model-free learning**: Traditional methods rely on specific assumptions about asset returns or risk, following the sequence of predict-then-optimize. However, DRL can learn directly from raw market data and make trading decisions, making it more robust to changes in market dynamics and self-contained in an end-to-end framework. In other words, the explicit prediction step on expected return and risk is ignored, and the agent directly learns the optimal action on portfolio allocation strategy given specific market conditions.

- **Adaptive strategies**: DRL agents can continually learn and update their strategies as new data becomes available, allowing them to adapt to evolving market conditions. Having a self-adaptive model also saves the need to re-assess and re-train an existing model developed based on historical data.
- **Complex decision-making**: DRL can capture non-linear relationships and intricate dependencies among assets, which can lead to better-informed and more effective portfolio allocation decisions. When coupled with deep neural networks, DRL can be powerful function approximators to learn the intricacies in the underlying structure of the portfolio risk and return.
- **Scalability**: DRL can handle large-scale problems with numerous assets and complex market structures, making it suitable for real-world portfolio optimization tasks that involve portfolios with many underlying assets.

In conclusion, deep reinforcement learning provides an automated and end-to-end solution to portfolio optimization by enabling more adaptive, data-driven decision-making. Its ability to learn complex strategies from raw market data and adapt to changing market conditions makes it a promising approach for managing investments and maximizing returns in the ever-evolving financial landscape.

The next section introduces the Markov decision process used to characterize the learning environment.

0.2.1 Introducing the Markov decision process

A Markov Decision Process (MDP) is commonly used to model the environment in reinforcement learning. An MDP is a mathematical framework that provides a formal description of an environment for RL. It manifests as a tuple (S, A, P, R, γ) where:

- S is the state space, which is the set of all possible states that the agent can be in. In the case of portfolio optimization, a state could be the historical asset prices up to a certain time frame.
- A is the action space, which is the set of all possible actions the agent can take. In the case of portfolio optimization, an action could be the vector of weights to be assigned to each asset at the current time step. An action also represents a sampling choice, where the agent attempts to learn from the black-box function (the environment) by sequential sampling.

- P is the state transition probability function. $P(s'|s,a)$ gives the probability of transitioning to state s' if action a is taken in state s. This function defines the dynamics of the environment and often remains unknown to the agent. In some environments, the transition probabilities might be deterministic, meaning that a given state-action pair always leads to the same next state. In other environments, the transitions could be stochastic, with some element of randomness involved in the form of a probability distribution over a set of possible states.
- R is the reward function. $R(s,a,s')$ gives the immediate reward received after transitioning to state s' by taking action a in state s. The reward function quantifies the immediate payoff that the agent receives after performing an action in a given state and transitioning to a new state. In portfolio management, this could be the financial return from a particular trading strategy.
- γ is the discount factor, a number between 0 and 1 that determines the present value of future rewards. A reward received k time steps in the future is worth γ^{k-1} times what it would be worth if it were received immediately. A discount factor close to 0 means that the agent is myopic and primarily focused on immediate rewards, while a discount factor close to 1 means that the agent has a long-term focus and values future rewards highly.

The Markov property is the assumption that the future state and reward depend only on the current state and action, and not on the history of past states and actions. The Markov property states that the probability of transitioning to any particular state depends solely on the current state and the decision made, not on the sequence of events that preceded it. Mathematically, this is expressed as $P(S_{t+1}=s'|S_t=s,A_t=a)=P(S_{t+1}=s'|S_1,A_1,...,S_t=s,A_t=a)$. This property simplifies the learning process by reducing the amount of information the agent needs to consider.

Of course, that state s needs to be comprehensive enough to support the memoryless property, including the status of every relevant event in time. However, in practice, we would only store necessary information (such as asset prices for the past 30 days) as an approximation due to limits in computer storage.

In an MDP, the goal of the RL agent is to learn a policy. A policy $\pi(a|s)$ is a function that specifies the probability of taking each action a in each state s. In the deterministic case, a policy maps each state to a single action. In the stochastic case, a policy gives a probability distribution over actions for each state. The agent seeks to find an optimal policy that maximizes the expected cumulative discounted reward over time, starting from any state. An optimal policy is the one that gives the best action to take in each state to achieve this goal of maximizing the cumulative return.

Another important component in an MDP is the value function. The state-value function $V^\pi(s)$ and action-value function $Q^\pi(s,a)$ for a policy π are real-valued functions that measure the expected cumulative discounted reward from each state or state-action pair. They satisfy the Bellman equations, which are recursive relationships involving the transition probabilities and rewards. More on this later.

The next section focuses on setting up the environment for training a DRL agent for portfolio optimization.

0.2.2 Setting up the learning environment

To set up the learning environment for portfolio optimization using deep reinforcement learning (DRL), we need to define the necessary components of a reinforcement learning problem, including the state space, action space, reward function, and transition dynamics. We elaborate on these four components in the following.

- **State Space ($\mathcal{S}$)**: The state space is a representation of the information available to the DRL agent at each time step t. In the context of portfolio optimization, a state $s_t \in \mathcal{S}$ typically includes historical price data for a given set of N assets. We can also add potentially useful features such as technical indicators and other relevant market features. The state may also include the current portfolio holdings or weights, as well as any additional constraints or transaction costs. Mathematically, a state s_t at time step t can be represented as a vector or matrix containing the relevant information:

$$s_t = \begin{bmatrix} P_t \\ I_t \\ \mathbf{w}_{t-1} \end{bmatrix},$$

  where P_t represents the price data, I_t represents any additional market indicators, and $\mathbf{w}_{t-1}$ represents the portfolio weights at the previous time step $t-1$.

