POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Differential Prediction-Enhanced Monte Carlo: A Fast Machine Learning-Aided Variance Reduction Method

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Luca Amadori**

Student ID: 248990
Advisor: Prof. Daniele Marazzina
Academic Year: 2024-25

Abstract

Monte Carlo simulations are essential for pricing financial derivatives but suffer from high computational costs. To address this, the Prediction-Enhanced Monte Carlo (PEMC) framework utilizes neural networks as control variates to reduce variance while maintaining unbiasedness. By shifting the bottleneck of data generation and model training to an offline phase, PEMC leverages these neural architectures to replace expensive online sample-path generation with highly efficient evaluations. However, standard PEMC requires large training datasets to be effective, leading to a computationally expensive offline phase.

This thesis explores the effectiveness of the PEMC approach and proposes Differential Prediction-Enhanced Monte Carlo, a method that integrates Principal Component Analysis (PCA) and differential PCA to filter input features based on their relevance to the payoff. By exploiting path-wise differentials, this approach allows the model to be trained on significantly smaller datasets without sacrificing accuracy. In this way, the model can be trained more frequently, generating predictors able to adapt more effectively to evolving market conditions.

Both methods are validated across realistic pricing scenarios. Numerical results show that PEMC achieves a variance reduction of approximately $30 - 70\%$ compared to standard Monte Carlo. Most importantly, the differential extension maintains this performance using orders of magnitude fewer training samples (e.g., 128 vs. 1.28 million), drastically reducing the computational bottleneck of the offline phase.

Keywords: Prediction-Enhanced Monte Carlo, Variance Reduction, Differential PCA, Deep Learning, Exotic Options Pricing, Control Variates.

Abstract in lingua italiana

Le simulazioni Monte Carlo sono essenziali per il pricing di derivati finanziari complessi, ma soffrono di costi computazionali elevati. Per risolvere questo problema, il framework Prediction-Enhanced Monte Carlo (PEMC) usa delle reti neurali come variabili di controllo per ridurre la varianza, mantenendo la stima non distorta. Spostando il collo di bottiglia della generazione dei dati e l'allenamento del modello ad una fase offline, il metodo PEMC sfrutta le architetture neurali per rimpiazzare la costosa generazione online di traiettorie campionarie casuali con valutazioni altamente efficienti. Tuttavia, per essere efficace, il metodo PEMC richiede grandi dataset di addestramento, comportando una fase offline computazionalmente costosa.

Questa tesi esplora l'efficacia del metodo PEMC e propone il Prediction-Enhanced Monte Carlo Differenziale, che integra l'Analisi delle Componenti Principali (PCA) con la PCA differenziale per filtrare le variabili di input del problema in base alla loro rilevanza nei confronti del payoff. Sfruttando i differenziali calcolati per ogni singolo percorso di simulazione, questo approccio permette di allenare il modello su dataset molto più piccoli senza sacrificare accuratezza. In questo modo, il modello può essere allenato più frequentemente, generando predittori in grado di adattarsi con maggiore efficacia alle condizioni di mercato del momento.

Entrambi i metodi vengono validati tramite scenari di pricing realistici. I risultati di questi esperimenti mostrano che il metodo PEMC ottiene una riduzione di varianza tra il 30 e il 70% circa rispetto al metodo Monte Carlo standard. Soprattutto, l'estensione differenziale mantiene queste prestazioni utilizzando ordini di grandezza in meno di campioni di addestramento (ad esempio, 128 contro 1.28 milioni), riducendo drasticamente il collo di bottiglia computazionale della fase offline.

Parole chiave: Prediction-Enhanced Monte Carlo, Riduzione della Varianza, PCA Differenziale, Deep Learning, Pricing di Opzioni Esotiche, Variabili di Controllo.

Contents

Introduction

Monte Carlo (MC) simulations are widely used in Computational Finance to compute the price of financial instruments, in particular when a closed-form pricing formula does not exist. Their main strength consists of providing unbiased estimates and precise error quantification through confidence intervals. However, when dealing with nested simulations or path-dependent payoffs, the MC method faces a trade-off: either pursue low error by using a large number of simulations, which is computationally intensive, or save computational time while sacrificing precision. A common way to enhance accuracy without increasing the problem size is using a variance reduction technique, e.g., the control variates (CV) method. Unfortunately, the classical CV approach requires the use of a suitable random variable with known expectation, hence it cannot be applied in many practical cases.

On the other hand, machine learning-based prediction models, called "fast evaluation" by practitioners [14], are quick alternatives to MC simulations that do not need specific conditions to be applied. These models can learn complex dependencies, but their main drawbacks are their black-box nature and lack of statistical control, which often lead to biased estimates without an error measure.

The trade-off between MC statistical properties, CV variance reduction and machine learning speed of execution makes their combination very intriguing. The realization of this idea is achieved in the Prediction-Enhanced Monte Carlo (PEMC) framework [20], which combines these methods to leverage their strengths and compensate for their weaknesses. In particular, the PEMC framework exploits a machine learning-based model to construct a control variate in a fast and parallelizable way, relegating the computationally intensive data generation and model training to an offline phase and exploiting the speed of machine learning models in the online evaluation. This approach allows for estimating the control variate's mean as the average of a large number of these quick simulations, bypassing the need for a closed-form formula while preserving a high precision in the estimate. This idea shifts the focus from the per-replication variance reduction, typical of the classical CV technique, to a total cost-aware one: in the PEMC framework, variance reduction is achieved on the entire set of simulated samples, not on a single simulation

run, in a cheap and parallelizable way.
The important advantages of the PEMC framework are that the adoption of machine learning techniques makes it faster with respect to MC simulations and it extends the use of CV to all the possible practical cases. At the same time, unbiasedness and rigorous error quantification typical of the MC method are preserved. On the other hand, the use of training datasets with a large number of rows is required by Lemma 6 and the Appendix in [20], to theoretically achieve significant variance reduction of the PEMC method compared to MC. Consequently, the offline phase (data generation and training) becomes very time-consuming, especially in complex scenarios such as pricing under stochastic local volatility models or in the Heath-Jarrow-Morton framework. A possible way to tackle this problem is to use more informative features, in such a way that a similar amount of information can be contained in fewer samples compared to the initial large dataset. To identify these highly informative variables, recent literature has introduced the combination of Principal Component Analysis (PCA) and differential PCA [15, 16]. This data preparation algorithm exploits partial derivatives of the label with respect to the features to obtain a low number of highly relevant variables. Building on this theoretical foundation, the main contribution of this thesis is the proposal of the Differential Prediction-Enhanced Monte Carlo (Differential PEMC) approach. By integrating the preprocessing step into the PEMC framework, the proposed method achieves the same level of variance reduction using a drastically lower number of training samples. The consequent reduction in training time is a significant innovation that could lead to a fully online use of the PEMC method, enabling more frequent model calibrations to better adhere to current market conditions.
The theoretical notions required for a complete understanding of the PEMC and Differential PEMC frameworks and their practical applications are in Chapter 1. Chapter 2 illustrates the problem solved by PEMC and describes the implementation of the method in five real-world scenarios, whose numerical results are reported in Chapter 3. Finally, the implementation of Differential PEMC to some of the examples described in Chapter 2 is presented in Chapter 4, together with their numerical results.

1 | Theoretical Framework

The theoretical concepts necessary to fully understand the problem and its implementation choices include the control variates method, skip connections in deep learning, the "log-sum-exp" trick, automatic differentiation, the differential PCA and the method to compute PCA on high-dimensional data. The first is the most important one, as the PEMC approach is fully based on it, while the use of skip connections is pivotal in implementing the model architectures described in Section 2.3. At the same time, the "log-sum-exp" trick is the crucial mathematical stratagem that makes it possible to implement the fourth example of Section 2.3. The last three concepts are related to the extensions of the PEMC approach; the differential PCA needs gradients computed by automatic differentiation and complex applications of the Differential PEMC approach necessitate the PCA for high-dimensional data to be computationally feasible.

1.1. Control Variates Technique

The control variates technique [10] is a variance reduction method. It exploits an auxiliary random variable with known first and second moments to obtain a better estimate of $\theta = \mathbb{E}[Y]$ compared to MC simulations.
Formally, let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space and $Y \in L^2(\Omega, \mathcal{F}, \mathbb{P})$ a square-integrable random variable, then the MC estimator for $\theta = \mathbb{E}[Y]$ is:

$$\hat{\theta}_{MC} = \frac{1}{n} \sum_{i=1}^{n} Y_i, \tag{1.1}$$

with n the number of MC samples and $Y_i, i = 1, ..., n$ i.i.d. random variables.
Let $X \in L^2(\Omega, \mathcal{F}, \mathbb{P})$ be another square-integrable random variable correlated with Y, and let $X_i, i = 1, ..., n$ be a set of i.i.d. copies of X, with known mean $\mathbb{E}[X]$, then:

$$\hat{\theta}_{CV} = \frac{1}{n} \sum_{i=1}^{n} \big(Y_i - \alpha(X_i - \mathbb{E}[X])\big) \tag{1.2}$$

is unbiased for θ:

$$\mathbb{E}[\hat{\theta}_{CV}] = \mathbb{E}\Big[\frac{1}{n}\sum_{i=1}^{n} Y_i - \frac{\alpha}{n}\sum_{i=1}^{n}(X_i - \mathbb{E}[X])\Big] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[Y_i] - \frac{\alpha}{n}\sum_{i=1}^{n}\mathbb{E}[X_i - \mathbb{E}[X]] =$$
$$= \mathbb{E}[Y] - \frac{\alpha}{n}\sum_{i=1}^{n}(\mathbb{E}[X] - \mathbb{E}[X]) = \theta,$$

for any value of α. Hence, $\hat{\theta}_{CV}$ could be used instead of (1.1) as an unbiased estimator for θ.
Furthermore, $Y_i(\alpha) = Y_i - \alpha(X_i - \mathbb{E}[X])$ has a variance:

$$Var(Y_i(\alpha)) = Var(Y) + \alpha^2 Var(X) - 2\alpha Cov(X, Y). \tag{1.3}$$

Hence, by accurately choosing X and α, it might be possible to achieve variance reduction with respect to MC, having $Var(Y_i(\alpha)) < Var(Y)$. Indeed, $Var(\hat{\theta}_{CV}) = \frac{Var(Y_i(\alpha))}{n}$ and $Var(\hat{\theta}_{MC}) = \frac{Var(Y)}{n}$. For this reason, the goal is to minimize (1.3) with respect to α, yielding:

$$\alpha^* = \frac{Cov(X, Y)}{Var(X)}.$$

By substituting this value in (1.3):

$$Var(Y_i(\alpha^*)) = Var(Y) + \frac{Cov(X,Y)^2}{Var(X)} - 2\frac{Cov(X,Y)^2}{Var(X)} = Var(Y) - \frac{Cov(X,Y)^2}{Var(X)} =$$
$$= Var(Y) - \frac{\rho^2 Var(Y)Var(X)}{Var(X)} = Var(Y)(1 - \rho^2),$$

with ρ the correlation between X and Y. As a consequence, by choosing the optimal α and any X such that $Cov(X, Y) \neq 0$ and $Var(X) \neq 0$, it is possible to reduce variance with respect to MC. Moreover, the higher the correlation between X and Y, the better the variance reduction.

1.2. Skip Connections in Deep Learning

Skip connections [12] are a deep residual learning tool that helps prevent the vanishing gradient problem. To understand the nature of this problem and the logic behind the use of skip connections, it is important to explain how the training of a neural network works. In simple terms, this procedure aims to minimize a certain loss function, usually the mean squared error between the label of the training dataset and the output of the

model, by changing the value of the network's parameters [25]. The update rule is:

$$w_i' = w_i - \lambda \frac{\partial L}{\partial w_i}, \tag{1.4}$$

with w_i the value of the i-th parameter before the update, w_i' the new parameter value, λ the learning rate and L the loss function. The mechanism that allows for computing the partial derivatives is called backpropagation. In practice, it exploits the chain rule for the computation of the partial derivative of a composite function to compute the partial derivative of the loss functions with respect to each parameter, starting from the last layer of the network and going backwards [31]. As a result, the partial derivative of the loss function with respect to the parameter w_1 is the product of partial derivatives that represent the layers from the one that contains w_1 to the last one, and the same holds for every parameter w_i. Especially for deep neural networks, many of the factors of the multiplication (usually the ones corresponding to activation functions and fully connected layers) are < 1, resulting in a very small product. This is the so-called "vanishing gradient problem", whose impact is that the parameters of the first layers of the model are updated by a very small amount, according to (1.4), leading to a loss of predictive power.
The problem can be solved using skip connections [12], which offer an alternative path for the gradient to flow during backpropagation. In practice, this method skips intermediate layers by directly adding the output of a shallower layer to the output of a deeper one. The mechanism is represented in Figure 1.1.

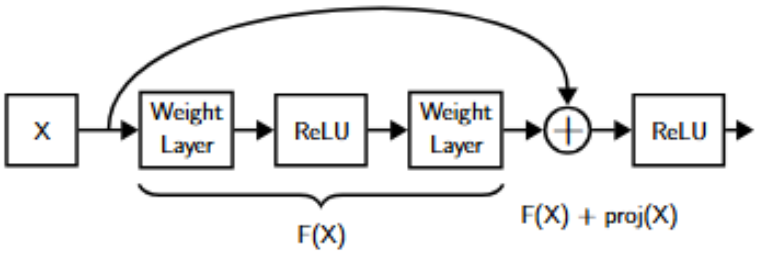

Figure 1.1: Skip connections in ResNet-style

In the figure above, F(X) is the residual function and proj(X) is the linear projection of X that matches the dimension of X with the dimension of F(X). In practice, F(X) could contain any number of layers, even though it is usually composed of one to three layers. A very important detail to notice is that the activation function of the final layer should be adopted after the addition.

1.3. Log-Sum-Exp Trick

The "log-sum-exp" trick [3] is used in statistical modeling and machine learning to prevent under- or overflow when computing the natural logarithm of the sum of exponential terms. The problem is due to the floating-point operations limits: if the exponent of an element of the sum is large and the corresponding exponential cannot be represented (overflow), then the exponential term becomes `inf` and so the sum; if the exponents of all the terms of the sum are too small (underflow) the sum becomes equal to 0, leading to a logarithm equal to $-\infty$. The trick that prevents these problems is based on the following mathematical property:

$$\ln\Big(\sum_{i=1}^{N} e^{x_i}\Big) = \ln\Big(\sum_{i=1}^{N} e^{x_i - c} \cdot e^{c}\Big) = c + \ln\Big(\sum_{i=1}^{N} e^{x_i - c}\Big).$$

For example, if $c = \max\{x_1, ..., x_N\}$, the effect is that the biggest term of the sum becomes $e^{c-c} = e^0 = 1$, while the exponents of the other terms become negative, hence the terms become less than 1, eliminating the risk of overflow.

1.4. Computing Gradients with Automatic Differentiation

1.4.1. Introduction to the Adjoint Method

The methods based on finite difference approximations are among the simplest for estimating price sensitivities. They consist of multiple MC simulations to estimate partial derivatives with perturbed input parameters, resulting in an increase in computing time with the number of input parameters.
On the other hand, the path-wise method [10], when applicable, produces unbiased estimators of price sensitivities without perturbing any parameters. Indeed, this technique computes the gradient of the price by differentiating the evolution of the underlying assets or the state variables along each path, making it possible to save time and improve the quality of the computed differentials.
Formally, let $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ be a filtered probability space with the filtration $\mathbb{F} = \{\mathcal{F}_n\}_{n=0,...,N}$ and $\theta \in \Theta \subseteq \mathbb{R}^k$ the vector of parameters with respect to which the sensitivities are computed. The discounted payoff function at the final state is $f(X_N(\theta))$, where $f : \mathbb{R}^d \to \mathbb{R}$ and $X = \{X_n\}_{n=0,...,N}$ an $\mathbb{F}$-adapted process with values in $\mathbb{R}^d$. The dynamics of X under a suitable risk-neutral measure $\mathbb{Q}$ are discretized by $X_{n+1} = F_n(X_n, \theta, Z_{n+1})$, $n = 0, ..., N-1$ with $Z_1, ..., Z_N$ i.i.d. standard normal random vectors independent

of $\mathcal{F}_0$. In practice, X could be, for example, a vector of underlying assets' prices or a vector of forward rates.
Let $V(\theta) = \mathbb{E}[f(X_N(\theta))]$, the path-wise method estimates $\nabla_\theta V$ (i.e., the vector of price sensitivities) as:

$$\nabla_\theta f(X_N) = \Big(\frac{dX_N}{d\theta}\Big)^T \nabla_x f(X_N), \tag{1.5}$$

where $\nabla_x f(X_N) \in \mathbb{R}^d$ is the gradient of the discounted payoff with respect to the final state X_N, $\frac{dX_N}{d\theta} \in \mathbb{R}^{d\times k}$ is the Jacobian matrix of the sensitivity of the final state with respect to θ and the equality comes from the chain rule of differentiation.
The path-wise estimator is unbiased if:

$$\mathbb{E}[\nabla_\theta f(X_N)] = \nabla_\theta \mathbb{E}[f(X_N)],$$

namely, if the gradient operator and the expectation can be interchanged. The realization of four conditions is sufficient to make the interchange valid [10]:

- $F_n(x, \theta, z)$ is differentiable with respect to x and θ with probability 1 for every $n = 0, ..., N-1$.
- Let $D_f \subseteq \mathbb{R}^d$ be the set of points where $f(x)$ is not differentiable, then $\mathbb{P}(X_N(\theta) \in D_f) = 0$.
- f is Lipschitz continuous.
- There exists an integrable random variable C_X such that $||X_N(\theta_1) - X_N(\theta_2)||_2 \leq C_X||\theta_1 - \theta_2||_2$.

In practice, the most important condition to check is the Lipschitz continuity of f, as highlighted in [9].
The estimator (1.5) can be computed with two techniques: the forward method and the adjoint method. The former is the classical way to calculate the path-wise estimator: as the name suggests, the computation proceeds forward in time. In particular, let:

$$D_n = \frac{\partial F_n}{\partial x}(X_n, \theta, Z_{n+1}) \in \mathbb{R}^{d\times d}, \quad B_n = \frac{\partial F_n}{\partial \theta}(X_n, \theta, Z_{n+1}) \in \mathbb{R}^{d\times k}, \tag{1.6}$$

the Jacobian matrices of the function F_n, $\forall n = 0, .., N-1$, then $J_n = \frac{dX_n}{d\theta} \in \mathbb{R}^{d\times k}$ is computed with the recursion:

$$J_{n+1} = D_n J_n + B_n,$$

with J_0 given and $n = 0, ..., N-1$. Thus, a matrix-by-matrix multiplication is computed at each step of the recursion (i.e., $D_n J_n$) and the cost of the recursion depends on k, the

number of components of θ, since $J_n \in \mathbb{R}^{d\times k}$. Finally, the estimator is computed as:

$$\nabla_\theta f(X_N) = J_N^T \ \ \nabla_x f(X_N).$$

On the other hand, the adjoint method executes the calculation backwards, leading to the same result as the forward method with less computational effort. More in detail, $\nabla_\theta f(X_N)$ is calculated as:

$$\nabla_\theta f(X_N) = J_0^T V_0 + \sum_{n=0}^{N-1} B_n^T V_{n+1} \tag{1.7}$$

with $V_n = D_n^T V_{n+1}$, $V_N = \nabla_x f(X_N) \in \mathbb{R}^d$, D_n, B_n defined as in (1.6) and $n = N-1, ..., 0$, i.e. the recursion starts from $n = N-1$ and moves towards $n = 0$. This arrangement of the formula leads to a vector recursion ($V_n = D_n^T V_{n+1}$) instead of the matrix recursion of the forward method ($J_{n+1} = D_n J_n + B_n$), leading to a computational cost of $O(d^2)$ instead of $O(d^2 k)$ for each step. It is important to notice that the cost of the vector recursion does not depend on the number of parameters k, since B_n and J_0, the quantities of the computation whose dimensions depend on k, are involved just in the accumulation part (computational cost $O(kd)$ per step). This characteristic makes the adjoint method beneficial when the goal is to compute sensitivities of a single function f with respect to multiple parameters, since the computational cost scales better with the state dimension d ($O(kd + d^2)$) compared to the forward method ($O(kd^2)$) [9].

1.4.2. The Adjoint Method in PyTorch

To sum up, the main innovation of the adjoint method with respect to the forward one is to fix the final price, usually with a MC simulation of X, and, instead of computing the sensitivities during the simulation, move backwards in the simulated path to keep track of the influence of each parameter on the final quantity. This idea is exploited by PyTorch [27] to compute the gradient of a function with respect to its input parameters using the Automatic Differentiation in reverse (or adjoint) mode (AAD) [29], whose theoretical description is given in Appendix A in [9]. The main use of the AAD in PyTorch is the computation of gradient of the loss function during the training procedure of a deep learning model, since PyTorch was born as a deep learning library [27], but the same logic can be applied to every scenario where the derivative of a function with respect to some parameters has to be computed (e.g., Greeks computation).

`torch.autograd` is the AAD engine in PyTorch, which exploits a computational graph

to calculate the gradient of a function with respect to a vector of parameters [28]. A computational graph is a directed acyclic graph (DAG) whose leaves are the input tensors (i.e., the parameters with respect to which the partial derivatives are computed), and the roots are the output tensors (i.e, the functions whose gradient is computed). The computational graph is built during the forward pass (i.e, the sequential execution of operations that depend on the input tensors), where the gradient function of each performed operation is stored in the DAG. Once the graph is complete, a call to `torch.autograd` on the DAG root starts the backward pass. In this phase, the engine computes the vector-Jacobian product step by step: the gradient of each function stored in the DAG during the forward pass is computed and multiplied by the gradient propagated from the subsequent nodes, equivalent to the recursion $V_n = D_n^T V_{n+1}$ in the adjoint method [30]. These gradients are accumulated in the respective tensor's `.grad` attribute and the chain rule makes it possible to propagate the computation to the leaf tensors, using (1.7).