- **Action Space ($\mathcal{A}$)**: The action space consists of all possible decisions the DRL agent can make at each time step. Depending on the specific problem domain, the action can be discrete or continuous. In portfolio optimization, an action $a_t \in \mathcal{A}$ represents the allocation of the portfolio among the N assets, taking into account any constraints or transaction costs. When an action is continuous, the agent selects a numeric weight for each asset and outputs a weight vector $\mathbf{w}_t \in \mathbb{R}^N$. When an action is discrete, it selects from a predefined set of allocation strategies. In both cases, the weights must satisfy the constraints, such as fully investing the available capital:

$$a_t = \mathbf{w}_t = \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ \vdots \\ w_{N,t} \end{bmatrix}, \quad \text{subject to} \quad \sum_{i=1}^{N} w_{i,t} = 1.$$

- **Reward Function ($r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$):** The reward function quantifies the immediate (short-term) benefit or penalty the DRL agent receives for taking a specific action in a given state. In portfolio optimization, the reward function can be designed to capture various objectives, such as maximizing returns, minimizing risk, or balancing the trade-off between risk and return. One common choice for the reward function is the portfolio return or the logarithmic rate of return:

$$r(s_t, a_t) = \log \left(\sum_{i=1}^{N} w_{i,t} \frac{P_{i,t}}{P_{i,t-1}} \right).$$

- **Transition Dynamics ($p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$):** The transition dynamics describe the probability of transitioning from one state to another given a specific action. It mostly remains unknown to the learning agent. In the context of financial markets, the transition dynamics are often stochastic and influenced by multiple factors. While it is generally challenging to model these dynamics explicitly, DRL agents can learn to make decisions based on the observed transitions from historical or simulated data. Mathematically, the transition dynamics are defined as:

$$p(s_{t+1}|s_t, a_t) = \mathbb{P}(S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t),$$

where $\mathbb{P}(\cdot)$ denotes the probability of transitioning to state s_{t+1} given the current state s_t and action a_t. In practice, DRL algorithms often learn implicitly from the observed transitions without explicitly modeling the transition dynamics, although one can choose to do so. See more discussion on model-based versus model-free learning algorithms in a later chapter.

With these components defined, we can set up the learning environment for portfolio optimization using DRL. The goal of the DRL agent is to learn an optimal policy $\pi^* : \mathcal{S} \to \mathcal{A}$, which maps states to actions, that maximizes the expected cumulative reward over a given investment horizon T. Formally, the objective can be expressed as:

$$\pi^* = \arg\max_{\pi} \mathbb{E} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, \pi(s_t)) \middle| s_0 \right],$$

where $\gamma \in [0, 1]$ is a discount factor that balances the importance of immediate (note the first reward is not discounted) and future (discounted) rewards. The DRL agent iteratively updates its policy based on its experiences, either by learning a value function $V^\pi : \mathcal{S} \to \mathbb{R}$, a state-action value function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, or by directly learning the policy mapping π.

Popular DRL algorithms that can be applied to portfolio optimization include Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Deep Deterministic Policy

Gradient (DDPG), among others. These algorithms can be adapted to handle the unique challenges of portfolio optimization, such as continuous action spaces, transaction costs, and risk constraints. No matter which learning algorithm we use, the learning environment is often assumed to stay the same without being subject to potential impact from the action taken. An example is trading large quantities of market orders, where one tries to avoid causing any potential market impact by engaging the dark pool (majority of the orders are traded anonymously) or following the crowd (start and end of a trading day when the trading volume is high).

In summary, setting up a simulated learning environment for portfolio optimization using DRL involves defining the state space, action space, reward function, and transition dynamics. The DRL agent's goal is to learn an optimal policy that maximizes the expected cumulative reward over the investment horizon, adapting to the complex and dynamic nature of financial markets as price data evolve.

Before we dive into the learning environment for DRL, let us spend a moment and better understand why the log return is a commonly used metric instead of the raw return itself.

0.2.3 More on log return

Recall that the return of an asset (at daily frequency) is defined as the daily percentage change in the asset price. This return, denoted as r_t for time step t, is also referred to as the simple return. As it turns out, a more mathematically convenient form of return is the logarithmic return, or log return for short.

The log return, denoted as $r_{log,t}$ for time step t and also known as the continuously compounded return, is the natural logarithm of the ratio of the asset prices at at two consecutive time points t and $t-1$. It is defined as follows:

$$r_{log,t} = \log\left(\frac{p_t}{p_{t-1}}\right) = \log(p_t) - \log(p_{t-1})$$

where p_t is the asset price at time t, and p_{t-1} is the asset price at time $t-1$.

Now we look at the relationship between the simple return (r_t) and the logarithmic return ($r_{log,t}$). The relationship can be approximated using the Taylor series expansion of the natural logarithm function for small values of r_t:

$$\log(1 + r_t) \approx r_t - \frac{1}{2}r_t^2 + ...$$

When the simple return r_t is small, higher-order terms become negligible, and we can approximate the logarithmic return as:

$$r_{log,t} = \log(1 + r_t) \approx r_t$$

This approximation holds well for small values of r_t, which are common in daily asset price changes. However, it's important to note that the approximation becomes less accurate for larger values of r_t. In general, the relationship between simple return and logarithmic return is nonlinear, and the approximation should be used with caution when dealing with large returns or long time horizons.