1.5. Differential PCA

In machine learning, reducing the initial set of features of the training dataset to a subset of significant ones can largely improve model performance. However, identifying correct features is complicated in scenarios such as pricing exotic options under complex models, particularly when selection is performed manually. One of the simplest solutions to the problem of dimension reduction is Principal Component Analysis (PCA). It performs an orthonormal transformation of input data to remove constant and redundant columns through the following steps [15]:

- Perform the eigenvalue decomposition of the covariance matrix of input features.
- Remove numerically zero eigenvalues and corresponding eigenvectors.
- Transform input data using the filtered eigenvalues and eigenvectors.

Formally, defining the variation of the input matrix X along the normalized eigenvector u_i as $\mathbb{E}[(X \cdot u_i)^2]$, the eigenvalue corresponding to u_i measures the variation of X along u_i. Consequently, filtering is unsupervised (i.e., it does not take into account the label of the dataset) and strictly based on variation. However, this metric is often inadequate for derivatives pricing and risk management, since a small variation of X along a very relevant axis can significantly affect the price or payoff measured by the dataset label, while a large variation along an irrelevant axis may not [16]. Therefore, the amount of dimension reduction that PCA can provide is limited, since it must use a numerically insignificant filtering threshold in order to avoid the elimination of important features.

These observations show that the PCA algorithm, fully unsupervised, can be improved by introducing supervision.
To overcome these limitations, one can use differential PCA [15] as a further preprocessing step. Unlike classic PCA, differential PCA is a supervised dimension reduction algorithm that filters inputs based on relevance to the payoff rather than variation. Formally, the strong relevance of some axis u is defined in [16] as $\mathbb{E}[(u \cdot Z)^2]$, where Z is the path-wise estimator of the vector of price sensitivities (also called path-wise differential), introduced in Subsection 1.4.1. For this reason, differential PCA assumes that the input data contains path-wise differentials along with features and labels (usually prices or payoffs) and that a prior PCA step is performed on both features and differentials, as illustrated on pages $30 - 31$ in [15].
The procedure of differential PCA is the following:

- Perform the eigenvalue decomposition of the noncentral covariance matrix of input path-wise differentials.
- Remove small eigenvalues and corresponding eigenvectors.
- Transform input data using the filtered eigenvalues and eigenvectors.

As stated in Appendix 2 in [15], "differential PCA is simply PCA on differential labels". The advantage of this further preprocessing step is that it preserves the orthonormality of inputs from a prior PCA step, while allowing safer and more aggressive filtering than PCA. Unlike variation, which takes into account just the magnitude of inputs, relevance uses the partial derivatives to identify risk factors driving the labels. This allows for the elimination of axes that may vary significantly but have negligible influence on the price or payoff. A full data preparation algorithm involving both PCA and differential PCA is on pages $32 - 34$ in [15].

1.6. PCA for High-Dimensional Data

Let X be an $N \times D$ data matrix, with N samples and D features. As seen above, the first step in both PCA and differential PCA is to perform an eigenvalue decomposition of the squared matrix computed as $\frac{X^T X}{N}$, which represents the covariance matrix for centered features in PCA and the noncentral covariance matrix for path-wise differentials in differential PCA. Consequently, this is a $D \times D$ matrix, and typical algorithms for computing its eigenvectors have a computational cost of $O(D^3)$. In practice, when $D \gg N$, a direct application of PCA to such matrices is computationally infeasible. However, Section 12.1.4 in [2] introduces a mathematical trick to make the application of PCA

computationally feasible even for a large number of features.
The eigenvector equation of the target matrix is:

$$\frac{1}{N}X^T X u_i = \lambda_i u_i,$$

with u_i and λ_i the i-th eigenvector and eigenvalue respectively. Pre-multiplying both sides of the equation by X:

$$\frac{1}{N}XX^T(Xu_i) = \lambda_i(Xu_i),$$

and defining $v_i := Xu_i$:

$$\frac{1}{N}XX^T v_i = \lambda_i v_i, \tag{1.8}$$

the eigenvector equation for $\frac{XX^T}{N}$, a $N \times N$ matrix, is obtained. Remarkably, this matrix has the same non-zero eigenvalues as the original matrix and the computational cost of the eigenvalue decomposition becomes $O(N^3)$, making the application of the PCA computationally feasible.
Finally, the eigenvectors of the original matrix can be retrieved by a transformation of the ones of the $N \times N$ matrix. Indeed, multiplying both sides of (1.8) by X^T:

$$\left(\frac{1}{N}X^T X\right)(X^T v_i) = \lambda_i(X^T v_i),$$

namely $(X^T v_i)$ is an eigenvector of the original matrix with eigenvalue λ_i. The final step is the normalization of these eigenvectors, which is achieved by:

$$u_i = \frac{1}{(N\lambda_i)^{1/2}}(X^T v_i).$$

2 | Problem Description and Implementation

2.1. Problem Description

Let $(\Omega, \mathcal{F}, \mathbb{F}, \mathbb{P})$ be a filtered probability space with the filtration $\mathbb{F} = \{\mathcal{F}_t\}_{t\geq 0}$. In derivatives pricing applications, the fundamental quantity to estimate is the value of the option at time $t = 0$, defined as:

$$V_0(\theta) = \mathbb{E}_\theta^{\mathbb{Q}}[f_\theta(Y)|\mathcal{F}_0], \tag{2.1}$$

where $Y \in \mathbb{R}^d$, $d = 1, 2, ...$ represents a d-dimensional random vector representing the underlying asset path or terminal value, $\mathbb{Q}$ is a suitable risk-neutral measure under which the conditional expectation is evaluated, $f : \mathbb{R}^d \to \mathbb{R}$ is the discounted payoff function and the subscript θ indicates that both f and $\mathbb{E}^{\mathbb{Q}}$ can depend on the d-dimensional parameter $\theta \in \Theta \subseteq \mathbb{R}^d$, with Θ the parameter space. For notation simplicity, $\mathbb{E}_\theta^{\mathbb{Q}}[\cdot|\mathcal{F}_0]$ is denoted by $\mathbb{E}^{\mathbb{Q}}[\cdot]$ in what follows.

In particular, it is crucial to obtain an unbiased estimate of (2.1) for any $\theta \in \Theta$. A very common way to retrieve it is to use the MC method, which consists of simulating a large number of $f_\theta(Y_i), i = 1, ..., N$ and averaging them in order to get the derivative price. However, these simulations can be very expensive, especially in contexts like path-dependent options [10]; for this reason, they can be substituted by an approximation of the quantity of interest produced by a machine learning model [7, 17]. Thanks to the possibility to use a convex and differentiable loss function (the mean squared error), their straightforward training using easily accessible solvers and their ability to track complex dependencies among data, neural networks are an ideal choice as the prediction model.

A very important feature of the MC method is represented by the simple statistical error quantification that it offers through confidence intervals. On the other hand, machine learning-based approaches do not allow this possibility [4, 17], hence it is necessary to mix these two approaches in order to merge the fast evaluation of machine learning models with the statistical properties of the MC method.

Taking a step further, variance reduction methods are standard tools exploited to increase the precision of the estimates. Among them, the control variates technique introduced in Section 1.1 offers a drop in MC variance subordinated to the knowledge of the expected value of an auxiliary function. Due to this requirement, there are many practical cases in which this approach can not be applied. In this context, the idea of merging this technique with machine learning-based methods, as highlighted above for the MC approach, is very effective since it expands CV usage to all the practical cases while maintaining unbiasedness and rigorous error quantification. The Prediction-Enhanced Monte Carlo (PEMC) approach [20] is based on this integration between statistical methods and machine learning: it proposes an estimator for (2.1) inspired by the CV technique that enhances the variance reduction of the latter while exploiting a neural network-based auxiliary function which guarantees a faster evaluation with respect to the MC method and relaxes the hypothesis of known mean typical of CV. The PEMC estimator is of the form:

$$\text{PEMC}(\theta) := \frac{1}{n}\sum_{i=1}^{n}\big(f_\theta(Y_i) - g(\theta, X_i)\big) + \frac{1}{N}\sum_{j=1}^{N} g(\theta, \tilde{X}_j), \tag{2.2}$$

where $N = 10n$, $X = X(\theta)$ is a variable that depends on $Y = Y(\theta)$ and $g(\theta, x)$ is the estimate of f_θ obtained by the neural network. Moreover, each pair (X_i, Y_i) is simulated together, whereas each $\tilde{X}_j$ has the same marginal distribution as X and is simulated independently from all (X_i, Y_i). For further implementation details on the PEMC approach, refer to Section 2.2.

Due to its rationale and the formula it uses, the PEMC approach can be seen as a modern version of the CV technique [20]. The link between the estimator in (2.2) and the CV one in (1.2) is even stronger than this, and it can be seen clearly when a parameter α is introduced:

$$\begin{aligned}\text{PEMC}(\theta, \alpha) &= \frac{1}{n}\sum_{i=1}^{n}\big(f_\theta(Y_i) - \alpha g(\theta, X_i)\big) + \frac{1}{N}\sum_{j=1}^{N}\alpha g(\theta, \tilde{X}_j) = \\ &= \frac{1}{n}\sum_{i=1}^{n} f_\theta(Y_i) - \alpha\Big(\frac{1}{n}\sum_{i=1}^{n} g(\theta, X_i) - \frac{1}{N}\sum_{j=1}^{N} g(\theta, \tilde{X}_j)\Big).\end{aligned}$$

Suppose g is fixed in the equation above, then the coefficient that minimizes the variance of $\text{PEMC}(\theta, \alpha)$ is:

$$\alpha^* = \frac{Cov\big(f_\theta(Y), g(\theta, X)\big)}{\big(n/N + 1\big)Var\big(g(\theta, X)\big)}.$$

Since $N \gg n$ and, in the ideal scenario, $g(\theta, X) = \mathbb{E}^{\mathbb{Q}}[f_\theta(Y)|\theta, X]$, then $Cov\big(f_\theta(Y), g(\theta, X)\big) = Var\big(g(\theta, X)\big)$ and $Cov\big(f_\theta(Y) - g(\theta, X), g(\theta, X)\big) = 0$ [20]. Consequently, the optimal coefficient becomes $\alpha^* = \frac{1}{n/N+1}$, which asymptotically approaches 1. The complete derivation of α^* and the related proofs are in Appendix A.
The PEMC estimator shares two important statistical properties with the CV one: unbiasedness and variance reduction with respect to the MC method. Indeed, the PEMC estimator is an unbiased estimator of (2.1) thanks to the fact that both X_i and $\tilde{X}_j$ have the same marginal distribution $\forall i, j,\ i = 1, ..., n, j = 1, ..., N$. An exhaustive discussion on variance reduction and cost efficiency of the PEMC approach with respect to the MC method can be found in [20].
To demonstrate its widespread effectiveness, PEMC is compared to the MC method and, when applicable, to the CV approach in various pricing applications described in Section 2.3, whose results are presented in Chapter 3.

2.2. PEMC Implementation

2.2.1. Programming Choices

The entire codebase is written in Python and makes extensive use of the PyTorch library [27]. To accelerate the simulation and the training and evaluation phases, the code leverages NVIDIA L4 GPU acceleration and employs vectorized operations to fully exploit the library's optimized C++ backend. The simulations discussed in this work were, in part, performed on the HPC Cluster of the Department of Mathematics of Politecnico di Milano, which was funded by MUR grant Dipartimento di Eccellenza 2023-2027.

2.2.2. The Choice of X

The choice of a suitable X in (2.2) is crucial to obtain a well-performing estimator. The main requirements that X should satisfy are:

- Being a low-dimensional transformation of Y.
- Being a reasonable predictor for $f_\theta(Y)$.
- Having a marginal distribution whose simulation is cheap and highly parallelizable.

According to these properties, an effective choice for X is the sum of Brownian increments [20]:

$$X := \sum_{i=1}^{T/\Delta t} \Delta W_{i\Delta t}, \tag{2.3}$$

where $\Delta W_{i\Delta t} \sim N(0, \Delta t), \forall i = 1, ..., \frac{T}{\Delta t}$. Indeed, in the applications proposed in Section 2.3, Y will be a stochastic process with dynamics driven by one or more stochastic differential equations, whose variance can be explained in a large portion by the sum of Brownian increments. Moreover, the chosen X has a Gaussian distribution which can be generated efficiently. Even though this choice of X gives sufficient variance reduction in the analyzed applications, when the simulation path is very long, it is possible to increase the dimension of X by dividing the sum into smaller chunks.

2.2.3. Data Generation

The first step to build (2.2) is the data generation. Given the parameter space Θ, a sample of $\theta_i \in \Theta$ is drawn using uniform sampling and it is used to generate a pair $(X(\theta_i), Y(\theta_i))$. Then $f_{\theta_i}(Y(\theta_i))$ is computed and is used as a label in the training data. Notice that in all the applications in the following section, f is considered as the payoff of a certain derivative, omitting the discount factor. On the other hand, the features vector is $(\theta_i, X(\theta_i))$. The training dataset is composed of N_{train} samples of this kind.
The nature of this data generation process allows an "on the fly" data generation and training: a small batch of samples is generated, the model is trained on this data and then the batch is discarded. This mechanism is pivotal in the PEMC approach, since a large volume of data is necessary in order to achieve a well-performing prediction model and without this workflow, it would not be possible to achieve such performance due to data storage overheads.
Practically, the effect of "on the fly" generation of the training data is obtained by the combination of the following lines of code:

```
class PEMCDataset(IterableDataset):
    def __init__(self, ...):
        #...rest of the code...
    def __iter__(self):
        for batch_idx in range(self.batches_per_epoch):
            # Computation of the batch size in order to manage the
                last batch, that could be smaller than the
               previous ones
            current_batch_size = int(min(self.batch_size, self.
               num_samples - batch_idx * self.batch_size))

            theta = torch.zeros((current_batch_size, self.n_params
               ), ...)
            for i, (low, high) in enumerate(self.intervals):
```

```
                theta[:, i].uniform_(low, high)

            #...simulations of X and payoff...

            yield theta, X, payoff
```

Listing 2.1: Custom training dataset class

```
class training:
    def __init__(self, ...):
        #...initialization code...

        self.train_dataset = PEMCDataset(..., batch_size=self.
            batch_size)
        self.train_loader = DataLoader(self.train_dataset,
            batch_size=None)

    #...validation logic...

    def fit(self, ...):
        #...training code...
        for epoch in tqdm(range(num_epochs)):
            #...rest of the code...
            for theta, x, y in self.train_loader:

        #...final part of the training logic...
```

Listing 2.2: Integration in the training loop

In particular, the `PEMCDataset` class inherits from `IterableDataset`, which is built for streaming data thanks to the fact that its `__init__` method does not allocate data in memory, unlike the standard `Dataset` class. Indeed, the data generation happens inside the `__iter__` method, where the simulation functions produce batches of data that feed the training part of the code. In this section, by setting `batch_size=None` inside the `DataLoader` that loads data from `PEMCDataset` and combining it with `for theta, x, y in self.train_loader:` inside the training loop, it is possible to obtain the effect of training the model using a batch of data and then discarding it. Because of the nature of the implementation, where the data is passed to the training class in small batches, the `__iter__` method of the dataset class is called many times during the computation of an epoch. To optimize this mechanism, `__iter__` can be turned into a generator function

by using `yield`, which makes the function use lazy evaluation. Indeed, when `yield` is called, the returned values are sent back to the caller of the generator function, in this case the `DataLoader`, but differently from the behaviour of `return`, the execution of the function is suspended and not terminated. In this way, when the `DataLoader` asks for a new batch of data, it is very fast to produce it.
Moving to data generation of the validation sets, the logic is slightly different: both the validation sets used, one for implementing the early-stopping logic and the other for hyperparameter tuning with Optuna, are generated and stored in full (i.e., not batch-wise) before the training occurs. This choice is a consequence of their reduced dimension compared to the training dataset's one (the number of rows of the validation sets is set to be 10% of N_{train}), which makes them storable in memory in their entirety. The implementation of this mechanism is very similar to the training dataset's one, with some small changes:

```
    class ValidationDataset(IterableDataset):
    def __init__(self, ...):
        #...initialization code...

        # Generate all theta values
        self.theta = torch.zeros(...)
        for i, (low, high) in enumerate(self.intervals):
            self.theta[:, i].uniform_(low, high)

        #...simulations of self.X and self.payoff...

    def __iter__(self):
            yield self.theta, self.X, self.payoff
```

Listing 2.3: Custom validation dataset class

Here the simulations happen inside `__init__` and the simulated values are returned in `__iter__`. Thanks to this generation procedure, it is possible to use the same validation dataset for early-stopping in each training epoch and to switch to a different validation dataset for hyperparameter tuning, which is the same for each Optuna trial.

2.2.4. Neural Network Training

The training phase employs the Adam optimizer [18] and the model is trained to minimize the Mean Squared Error (MSE) loss:

$$\min_{g} \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \left(f_{\theta_i}(Y(\theta_i)) - g(\theta_i, X(\theta_i))\right)^2.$$

This choice of the loss function is a consequence of the fact that $g(\theta, X(\theta))$ should approximate $\mathbb{E}^{\mathbb{Q}}[f_\theta(Y(\theta))|\theta, X(\theta)]$ and mathematically the minimizer of the chosen loss function is such conditional expectation. Notably, since the property holds pointwisely for each fixed $\theta \in \Theta$, it also holds in the PEMC setting where θ is uniformly random (see Appendix [20]).

During the training process, the early-stopping technique is implemented. As suggested in [20], the metric used to evaluate the early-stopping is a slight variation of the Mean Absolute Relative Error (MARE):

$$\frac{\left|\frac{1}{N}\sum_{i=1}^{N} g(\theta_i, X(\theta_i)) - \frac{1}{N}\sum_{i=1}^{N} f(Y(\theta_i))\right|}{\left|\frac{1}{N}\sum_{i=1}^{N} f(Y(\theta_i))\right|}, \tag{2.4}$$

with N the number of rows of the dataset on which the metric is computed. The use of this quantity instead of the classical MSE to assess early-stopping is motivated by the difficulty in interpreting the raw MSE value in this context; specifically, it is unclear solely from MSE when the network is sufficiently trained. Conversely, the modified MARE is highly effective in the context of PEMC as it directly measures how well $\mathbb{E}^{\mathbb{Q}}[g]$ approximates $\mathbb{E}^{\mathbb{Q}}[f]$. This is crucial for the PEMC approach: while the term $\frac{1}{N}\sum_{j=1}^{N} g(\theta, \tilde{X}_j(\theta))$ in Eq. (2.2) is guaranteed to converge to $\mathbb{E}^{\mathbb{Q}}[g]$ by the law of large numbers, the modified MARE ensures that this limit $\mathbb{E}^{\mathbb{Q}}[g]$ aligns with the true target expectation $\mathbb{E}^{\mathbb{Q}}[f]$, leading to a minimization of the systematic bias. Furthermore, a low value of (2.4) indicates low bias in the training, whereas a low MSE could simply result from the model learning the statistical noise contained in the data. Moreover, along with keeping track of the number of consecutive epochs in which the modified MARE does not decrease, the training procedure stops if the value of the loss function is $< 1\%$, indicating exceptional performance of the PEMC estimator [20].

To ensure a correct and efficient training procedure, weights and biases initialization is done. The former follows [11], whereas the latter simply sets the biases to 0. The choice of the "Kaiming He" weights initialization technique is due to the massive use of ReLU activation functions in the application's networks, which are compatible with this initial-

ization choice.

2.2.5. Hyperparameter Tuning

Hyperparameter tuning is performed exploiting the Optuna library:

```
def run_optuna_study():
    hyperparameters_val_set = ValidationDataset(...)
    hyperparameters_loader = DataLoader(hyperparameters_val_set,
       batch_size=None)

    def objective(trial):
        model = None
        trainer = None
        try:
            #...ranges of the hyperparameters to tune...

            model = PEMCNetwork(...)
            trainer = training(...)
            trainer.fit(..., val_loader=early_stopping_loader)

            #...compute the loss on "hyperparameters_val_set"...

            return loss

        except Exception as e:
            print(f"Trial␣failed␣with␣error:␣{e}")
            raise e
        finally:
            if model is not None: del model
            if trainer is not None:
                if hasattr(trainer, 'train_dataset'): del trainer.
                   train_dataset
                del trainer
            gc.collect()
            torch.cuda.empty_cache()

    study = optuna.create_study(direction="minimize", sampler=
       optuna.samplers.TPESampler())
    study.optimize(objective, n_trials=n_trials)
```

```
32      return study.best_params
```

Listing 2.4: Hyperparameter tuning with Optuna

The MSE loss that determines the best set of hyperparameters is computed on `hyperparameters_val_set`, which is the same among all the different Optuna trials. Furthermore, the block in `finally` is used for memory management: after a set of hyperparameters is evaluated, whether the evaluation was successful or not, the model, the training dataset and the `training` object created during the trial are deleted in order to avoid `CUDA out of memory` errors. Finally, the hyperparameters that minimize the MSE loss are returned in order to be used in the final retraining of the model.