When it comes to the practice usage, the logarithmic returns are often preferred over simple returns in finance and portfolio optimization for several reasons. These reasons, as illustrated in the following, are mainly due to the mathematical properties of logarithmic returns that make them more suitable for modeling financial assets and optimizing portfolios.

First, logarithmic returns are additive, making them more suitable for modeling the compounding effect of financial assets over time. With simple returns, we would need to multiply the returns sequentially in order to calculate the terminal return. However, with logarithmic returns, we can simply add the periodic returns, as shown in the following. This additive property simplifies calculations and allows for more accurate representation of returns over time. By maximizing the cumulative logarithmic returns, the DRL agent learns to consider the compounding effect of both positive and negative returns, which inherently includes the impact of risk as well.

$$r_{log,T} = \sum_{t=1}^{T} r_{log,t} = \sum_{t=1}^{T} \log(1 + r_i)$$

Second, logarithmic returns have a more symmetrical distribution around zero compared to simple returns. This property allows the agent to treat positive and negative returns more symmetrically, which helps the agent to better understand the risk-return tradeoff. To see such symmetry, assume the price of an asset changes from p_t at time t to p_{t+1} at time $t + 1$, and then back to p_t at time $t + 2$. The corresponding log returns are $r_{log,t+1} = \log\left(\frac{p_{t+1}}{p_t}\right)$ and $r_{log,t+2} = \log\left(\frac{p_t}{p_{t+1}}\right)$, respectively. These two quantities are symmetric around zero since $\log\left(\frac{p_{t+1}}{p_t}\right) = -\log\left(\frac{p_t}{p_{t+1}}\right)$.

In addition, when maximizing the cumulative logarithmic return, the agent is inherently risk-averse. This is because of the concave nature of the logarithm function. Specifically, the expected logarithmic return is always less than or equal to the logarithm of the expected return due to Jensen's inequality:

$$E[\log(1 + r)] \leq \log(1 + E[r])$$

In other words, the investment with lower volatility will have a higher expected logarithmic return, as the natural logarithm function is more sensitive to negative returns than positive returns of the same magnitude.

The next section illustrates how to implement this learning environment for portfolio optimization using DRL.

0.2.4 Implementing the learning environment

In this implementation, we introduce a custom `PortfolioOptimizationEnv` class, which extends the OpenAI Gym environment to create a reinforcement learning environment specifically designed for portfolio optimization. The class is tailored for stock market data and focuses on optimizing the allocation of assets in a portfolio in order to achieve maximum returns while implicitly considering the risk using log returns. The environment utilizes historical stock data obtained through the `yfinance` library, which provides access to Yahoo Finance data.

The `PortfolioOptimizationEnv` class takes a list of stock tickers, a window size, a start date, an end date, an initial balance, and an optional random seed as inputs. The environment downloads the historical adjusted closing prices of the specified stocks within the given date range. The action and observation spaces are defined using Gym's 'Box' space, where the action space represents the portfolio weights for each stock, and the observation space represents the historical stock prices within the given window. Here, we limit the observable state to the historical stock prices up to the specified window size.

When resetting the environment, the `reset()` method initializes the balance and current step, and returns the initial observation. In the `step()` method, the provided action (portfolio weights) is first normalized to ensure they sum to one. The method then calculates the asset returns, updates the portfolio balance based on the weighted asset returns, and computes the reward as the logarithmic return of the portfolio. Finally, the method updates the current step, checks the `step()` method controls how the environment responds to a specific action from the agent. Such control comes in the form of feedback consisting of the corresponding reward and the next state.

This custom environment enables a reinforcement learning agent to learn an optimal asset allocation strategy over time by optimizing the cumulative rewards, which are represented by the logarithmic returns of the portfolio. The environment can be used in conjunction with various deep reinforcement learning algorithms to train the agent and explore different portfolio optimization strategies.

The following code listing provides the definition of the `PortfolioOptimizationEnv` class as the learning environment for the DRL agent.

```python
import gym
import numpy as np
import pandas as pd
import yfinance as yf
import torch

class PortfolioOptimizationEnv(gym.Env):

    def __init__(self, tickers, window_size, start_date, end_date,
                 initial_balance, seed=None):
        super().__init__()

        # Initialize the environment's properties
        self.tickers = tickers
        self.window_size = window_size
        self.initial_balance = initial_balance
        # Download historical stock data
        self.data = self.get_data(tickers, start_date, end_date)
        # Define the action and observation spaces
        self.action_space = gym.spaces.Box(low=0, high=1, shape=(len(tickers),))
        self.observation_space = gym.spaces.Box(low=0, high=np.inf, shape=(window_size, len(
        # Set the random seed for reproducibility
        if seed is not None:
            np.random.seed(seed)
            self.action_space.seed(seed)

    def get_data(self, tickers, start_date, end_date):
        # Download historical stock data using yfinance
        data = yf.download(tickers, start=start_date, end=end_date)['Adj Close']
        return data.dropna()  # drop rows with nan values

    def reset(self):
        # Reset the environment to its initial state
        self.balance = self.initial_balance
        self.current_step = self.window_size

        # Return the initial observation (historical stock prices)
        obs = self.data.iloc[self.current_step - self.window_size:self.current_step].values

        return obs.reshape(self.observation_space.shape)

    def step(self, action):
        # Normalize the action values (portfolio weights) so they sum to one
        action = action / (np.sum(action) + 1e-8)  # where 1e-8 is a small constant
        # Calculate the previous balance and asset returns
        prev_balance = self.balance
        asset_prices = self.data.iloc[self.current_step].values
        asset_returns = asset_prices / self.data.iloc[self.current_step - 1].values - 1

        # Update the portfolio balance based on the weighted asset returns
        self.balance = self.balance * (1 + np.sum(asset_returns * action))
        # Calculate the reward as the logarithmic return of the portfolio
        reward = np.log(self.balance / prev_balance)
        # Update the current step and check if the episode is done
        self.current_step += 1
        done = self.current_step == len(self.data) - 1
        # Generate the next observation (historical stock prices)
        obs = self.data.iloc[self.current_step - self.window_size:self.current_step].values
        # Include the current balance in the info dictionary
        info = {'balance': self.balance}

        return obs.reshape(self.observation_space.shape), reward, done, info
```

Now let us test out this class. In the following code listing, we select three stock tickers (Apple, Google, and Microsoft), a window size of 30 days, a date range of 2022, and an initial balance of 1,000. After resetting the environment, we randomly sample a vector of portfolio weights from the action space and pass it to the `step()` function to obtain the corresponding reward (portfolio return) and next state. The result shows a 1.3% daily return on the portfolio.

```
tickers = ['AAPL', 'GOOG', 'MSFT']
window_size = 30
start_date = '2022-01-01'
end_date = '2023-01-01'
initial_balance = 1000
seed = 8

env = PortfolioOptimizationEnv(tickers, window_size, start_date, end_date,initial_balance, s

# Get the initial state
state = env.reset()

# Sample and execute a random action
action = env.action_space.sample()
next_state, reward, done, _ = env.step(action)
```

```
# print(f"State: {state}")
print(f"Action: {action}")
# print(f"Next state: {next_state}")
print(f"Reward: {reward}")
print(f"Done: {done}")

###### output #####
# Action: [0.32697228 0.98727685 0.31871083]
# Reward: 0.01319677113043728
# Done: False
```

Note that the action (portfolio weights) only sums to one after being normalized in the `step()` function.

We can also examine the new balance after taking the action.

```
print(env.balance)

##### output #####
# 1013.2842332072671
```

The role of the DRL agent is to learn from the environment and make smarter actions so as to maximize the cumulative log return. The next section shifts the focus to the DRL agent.

0.2.5 Understanding the Decisioning Mechanism of A DRL Agent

The sequential decision-making process starts with the DRL agent making an action based on the initial state, receiving a reward and the next state, and then making a follow-up decision. Such iterative interactions between the agent and the environment is characterized in Figure 1. In the current iteration, the agent makes an action a_t based on the current state s_t and reward r_t, both obtained as a feedback from the environment based on the previous action a_t. Making an action amounts to determining which location to sample in the total action space, and each sampling decision made during the decision making process relies on a policy trained using historically collected observations. These observations arise from the iterative interactions between the agent and the environment, and assume the form of transition tuples that include observed states and rewards with the accompanying actions. Finally, the environment provides the next state s_{t+1} and reward r_{t+1} to the agent as the input to the next iteration, where the specific state transition and reward generation mechanism often remains unknown to the agent.

As the environment is often out of our control, the best hope is to learn a smart agent that makes the best choice at each step. Here, each step faces multiple, or infinitely many, choices

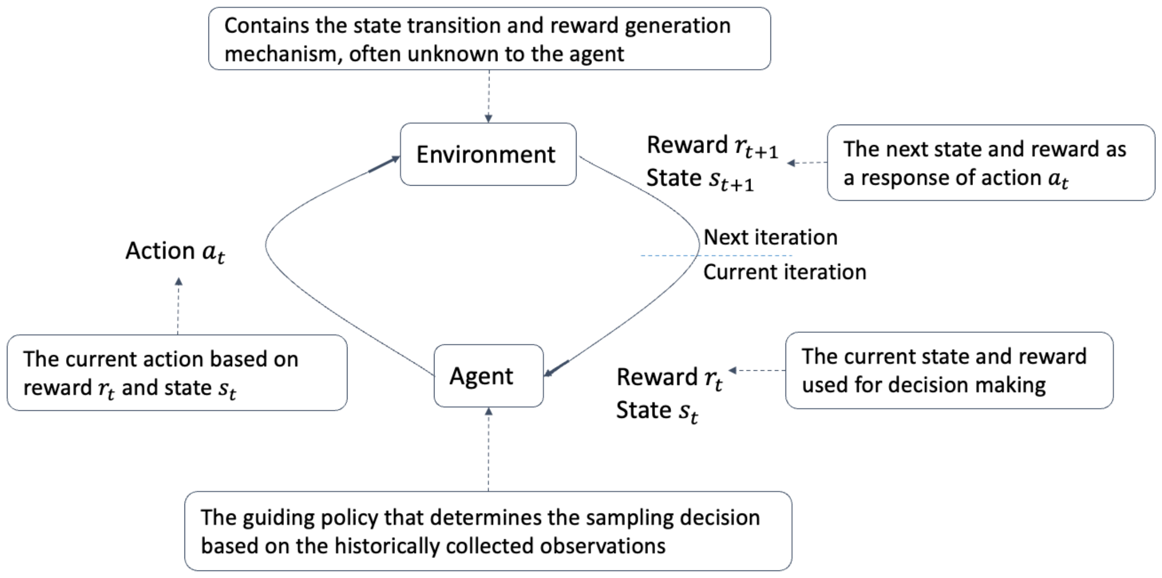
Contains the state transition and reward generation mechanism, often unknown to the agent
Environment
Reward r_{t+1}
State s_{t+1}
The next state and reward as a response of action a_t
Action a_t
Next iteration
Current iteration
The current action based on reward r_t and state s_t
Agent
Reward r_t
State s_t
The current state and reward used for decision making
The guiding policy that determines the sampling decision based on the historically collected observations