2.2.6. Computing the PEMC Estimator

During the evaluation phase, a specific $\hat{\theta} \in \Theta$ is given, e.g., calibrated from the market. In order to compute the PEMC estimator, it is necessary to generate n pairs of $(Y_i(\hat{\theta}), X_i(\hat{\theta}))_{i=1,\dots,n}$ from the risk neutral measure $\mathbb{Q}$ and $N = 10n$ samples of $(\tilde{X}_j(\hat{\theta}))_{j=1,..,N}$ with the same marginal distribution of X, that are independent of the couples $(Y_i(\hat{\theta}), X_i(\hat{\theta}))_{i=1,\dots,n}$ previously generated. Given the choice of X discussed in Subsection 2.2.2, its marginal distribution will always be Gaussian in the applications. Finally, the PEMC estimator can be computed using formula (2.2):

$$\mathrm{PEMC}(\hat{\theta}) := \frac{1}{n}\sum_{i=1}^{n}(f_{\hat{\theta}}(Y_i(\hat{\theta})) - g(\hat{\theta}, X_i(\hat{\theta}))) + \frac{1}{N}\sum_{j=1}^{N} g(\hat{\theta}, \tilde{X}_j(\hat{\theta})).$$

Overall, both the data generation and the training part are very expensive in terms of computational resources and time, for this reason, in the PEMC approach they are conducted offline. In this way, the pre-trained function g can be stored and plugged into real-time pricing applications whenever a new set of evaluation parameters arises.

2.3. Comparison Between PEMC, MC and CV in Applications

The effectiveness of the PEMC estimator's variance reduction with respect to MC and, when available, CV is computed in terms of root mean squared error (RMSE) on a large number of independent experiments, as in [20]. In particular, a set of values of n is decided and hundreds of independent experiments are performed for each element of the set, considering n as the number of samples of MC, PEMC and CV. Each experiment

produces a value of PEMC, MC and CV estimator, which is computed using $\hat{\theta} \in \Theta$ (called `theta_eval` in the code) as the parameter set. The RMSE is computed against a ground truth value of the estimator, calculated using a large number of MC simulations using $\hat{\theta}$ as well.

```
# Evaluation parameters
num_runs = ...
n_values = [...]
theta_eval = [...]
batch_eval = ...

set_all_seeds(42)

theta_tensor = torch.tensor(theta_eval, ...).repeat(batch_eval, 1)

#...Ground truth computation...

# RMSE Evaluation
rmseMC = np.zeros(len(n_values))
rmsePEMC = np.zeros(len(n_values))

for i, n in enumerate(n_values):
    print(f"Evaluation with n={n}")
    errMC = 0
    errPEMC = 0

    for j in range(num_runs):
        MC = evaluator.evaluate_MC(n, theta_tensor, ...)
        PEMC = evaluator.evaluate_PEMC(model, 10 * n, n,
           theta_tensor, ...)

        errMC += (MC - ground_truth) ** 2
        errPEMC += (PEMC - ground_truth) ** 2

    rmseMC[i] = np.sqrt(errMC / num_runs)
    rmsePEMC[i] = np.sqrt(errPEMC / num_runs)

# Results
errors = pd.DataFrame(
    data=[rmseMC, rmsePEMC],
```

```
    columns=[f'n={n}' for n in n_values],
    index=['Monte Carlo (MC)', 'PEMC']
)
print(errors)
```

Listing 2.5: Comparison between PEMC, MC and CV results

The choice of the RMSE as a metric to evaluate variance reduction is a consequence of the theoretical unbiasedness of the PEMC estimator [20]. In statistics, the RMSE of an estimator with respect to the parameter to estimate is equal to the square root of the sum of the estimator's variance and its squared bias. Consequently, the RMSE of an unbiased estimator with respect to the parameter to estimate is exactly equal to the estimator's standard deviation. In the PEMC approach, where the ground truth is estimated from a large number of MC simulations, the theoretical equivalence is not exact, since the RMSE also contains an error term related to the fact that the ground truth is not the exact value of the parameter to estimate, but still, the RMSE remains a variance-related metric.
The following paragraphs focus on each application case, highlighting the additions and the modifications to the implementation with respect to the general scheme described up to this point.

2.3.1. PEMC Variance Reduction

The first use case for the PEMC approach is the computation of the expected payoff of an arithmetic Asian call option, where the dynamics of the underlying asset are modeled through a Geometric Brownian Motion:

$$dS_t = rS_t dt + \sigma S_t dW_t^S.$$

The payoff is:

$$f_\theta(\{S_t\}_t) = \left(\frac{1}{n_D}\sum_{i=1}^{n_D} S_{t_i} - K\right)^+,$$

with n_D the sampling frequency, K the option's strike price and the exponent "+" indicating the positive part of the expression in brackets. Furthermore, the option's maturity is $T = 1$ and simulations uses $\Delta t = \frac{1}{n_D} = \frac{1}{252}$.
In this framework, there exists a well-known control variate, the geometric Asian option, hence the PEMC approach is compared to both the MC and CV techniques. The CV

estimator is computed as:

$$\text{CV} = \frac{1}{n}\sum_{i=1}^{n}\left(f_\theta(\{S_t^{(i)}\}_t) - P_G(\theta, \{S_t^{(i)}\}_t)\right) + P_G^{exact}(\theta),$$
$$P_G(\theta, \{S_t\}_t) = \left(\left(\prod_{i=1}^{n_D} S_{t_i}\right)^{\frac{1}{n_D}} - K\right)^+.$$

The derivation of $P_G^{exact}(\theta)$, the closed-form expected payoff of a geometric Asian option with discrete monitoring under the Black&Scholes model, is in Appendix B. Moreover, two versions of the PEMC estimator are taken into consideration: the first relies on a 1-dimensional X, computed as the sum of 252 Brownian increments, while the second one considers a vector X composed of 14 components, calculated as:

$$[X]_i = \sum_{j=18(i-1)+1}^{18i} \Delta W_j^S, \quad \text{for i} = \{1, ..., 14\}.$$

The parameter space Θ and the evaluation parameters are summarized in Table 2.1.

Mode	r	S_0	σ	K
Training	$[0.01, 0.03]$	$[80, 120]$	$[0.05, 0.25]$	$[90, 110]$
Evaluation	0.02	100	0.2	100

Table 2.1: Parameter setup for PEMC variance reduction experiment

The neural network is trained on $N_{train} = 1.28 \cdot 10^6$ samples. It takes as input a scaled version of $\theta = (r, S_0, \sigma, K)$ and X: the former is scaled using min-max scaling, while the latter employs standard scaling. The choice of these methods is due to some considerations: the fact that the elements of θ are uniformly sampled from intervals makes min-max scaling the right choice, since it preserves the exact shape of the original distribution while rescaling the data to $[0, 1]$; on the other hand, the absence of bounds in the distribution of X and, more importantly, the Gaussianity of the data leads to the use of standardization, that preserves the normal distribution.
The model architecture is described in Figure 2.1. In particular, the neural network is composed of two branches, the "θ Network Branch" and the "X Network Branch". The former processes the input through 2 fully connected layers with batch normalization and ReLU activation and a dropout layer with a dropout rate 0.5; the latter uses a simple structure of 2 fully connected layers, each followed by a dropout layer with a dropout rate of 0.5. The outputs of both branches are concatenated and fed into the "Combined

Network", where they pass through a couple of fully connected layers with batch normalization, ReLU activation and dropout, and a final fully connected layer produces the final prediction. A skip connection (see Section 1.2) is used in the "Combined Network" to prevent the vanishing gradient problem.

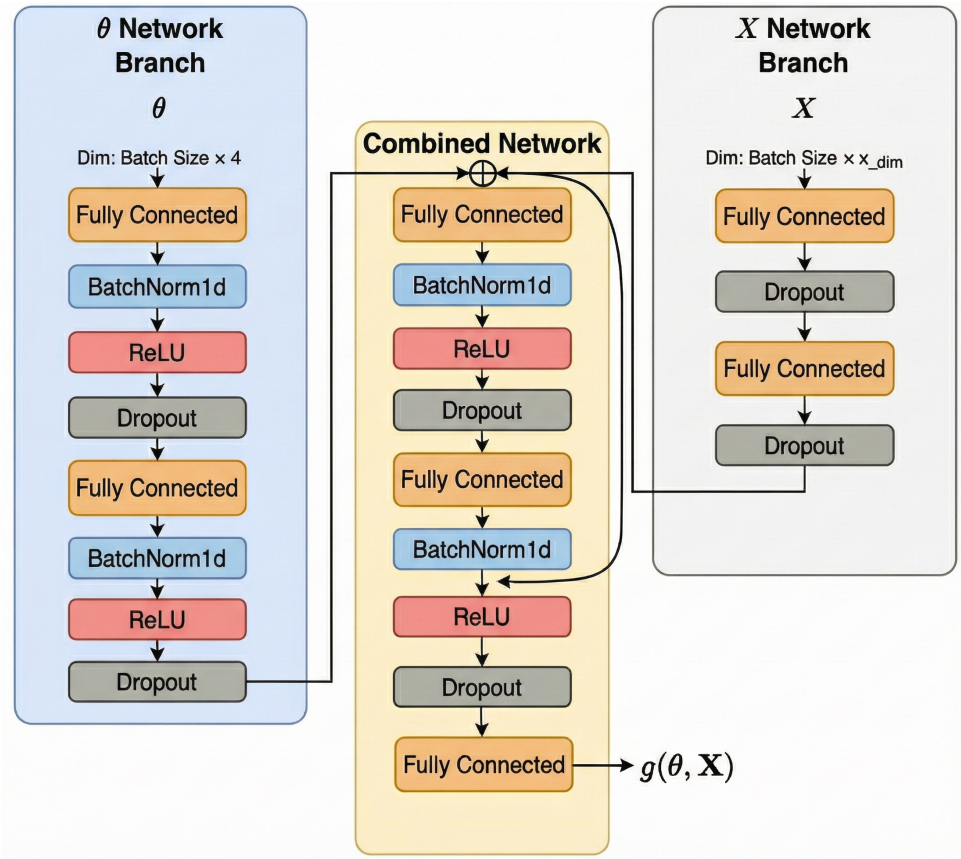

Figure 2.1: Neural network architecture for PEMC variance reduction experiment

The model is trained using Adam optimizer with a learning rate of $1 \cdot 10^{-3}$ and the architectural parameters are specified in Table 2.2.