Figure 1: Iterative interaction between the agent and the environment.

of actions. We need a way to quantify the utility of each action for each state in order to make the best decision. Such quantification is provided by the value function. The value function in reinforcement learning describes the expected future rewards for an agent starting from a given state (or state-action pair), under a particular policy. It provides a quantitative measure of how good it is for the agent to be in a given state (or to take a particular action from a state).

There are two types of value functions: the state-value function and the action-value function.

The state-value function for a policy π, denoted $V^\pi(s)$, is the expected return from state s when actions are chosen according to π. Mathematically:

$$V^\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

where G_t is the long-term return at time t, which is the sum of future discounted rewards, and the expectation is taken over all possible action sequences following policy π and starting from S_t. G_t, as defined in the following, represents the long-term return instead of short-term reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots$$

This can be rewritten in a recursive form as:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

where:

- R_{t+1}, R_{t+2}, ... are the immediate rewards that the agent receives at time steps $t+1$, $t+2$, ..., respectively.
- γ is a discount factor with $0 \leq \gamma \leq 1$, which determines the present value of future rewards. Specifically, a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma < 1$, the sum in the definition of G_t has a finite limit, which makes the problem more manageable mathematically and computationally.
- G_{t+1}, G_{t+2}, ... are the (long-term) returns at time steps $t+1$, $t+2$, ..., respectively.

The return G_t is a crucial concept in reinforcement learning because it quantifies the total future reward that an agent can expect to receive, taking into account not only the immediate reward R_{t+1} but also the discounted future rewards. The goal in reinforcement learning is thus to learn a policy that maximizes the expected long-term return from each state.

On the other hand, the action-value function for a policy π, denoted as $Q^\pi(s, a)$, is the expected return from taking action a in state s and thereafter following π. Mathematically:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

It turns out we can re-express these two value functions using the Bellman equation, where the current value can be expressed in terms of future values. Let us look at how this is done.

0.2.6 Deriving the Bellman equations for state and action value functions

The **Bellman equation**, named after Richard Bellman, is a fundamental concept in the field of dynamic programming and reinforcement learning. The equation describes an approach to problem-solving where the problem is broken down into overlapping sub-problems, and the solution to the larger problem is expressed in terms of the solutions to the sub-problems. In the context of reinforcement learning, the Bellman equation describes the relationship between the value of a state (or state-action pair) and the values of its successor states (or state-action pairs).

The Bellman equation provides a recursive relationship between the value of a state (or state-action pair) and the values of its successor states (or state-action pairs). This recursive relationship is based on the principle of optimality, which states that if a policy is optimal, then it must be the case that whatever the initial state and action are, the remaining actions must constitute an optimal policy with regard to the state resulting from the first action. In other words, for a state (or action) to be optimal, all the follow-up states (or actions) need to be optimal as well. The principle of optimality then gives a recursive definition to the state-value and action-value functions.

Let's start with the definition of the state-value function $V^\pi(s)$ under a policy π. It's the expected return G_t given that the agent is in state s at time t:

$$V^\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Recall the definition of the return G_t is the sum of discounted future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots = R_{t+1} + \gamma G_{t+1}$$

Substituting G_t into the definition of $V^\pi(s)$ gives:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s]$$

By linearity of expectation, this can be broken down into:

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} | S_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} | S_t = s]$$

The first term represents the expected immediate reward after state s. The second term represents the expected discounted return from the next state onward. Noting that G_{t+1} is essentially the state-value of the next state S_{t+1}, and expanding the expectations using the law of total expectation, we get:

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r | s, a)[r + \gamma V^\pi(s')]$$

where $\pi(a|s)$ is the probability of taking action a under policy π in state s, $p(s', r|s, a)$ is the probability of transitioning to state s' and receiving reward r when taking action a in state s, and γ is the discount factor. This is the Bellman equation for the state-value function.

The derivation of the Bellman equation for the action-value function $Q^\pi(s, a)$ is similar. Starting from the definition of $Q^\pi(s, a)$:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Substituting G_t and breaking down the expectation gives:

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} | S_t = s, A_t = a] + \gamma \mathbb{E}_\pi[G_{t+1} | S_t = s, A_t = a]
\end{aligned}
$$

Now, remember that G_{t+1} is the return from the next time step onward, which is the expected value of taking action A_{t+1} under policy π in the next state S_{t+1}, so $G_{t+1} = \mathbb{E}_\pi[Q^\pi(S_{t+1}, A_{t+1})]$. Substituting this and expanding the expectations, we get:

$$
Q^\pi(s,a) = \sum_{s',r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s', a') \right]
$$

This is the Bellman equation for the action-value function.

In both cases, the Bellman equation expresses the value of a state (or state-action pair) in terms of the expected immediate reward and the expected value of the next state (or next state-action pair), thus providing a basis for iterative algorithms to estimate the value functions.

The Bellman equation is a fundamental concept in reinforcement learning. It forms the basis for many RL algorithms, such as value iteration, policy iteration, and Q-learning, which seek to find the optimal policy by iteratively updating the value function estimates until they satisfy the Bellman equation (i.e., until they reach a fixed point of the Bellman update operation). In deep reinforcement learning, the Bellman equation is used to define the loss function for training the neural networks that represent the value functions.

Let us look at implementing a working DRL agent based on the previous environment setup.

0.2.7 Training an DRL agent

There are a wide range of algorithms that can be used to train a DRL agent. The choice of algorithm depends on many factors, such as the complexity of the task, the size of the state and action spaces, whether the action space is discrete or continuous, and the amount of computational resources available. Common algorithms are based on either value iterations, policy gradients, proximal policy optimization (PPO), or a combination of both value and policy based algorithms such as the actor-critic architecture. To bypass these technical details and better focus on the bigger picture from the endgame perspective, we resort to off-the-shelf packages that provide self-contained implementations of these algorithms, and delay a more technical algorithmic introduction to later chapters.

For ease of prototyping, we use the Stable Baselines3 package, which is a high-level RL library based on PyTorch and provides implementations of many state-of-the-art RL algorithms.

In the following code listing, we first initialize the environment by configuring a few global input parameters and instantiating the custom environment class defined earlier. In this case, the agent will be interacting with a portfolio containing three stocks (‘AAPL’, ‘GOOG’, ‘TSLA’) over a specific period (‘2010-01-01’ to ‘2022-12-31’). The window size is set to 20, which means the agent can see the past 20 days of price information. The initial balance of the portfolio is set to $10,000. Besides, the environment is wrapped with `DummyVecEnv` to make it vectorized because Stable Baselines3 algorithms require a vectorized environment to run.

```
# Initialize the environment
tickers = ['AAPL', 'GOOG', 'TSLA']
window_size = 20
start_date = '2010-01-01'
end_date = '2022-12-31'
initial_balance = 10000

env = PortfolioOptimizationEnv(tickers, window_size, start_date, end_date, initial_balance)
env = DummyVecEnv([lambda: env])  # The algorithms require a vectorized environment to run
```

Next, we initialize the agent using the PPO algorithm and choose the `MlpPolicy` as the policy model for the agent, which is a type of neural network suitable for environments with continuous state spaces. We then train the agent for a specified number of time steps (in this case, 10,000 time steps) using the `learn` method. Finally, the model is saved using the ‘save‘ method, so it can be loaded and used later without the need to retrain the model. See the following code listing. Also, do enable GPU acceleration if it is available.

```
# Initialize the agent
model = PPO('MlpPolicy', env, verbose=1)

# Train the agent
model.learn(total_timesteps=10000)

# Save the agent
model.save('ppo_portfolio_optimization')
```

Let us examine the performance of the DRL agent via the wealth curve of the portfolio managed by the agent. The wealth curve illustrates how the portfolio’s balance changes over time as the agent interacts with the environment. Specifically, we would like to trace the evolution of the daily balance for the same historical period used to train the agent.

In the following code listing, we define the `plot_wealth_curve` function to take a trained model and an environment as input and plot the wealth curve of the portfolio managed by the agent. It starts by resetting the environment and initializing some variables: `obs` (observation), `done` (whether the episode is done), and `balances` (to store the portfolio balances at each step).

Next, the function enters a loop to use the model and predict an action based on the current observation at each step. This action is returned to the environment, which then reports a new observation, a reward, a done flag, and an info dictionary. The portfolio balance at the current step, obtained from the `info` dictionary, is then appended to the `balances` list. This loop continues until the episode is done, i.e., until done is `True` for all environments in the vectorized environment.

Finally, the function plots the portfolio balances over time to create the wealth curve. The x-axis represents the step (time), and the y-axis represents the portfolio balance.

```python
import matplotlib.pyplot as plt

def plot_wealth_curve(model, env):
    obs = env.reset()
    done = [False]
    balances = []

    while not all(done):
        action, _ = model.predict(obs, deterministic=True)
        obs, reward, done, info = env.step(action)
        balances.append(info[0]['balance'])

    plt.plot(balances)
    plt.xlabel('Step')
    plt.ylabel('Balance')
    plt.title('Wealth Curve')
    plt.show()

    return balances

# Load the trained agent
model = PPO.load('ppo_portfolio_optimization')
balances = plot_wealth_curve(model, env)
```

Note that the `deterministic=True` flag in the `model.predict` function means that the model's deterministic action prediction is used, i.e., the model's output is deterministic given the same input. This is opposed to stochastic prediction, where the model's output can be different even with the same input, due to randomness in the model's policy (such as suggesting action based on a probability distribution).

Running the codes generates Figure 2.