θ Network Branch	X Network Branch
Output dim: 10	Hidden dim: 32

Table 2.2: Neural network parameters for PEMC variance reduction experiment

2.3.2. Boost PEMC Variance Reduction

In practical cases where a CV estimator exists, a particular PEMC estimator called "Boost PEMC estimator" can be built on top of it to obtain a further variance reduction. As

a demonstration of the effectiveness of this technique, the previous real-world scenario is used, i.e. arithmetic Asian option under the GBM model, with the same parameters. In particular, the label is changed from $f_\theta(\{S_t\}_t)$ to $f_\theta(\{S_t\}_t) - P_G(\theta, \{S_t\}_t) + P_G^{exact}(\theta)$, in such a way that the model learns:

$$\mathbb{E}^{\mathbb{Q}}\big[f_\theta(\{S_t\}_t) - P_G(\theta, \{S_t\}_t) + P_G^{exact}(\theta)|\theta, X(\theta)\big] = \mathbb{E}^{\mathbb{Q}}\big[f_\theta(\{S_t\}_t) - P_G(\theta, \{S_t\}_t)|\theta, X(\theta)\big] + P_G^{exact}(\theta).$$

Thanks to this formulation, it is possible to train the model on $f_\theta(\{S_t\}_t) - P_G(\theta, \{S_t\}_t)$ and use the estimator:

$$\text{Boost PEMC}(\hat{\theta}) \coloneqq \frac{1}{n}\sum_{i=1}^{n}(f_{\hat{\theta}}(\{S_t^{(i)}\}_t) - P_G(\hat{\theta}, \{S_t^{(i)}\}_t) - g(\hat{\theta}, X_i(\hat{\theta}))) + \frac{1}{N}\sum_{j=1}^{N} g(\hat{\theta}, \tilde{X}_j(\hat{\theta})) + P_G^{exact}(\hat{\theta}).$$

The model used for this application is the one in Figure 2.1 with the parameters contained in Table 2.2. As in the previous application, θ and X are scaled before being passed to the neural network. However, in this experiment, only the estimator with dim(X)= 1 is computed.

2.3.3. Variance Swaps Under Heston Model

A more complex real-world scenario is addressed with the pricing of a variance swap under the Heston model. Indeed, in this case, both the underlying asset's price and the instantaneous volatility are stochastic processes, whose dynamics are described as:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^S, \tag{2.5}$$
$$dv_t = \kappa(\eta - v_t)dt + \delta\sqrt{v_t} dW_t^v, \tag{2.6}$$

where $\langle dW_t^S, dW_t^v\rangle = \rho dt$ and r, κ, η and δ are constant parameters.
The dynamics are simulated by the Euler scheme with full truncation as:

$$v_{t+\Delta t} = v_t + \kappa(\eta - v_t^+)\Delta t + \delta\sqrt{v_t^+}\Delta W_t^v \tag{2.7}$$
$$S_{t+\Delta t} = S_t \exp\left(\left(r - \frac{1}{2}v_t^+\right)\Delta t + \sqrt{v_t^+}\Delta W_t^S\right) \tag{2.8}$$

with $v_t^+ = \max(0, v_t)$, $\Delta W_t^S = \sqrt{\Delta t}\cdot Z_1$ and $\Delta W_t^v = \sqrt{\Delta t}\cdot(\rho Z_1 + \sqrt{1-\rho^2}Z_2)$ where Z_1 and Z_2 are two independent standard Gaussian random variables.
Since the payoff of a vanilla variance swap [33], considering a notional amount of 100 and

a strike $K = 0$ is:

$$f_\theta(\{S_t\}_t) = 100 \cdot \frac{252 \cdot \sum_{t=1}^{T/\Delta t} \left(\ln(S_t/S_{t-1})\right)^2}{T/\Delta t},$$

during the simulation it is enough to compute `log_return` $= \left(r - \frac{1}{2}v_t^+\right)\Delta t + \sqrt{v_t^+}\Delta W_t^S$ instead of $S_{t+\Delta t}$ in (2.8). In the payoff formula, $T = 1$ and $\Delta t = 1/252$. The parameter space is in Table 2.3, where $[\cdot,\cdot]^2$ means that the parameter is uniformly sampled by the interval and then squared.

Mode	S_0	r	v_0	κ	η	δ	ρ
Training	$[50, 150]$	$[0.01, 0.05]$	$[0.1, 0.375]^2$	$[1.5, 4.5]$	$[0.1, 0.3]^2$	$[0.1, 1.0]$	$[-0.9, -0.2]$
Evaluation	100	0.02	0.25^2	3.0	0.2^2	0.6	-0.4

Table 2.3: Parameter setup Heston model

The prediction model used in this case is composed of two consecutive Multi-Layer Perceptron (MLP) blocks connected through skip connections (see Section 1.2), meaning that two residual functions are considered, each containing all the layers of an MLP block. The model receives as input:

$$(\underbrace{r, \kappa, \eta, \delta}_{\theta}, \underbrace{W_T^S, W_T^v}_{X}),$$

with $W_T^* := \sum_{i=1}^{T/\Delta t} \Delta W_{i\Delta t}^*$. The output of the first MLP block is fed into the second MLP block, producing the final prediction. The model is trained on $N_{train} = 3 \cdot 10^6$ samples. The complete architecture specification is provided in Table 2.4.

1st MLP block	**2nd MLP block**
Hidden dim: 512	Hidden dim: 256
Output dim: 256	Output dim: 256

Table 2.4: Neural network setup Heston model

2.3.4. Variance Swaps Under Stochastic Local Volatility Model

Dealing with a Stochastic Local Volatility (SLV) model adds a new layer of complexity to the problem, given by the presence of a full volatility surface discretized on a dense 2D grid. Indeed, under the SLV model, the dynamics of the asset price $\{S_t\}_t$ are described

by the following SDEs:

$$dS_t = rS_t dt + \sigma(S_t, t)e^{v_t} S_t dW_t^S, \tag{2.9}$$

$$dv_t = \kappa(\eta_t - v_t)dt + \delta dW_t^v, \tag{2.10}$$

with $\eta_t := -\frac{\delta^2}{2\kappa}(1 + e^{-2\kappa t})$, $\langle dW_t^S, dW_t^v \rangle = \rho dt$ and r, κ, δ, ρ constant parameters. Furthermore, $\sigma(\cdot,\cdot) : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}^+$ is a 2D function representing the local volatility surface and e^{v_t} is a stochastic multiplier such that $e^{v_0} = 1$ and $\mathbb{E}^{\mathbb{Q}}[e^{2v_t}] = 1$. In order to make it possible to store a different surface σ for each data point in the training dataset, it is stored as a discrete grid indexed by asset price and time. In particular, each point of the grid has a value of local volatility computed as in [5]:

$$\sigma_{base}^2(x,t) = \frac{\sum_{i=0}^{2} p_i \tau_i e^{-x^2/(2t\tau_i^2) - t\tau_i^2/8}}{\sum_{i=0}^{2} (p_i/\tau_i) e^{-x^2/(2t\tau_i^2) - t\tau_i^2/8}}, \tag{2.11}$$

where $x := \ln(S_t/S_0)$, t is the time to maturity, $p_0 = 0.3$, $p_1 = 0.5$, $p_2 = 0.2$ $\tau_0 = 0.4$, $\tau_1 = 0.3$, $\tau_2 = 0.6$ as in Figure 2 in [5]. On top of $\sigma_{base}(x,t)$ it is added a $N(0,\epsilon^2)$ noise independently to each point of the grid, obtaining $\sigma_{grid}(x,t)$.
The simulation of (2.9)-(2.10) exploits the Euler scheme:

$$S_{t+\Delta t} = S_t \exp\left((r - \frac{1}{2}\tilde{v}_t^2)\Delta t + \tilde{v}_t \Delta W_t^S\right), \quad \text{where } \tilde{v}_t = \hat{\sigma}(S_t, t)e^{v_t}, \tag{2.12}$$

$$v_{t+\Delta t} = v_t + \kappa(\eta_t - v_t)\Delta t + \delta \Delta W_t^v. \tag{2.13}$$

where the same observations as in the previous real-world case about the Brownian increments generation and the log-returns simulation can be done. Different from the Heston case, the notional amount in the computation of the payoff is assumed to be 1. Finally, $\hat{\sigma}$ is obtained by interpolation of the grid σ_{grid} [6].
The input to the model is defined as:

$$\text{feature}_i = \{\{\sigma_{grid}^2(x,t)\}_{x,t}, S_0, \kappa, \delta, \rho, r, \epsilon, v_0, (W_T^S, W_T^v)\}.$$

The parameter space Θ is in the following table:

Mode	S_0	κ	δ	ρ	r	ϵ	v_0
Training	$[50, 150]$	$[1.5, 4.5]$	$[0.1, 1.0]$	$[-0.9, -0.2]$	0.02	0.02	0
Evaluation	100	3.0	0.5	-0.5	0.02	0	0

Table 2.5: Parameter setup SLV model

Moreover, the training dataset is set to have $3 \cdot 10^6$ data points, $T = 1$ and $\Delta t = 1/252$. A two-branch neural network architecture is used, as illustrated in Figure 2.2.

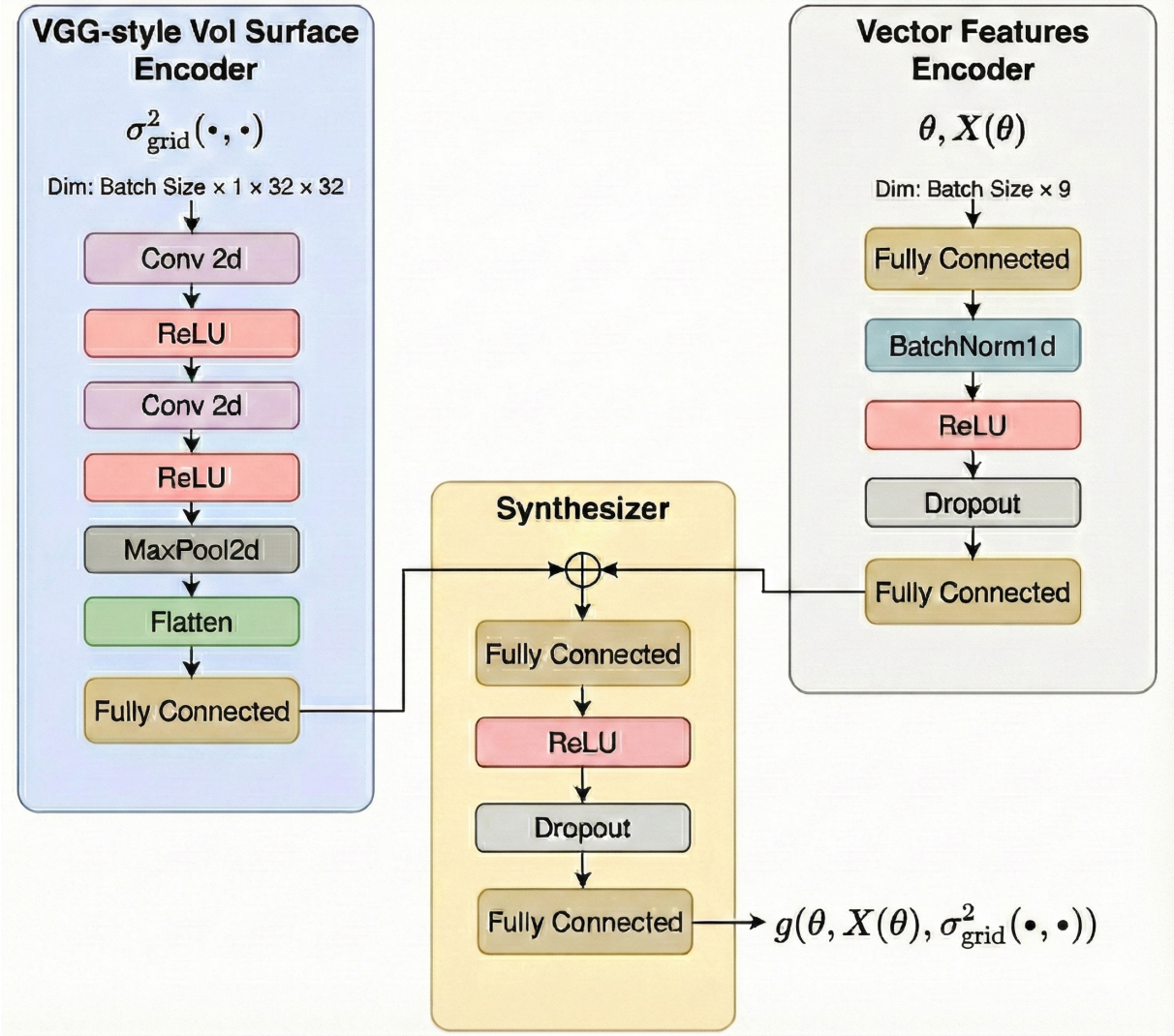

Figure 2.2: Neural network architecture for the SLV model.