Despite the seemingly amazing performance, a few cautionary notes should immediately follow here. First, this is mostly likely a result of overfitting. As we will see later, the out-of-sample

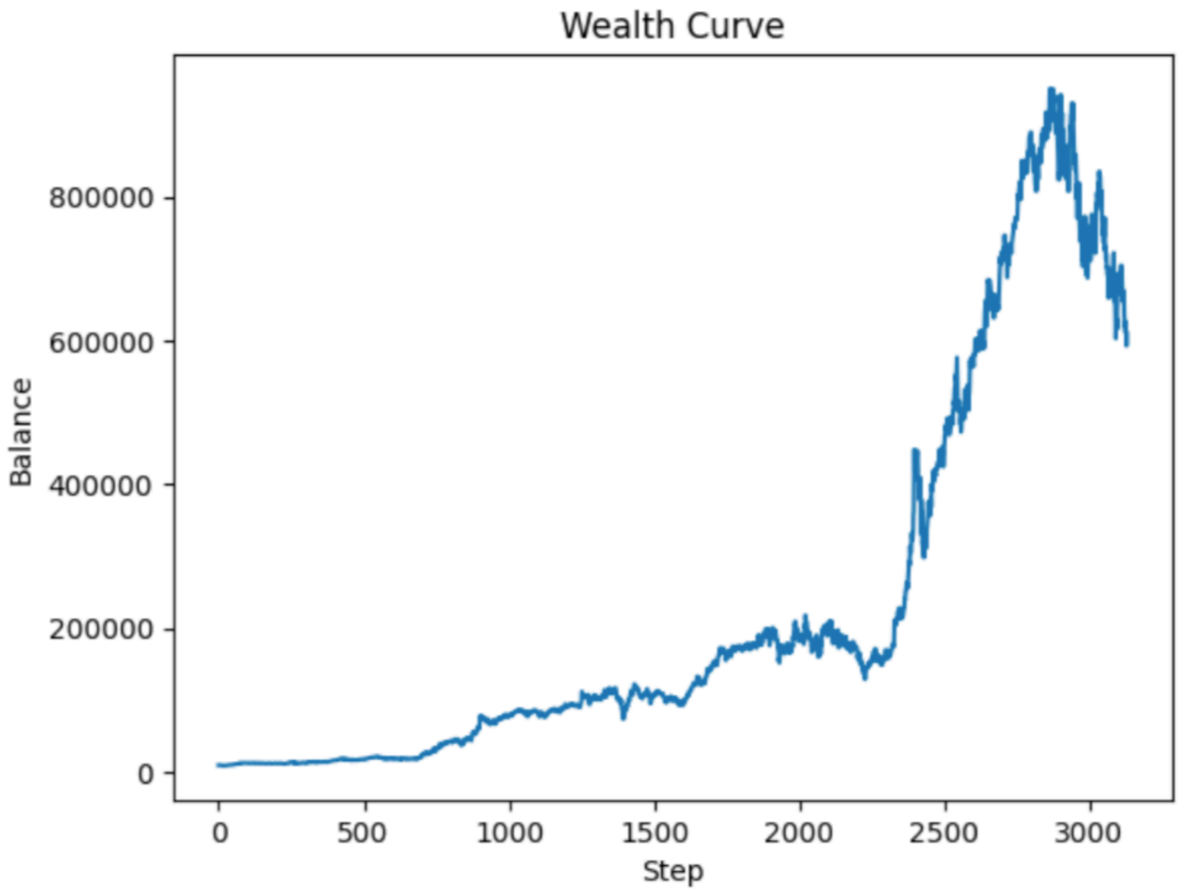

Figure 2: Wealth curve for the training period.

performance provides a drastically different performance compared to the in-sample performance. Second, this is an idealized environment. In reality, market frictions such as transaction cost need to be considered. Third, any trading strategy need to go through sufficient backtesting, out-of-sample testing, and live testing so as to obtain a good confidence on its actual performance.

Let us test this DRL agent for the first four months of 2023. This requires defining a new environment for out-of-sample testing purpose, as shown in the following code listing.

```
# Define the test period
test_start_date = '2023-01-01'
test_end_date = '2023-05-01'

# Create a new environment with the test data
test_env = PortfolioOptimizationEnv(tickers, window_size, test_start_date, test_end_date, in

# Wrap the environment in a DummyVecEnv for compatibility with Stable Baselines
test_env = DummyVecEnv([lambda: test_env])

# Plot the wealth curve on the test data
balances = plot_wealth_curve(model, test_env)
```

Running the codes generates Figure 3, suggesting a slight loss at the end of the trading period, besides a wide spread up to 10% in the daily portfolio value. This is a good example when developing quantitative trading strategies: a seemingly excellent trading strategy based on historical data may still lead to a poor test performance due to overfitting.

Finally, let us calculate the terminal return, also known as the total return. It is a measure of the gain or loss made from an investment over a period of time, and is expressed as a percentage of the investment's initial balance. See the following code listing for the calculation process.

```
def calculate_terminal_return(initial_balance, final_balance):
    return ((final_balance - initial_balance) / initial_balance) * 100

final_balance = balances[-1]
terminal_return = calculate_terminal_return(initial_balance, final_balance)

print(f'Terminal return: {terminal_return}%')

##### output #####
# Terminal return: -0.9989908567661042%
```

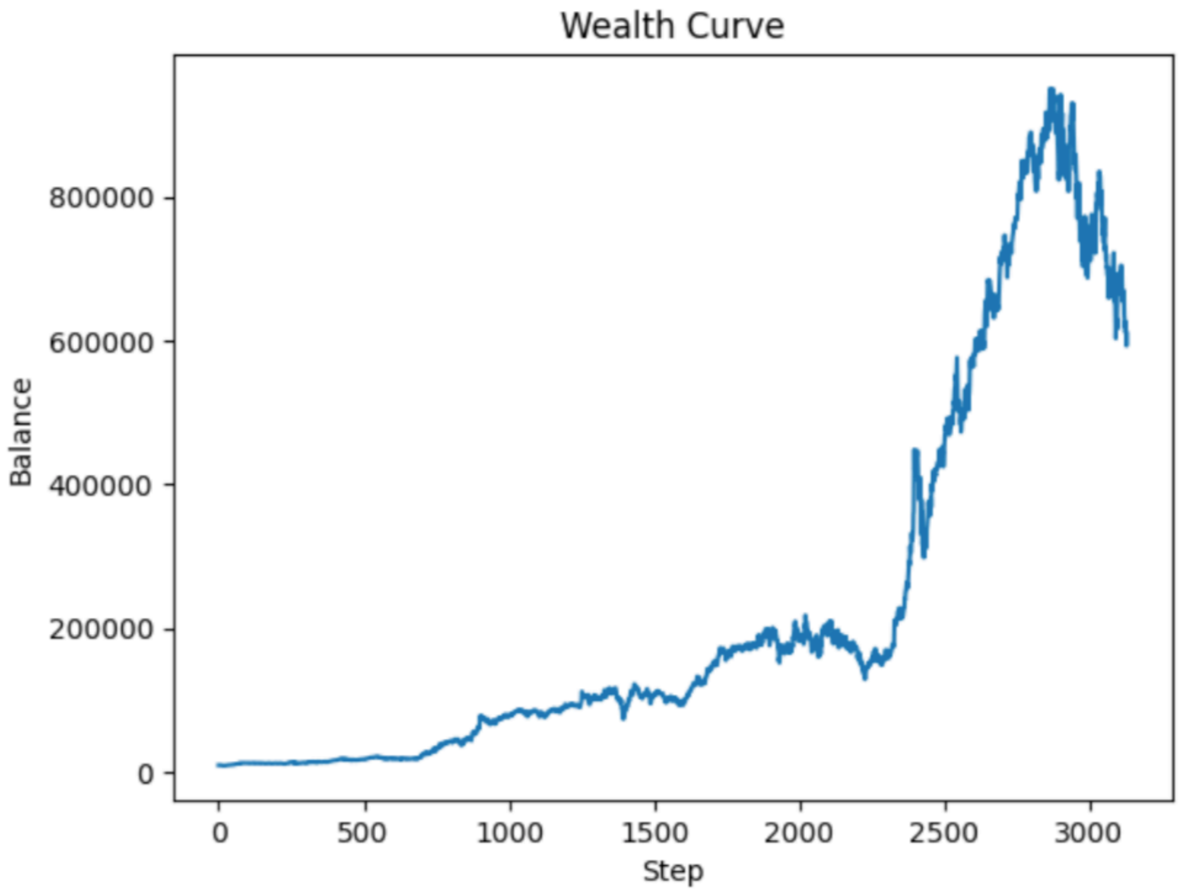

Figure 3: Wealth curve for the test period.

0.3 Summary

In this chapter, we introduced the theoretical background of both reinforcement learning and portfolio optimization, and showed their interaction from an endgame perspective. This perspective connects theory to practice, which manifests as a custom gym environment created to simulate a portfolio of stocks, and a DRL agent trained using Stable Baselines' implementation of the PPO algorithm to manage the portfolio. This a promising area that leverages the strengths of DRL to tackle the complexities inherent in financial markets.

We began with a rigorous mathematical introduction to portfolio optimization, focusing on the Modern Portfolio Theory. We explored how to define the return of a single asset as a percentage return, and then expanded this concept to a portfolio of assets. We introduced the calculation of expected returns and the covariance matrix, which provides a measure of the relationships between different assets in a portfolio. The construction of the Markowitz efficient frontier was also outlined, demonstrating the trade-off between risk and return.

The chapter then transitioned into the realm of reinforcement learning, with an emphasis on DRL. We provided a mathematical introduction to key concepts in reinforcement learning such as the Markov Decision Process, policy, state-value function, action-value function, and return. The pivotal Bellman equations were derived and explained in depth, offering a comprehensive understanding of their role in reinforcement learning algorithms.

We moved onto a practical perspective with the construction of a DRL environment for portfolio optimization, using real market data. We started by defining the environment, which included downloading historical stock data and defining the action and observation spaces. The agent's actions in this environment were to allocate a portfolio among a number of stocks, and the observations included historical stock prices. We then trained the agent using PPO and evaluated its performance by running the policy on both training and test environments.

Finally, we learned how to visualize the agent's performance over time by plotting the wealth curve, and how to calculate and interpret the terminal return of the agent's policy.

This chapter provides the foundation for future chapters, where we will delve into specific DRL algorithms and their application in the portfolio management task. The combination of theory and practice in this chapter provides a comprehensive understanding of the opportunities and challenges in applying DRL to portfolio optimization.

References

Markowitz, Harry. 1952. "Portfolio Selection." *The Journal of Finance* 7 (1): 77–91.