The first branch takes as input σ^2_{grid}, which passes through a VGG-style [32] convolutional neural network (CNN) architecture composed of two 2D convolutional layers with ReLU activations, followed by a MaxPool2D operation and a flattening that leads to a fully connected layer. The second branch processes θ and X through a fully connected layer with batch normalization and ReLU activation, followed by a dropout layer and a final fully connected layer. The outputs of the two branches are concatenated and fed into a "Synthesizer" module, which combines them through a fully connected layer with ReLU activation, followed by a dropout layer and a final fully connected layer that produces the output. Further architectural details are in Table 2.6.

VGG-style Vol Surface Encoder	Vector Features Encoder
Conv2D: Kernel size= 3, stride= 1, padding= 1	Hidden dimension= 512
MaxPool2D: Kernel size= 2, stride= 2, padding= 0	Output dimension= 128

Table 2.6: Neural network setup SLV model

In order to have a faster training and evaluation procedure, some optimizations are used in the code. For example, the `SLVContext` class is created to initialize all the variables that do not depend on θ and to compute σ_{base}. In this way, when the simulation function is called many times (e.g., during data generation), this class can be called just once and then passed as an argument to the simulation function, saving computational time due to the fact that the grid and other parameters are pre-computed. Furthermore, the `only_inputs` and the `train` variables are used as a flag to adapt to the different utilization of the simulation function: the former is set to `True` in case it is not necessary to simulate the payoff function, returning just σ^2_{grid} and X (useful during the PEMC estimator computation) and saving a considerable amount of time that the interpolation of the volatility surface would have consumed; the latter has a subtle use that enhance reproducibility during the evaluation procedure and helps in saving memory. In particular, the presence of `train` is linked to the fact that ϵ is set to 0 during evaluation, making $\sigma_{grid} = \sigma_{base}$. For this reason, during training (`train` = `True`) the Gaussian noise to be added to σ_{base} should be computed:

```
1    if train:
2        noise = torch.randn((N, 1, ctx.steps_x, ctx.steps_t),
             device=ctx.device)
3        sigma_grid = sigma_base_expanded + epsilon_vec * noise
4    else:
5        sigma_grid = sigma_base_expanded.expand(N, -1, -1, -1)
```

Listing 2.6: Use of "train" flag inside the simulation function

but during evaluation (`train` = `False`) it should not be simulated and then multiplied by 0, since calling `torch.randn` advances the random seed, introducing small differences between each experiment. Moreover, since the noise should be multiplied by 0, it is useless to store in memory the `noise` tensor.

The simulation function exploits a further optimization: a slight variation of the "log-sum-exp" trick illustrated in Section 1.3. In particular, it is applied to the computation of (2.11):

```
    # ... calculate 'exponent' ...

    max_exp = torch.max(exponent, dim=2, keepdim=True)[0]
    exp_rel = torch.exp(exponent - max_exp)

    numerator_rel = torch.sum(p_expanded * tau_expanded * exp_rel,
        dim=2)
    denominator_rel = torch.sum(exp_rel * p_expanded /
       tau_expanded, dim=2)

    self.sigma_base = torch.sqrt(numerator_rel / denominator_rel)
```

Listing 2.7: Log-sum-exp trick

Here, the "log-sum-exp" trick is slightly modified with respect to its classical form since there is no logarithm to compute, but the core logic to prevent underflow is the same as in the original trick: subtracting the maximum value of the exponent from each of the exponents. In this case, since both the numerator and the denominator use this trick, the computation becomes:

$$\sigma^2_{base}(x,t) = \frac{e^{\texttt{max\_exp}} \sum_{i=0}^{2} p_i \tau_i e^{\texttt{exponent-max\_exp}}}{e^{\texttt{max\_exp}} \sum_{i=0}^{2} (p_i/\tau_i) e^{\texttt{exponent-max\_exp}}},$$

leading to a simplification of $e^{\texttt{max\_exp}}$ that leads to the formula with `exp_rel` used in the code. Without this trick, both `numerator_rel` and `denominator_rel` would be equal to 0 due to the underflow caused by a highly negative exponent in the exponential term when the maturity T of the option is approached.
Finally, the training and evaluation speed is enhanced by compiling both the model and the simulation functions:

```
    self.model = torch.compile(self.model)
    ...
    # Compile simulation functions
    slv_train_fn = partial(simulate_slv, train=True, only_inputs=
       False)
    opt_slv_train = torch.compile(slv_train_fn)
    slv_eval_fn = partial(simulate_slv, train=False, only_inputs=
       False)
    opt_slv_eval = torch.compile(slv_eval_fn)
    slv_inputs_fn = partial(simulate_slv, train=False, only_inputs
       =True)
```

```
9     opt_slv_inputs = torch.compile(slv_inputs_fn)
```

Listing 2.8: Compilation of model and simulation function

and by using `GradScaler` and `autocast` inside the training and validation functions.

2.3.5. Swaptions Under Heath-Jarrow-Morton Model

The most complex application of the PEMC approach taken into account is the pricing of a swaption under the one-factor Heath-Jarrow-Morton (HJM) model [13]. A swaption is a financial derivative that gives the holder the right to enter into an interest rate swap at a future date. In this example, the start date of the swap and the maturity of the swaption coincide at time $t'_0 = 5$, hence the payoff of the option is:

$$\max(0, V_{t'_0}),$$

with

$$V_{t'_0} = C\Big((1+R)\sum_{i=1}^{n_p} B(t'_0, t'_i)\Delta t' + B(t'_0, t'_{n_p}) - 1\Big), \tag{2.14}$$

with $C = 100$ the notional amount, $n_p = 20$ the number of swap payment periods, each of length $\Delta t' = 1$, starting at time t'_0 and ending at time t'_{n_p}, $t'_i = t'_0 + i\Delta t'$, $R = \exp\Big(-\frac{\sum_{i=1}^{n_p} f(0,t'_i)}{T^* - t'_0}\Big)$ and $B(t'_0, t'_i)$ the discount factor from t'_0 to t'_i.
The price $B(t,T)$ of a zero-coupon bond maturing at T computed at time t is:

$$B(t,T) = \exp\Big(-\int_t^T f(t,u)du\Big). \tag{2.15}$$

Under the HJM model, the dynamics of the forward rate curve are directly modeled as:

$$df(t.T) = \mu(t,T)dt + \sigma(t,T)^T dW(t), \tag{2.16}$$

with $W(t)$ a Brownian motion and $\mu(t,T)$ and $\sigma(t,T)$ the time-dependent drift and volatility function, respectively. In the HJM framework, the latter and the former are linked by:

$$\mu(t,T) = \sigma(t,T)^T \int_t^T \sigma(t,u)du. \tag{2.17}$$

This is the no-arbitrage condition of the framework. Thus, the model is specified by defining $f(0,T)$ and the entire volatility structure $\sigma(t,T)$. In this example, an exponential

decay model inspired by [10] is exploited for the latter:

$$\sigma_{base}(t,T) = \sigma_0 \exp\left(-\alpha_\sigma(T-t)\right), \tag{2.18}$$

while the former is set to:

$$f_{base}(0,T) = f_0 + c_f(1-\exp(-\alpha_f T)), \tag{2.19}$$

where σ_0, α_σ, f_0, c_f and α_f are part of θ.
The simulation of (2.15)-(2.19) requires numerical discretization. Following the method used in [10], both the time and maturity axis are discretized, in particular $T = 0, \Delta t, 2\Delta t, ..., T^*$ while $t = 0, \Delta t, ..., t_0' - \Delta t$, with $\Delta t = 1/52$ and $T^* = 25$. This choice differs from the one adopted in [10], where the same discretization grid is used for both t and T, aiming to save as much memory and computational time as possible, since the training and evaluation procedures are extremely resource-consuming in this example.
Given the time grids, the discretization of (2.18) and (2.19) is straightforward. During the training procedure, a noise $N(0, \left(\frac{\sigma_0}{2(T+5)}\right)^2)$ is added on top of (2.18), while $N(0, \left(\frac{1}{100(T+5)}\right)^2)$ is summed to (2.19). In the evaluation phase, no noise is added. Then, the no-arbitrage condition (2.17) is discretized using the trapezoidal rule, while the diffusion term is evaluated through a standard Euler scheme. Finally, it is possible to compute:

$$\hat{f}(t_i,T_j) = \hat{f}(t_{i-1},T_j) + \mu(t_{i-1},T_j)\Delta t + \sigma(t_{i-1},T_j)\sqrt{\Delta t}N(0,1), \quad \begin{cases} t_i \in \{0, \Delta t, ..., t_0' - \Delta t\} \\ T_j \in \{0, \Delta t, ..., T^*\} \end{cases}$$

Given the set of forward rate values, it is possible to compute the discretized form of (2.15):

$$\hat{B}(t_i,T_j) = \exp\left(-\sum_{l=i}^{j-1} \hat{f}(t_i,T_l)\Delta t \right). \tag{2.20}$$

with t_i and T_j as above. Now, all the data is available to compute $V_{t_0'}$ as in (2.14). Remarkably, $V_{t_0'}$ is divided by C when used for training and validation in a sort of normalization procedure that proved effective to enhance the precision of the training and validation procedure. As a consequence, the output of the model is multiplied by C before the computation of the PEMC estimator.
After dealing with the computation of the label of the training and validation datasets, the focus moves towards the feature vector (θ, X). In this application, X is a two-dimensional vector (X_1, X_2), built by splitting the time horizon t_0' into two parts based on the number of discretization steps $N = \frac{t_0'}{\Delta t}$. Let $M = N//2$ ("//" is the integer division in Python)

be the midpoint index, then X_1 is the sum of Brownian increments corresponding to the first half of the steps (i.e. from index 0 to $M-1$), whereas X_2 represents the sum of Brownian increments of the second half (i.e. from index M to $N-1$). The feature vector can be written as:

$$\text{feature}_i = \{\{\sigma(t_i, T_j)\}_{i,j}, \{f(0, T_j)\}_j, \sigma_0, \alpha_\sigma, f_0, c_f, \alpha_f, (X_1, X_2)\}. \tag{2.21}$$

The training dataset has a dimension $N_{train} = 3 \cdot 10^6$ and the parameter specifications are in Table 2.7.

Mode	σ_0	α_σ	f_0	c_f	α_f
Training	$[0.01, 0.03]$	$[0.001, 0.9]$	$[0.01, 0.03]$	$[0.01, 0.05]$	$[0.001, 0.9]$
Evaluation	0.015	0.45	0.02	0.03	0.5

Table 2.7: Parameter setup HJM model

The model used to process the input (2.21) is a three-branch neural network architecture. represented in Figure 2.3. The first branch, named "2D Function Encoder", receives as input the 2D volatility grid $\{\{\sigma(t_i, T_j)\}_{i,j}$ and passes it through a CNN architecture with two 2D convolutional layers with batch normalization and ReLU activation. The branch ends with an average pooling operation and a flattening that produces an embedding. The second branch is the so-called "1D Function Encoder", which processes the grid $\{f(0, T_j)\}_j$ through the 1D version of the CNN architecture used in the first branch. The third branch handles $(\theta, X(\theta))$ through two fully connected layers with batch normalization and ReLU activation. In addition, a skip connection (see Section 1.2) with a residual function that contains just one layer is implemented in this branch, as illustrated in Figure 2.3. Finally, the three embeddings produced by the branches are concatenated to $(\theta, X(\theta))$ and fed into a "Synthesizer" module, which is composed of two fully connected layers with batch normalization and ReLU activation, followed by a final fully connected layer that produces the prediction. Moreover, skip connections with a residual function that contains just one layer are also in this module. The architectural parameters are in Table 2.8.

Similarly to the previous applications, some optimizations are done in order to save computational time and memory. In particular, a class `HJMContext` is defined and used as `SLVContext` in the previous real-world application. Flag variables are exploited in order to customize the simulation function execution and both the model and the simulation function are compiled. Finally, `GradScaler` and `autocast` are employed to optimize the training procedure.

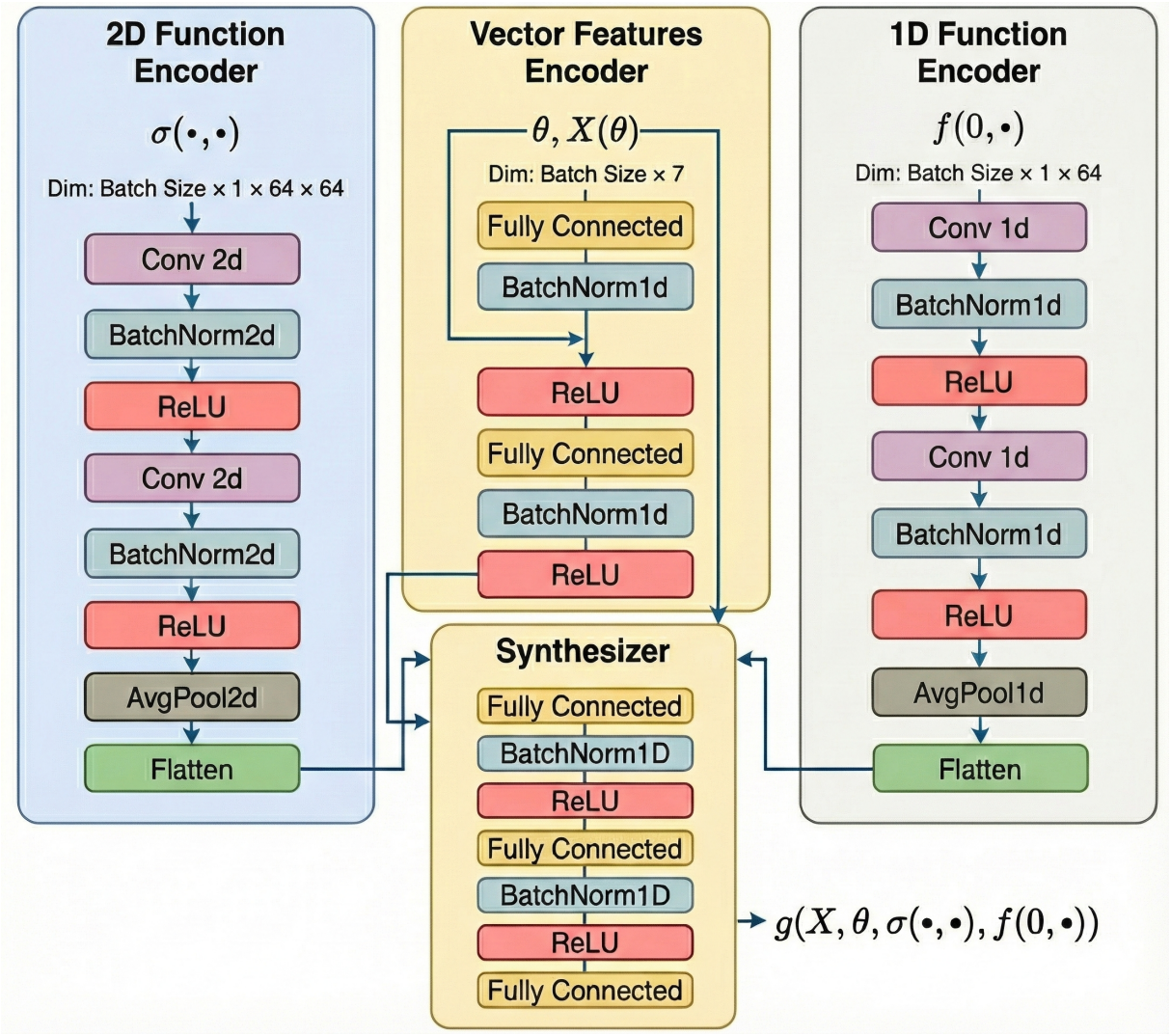

Figure 2.3: Neural network architecture for the HJM model

2D Function Encoder	1D Function Encoder	Vector Features Encoder
Kernel size: (1, 3)	Kernel size: 10	
Stride: (1, 3)	Stride: 3	
Padding: 0	Padding: 0	Hidden dim: 512
AvgPool2d kernel size: (2, 2)	AvgPool1d kernel size: 2	Output dim: 128
AvgPool2d stride: (2, 2)	AvgPool1d stride: 2	
AvgPool2d padding: 0	AvgPool1d padding: 0	

Table 2.8: Neural network setup HJM model

3 | Results

The numerical results of the examples described in Section 2.3 are reported in the tables below. Given that all estimators are unbiased, the analysis focuses on variance-related metrics.

Each of the sections has the following structure: an initial paragraph that specifies the parameters for the computation of the root mean squared error, two tables with the values of RMSE related to the estimators and the percentage reduction that PEMC achieves compared to the baseline model, and a final comment on the results. The code is available in this GitHub repository.

3.1. PEMC Variance Reduction Results

The first real-world pricing scenario uses 300 experiments to compute the RMSE and considers a sample size $n \in \{1000, 4000, 9000\}$. The ground truth is computed on $2 \cdot 10^9$ MC simulations. The resulting metrics are in the Table 3.1.

Method	**n = 1000**	**n = 4000**	**n = 9000**
Monte Carlo (MC)	0.264495	0.129612	0.080627
PEMC (dim(X) = 1)	0.149688	0.074927	0.048125
PEMC (dim(X) = 14)	0.081530	0.043113	0.027067
Geometric CV	0.010705	0.005068	0.003387

Table 3.1: RMSE for PEMC variance reduction example

Method	**n = 1000**	**n = 4000**	**n = 9000**
PEMC (dim(X) = 1)	43.406%	42.192%	40.311%
PEMC (dim(X) = 14)	69.175%	66.737%	66.429%

Table 3.2: RMSE reduction of the PEMC estimators with respect to MC in the PEMC variance reduction example

As it is possible to see in Table 3.2, both the PEMC estimator with dim(X) = 1 and the one with dim(X) = 14 achieve a superior performance in terms of RMSE compared to the MC estimator. In particular, the former reduces MC's RMSE by $40 - 43\%$, while the latter attains an impressive $66 - 69\%$ reduction. The estimator with $\dim(X) = 14$ outperforms its 1D counterpart due to the path-dependent nature of Asian option payoffs. While a single-dimensional vector captures limited path information, a 14-dimensional representation is more granular. This enables the neural network to better capture the temporal trajectory, resulting in a more precise approximation of the payoff. However, the CV estimator is the best-performing one, with an RMSE that is considerably smaller than the competitors. This is an expected result, since the CV estimator's construction is based on mathematical rules, which offer more guarantees than a neural network approximation.

3.2. Boost PEMC Variance Reduction Results

The second application of the PEMC approach is a slight variation of the first one, hence it employs the same number of experiments, the same sample size for the computation of the ground truth value and the same set of sample sizes for the evaluation. The results are in Table 3.3.

Method	**n = 1000**	**n = 4000**	**n = 9000**
Geometric CV	0.010705	0.005068	0.003387
Boost PEMC (dim(X) = 1)	0.006087	0.003263	0.002114

Table 3.3: RMSE for Boost PEMC variance reduction example

Method	**n = 1000**	**n = 4000**	**n = 9000**
Boost PEMC (dim(X) = 1)	43.145%	35.628%	37.577%

Table 3.4: RMSE reduction of the Boost PEMC estimator with respect to MC

The use of an existing control variate as a basis is beneficial for the PEMC approach, which outperforms the CV estimator by a significant amount (i.e., $36 - 43\%$ reduction in RMSE), as shown in Table 3.4. This result is due to the fact that the neural network, unlike the standard CV estimator, can learn the complex dependencies of the residual difference between the arithmetic and geometric payoff, leading to a superior reduction. The performance of the Boost PEMC estimator with respect to the MC one is comparable to the results in Table 3.2 for the PEMC estimator with dim(X) = 1, highlighting the fact

that the PEMC approach can consistently outperform the baseline model upon which it is built (MC in the first example and CV here). These results are remarkable, since they show that the PEMC estimator is not just an alternative to the CV counterpart in case the latter does not exist, but also an upgrade in cases where the CV exists.

3.3. Variance Swaps Under Heston Model Results

In the example regarding variance swap pricing under the Heston model, the ground truth is computed by averaging $5 \cdot 10^7$ samples and the set of sample sizes used is $\{1000, 2000, 4000, 6000, 8000, 10000, 20000\}$, repeating each experiment 200 times to assess sampling variability. The corresponding RMSEs are in Table 3.5.

Method	**n = 1000**	**n = 2000**	**n = 4000**	**n = 6000**	**n = 8000**	**n = 10000**	**n = 20000**
MC	0.106386	0.068633	0.053384	0.040051	0.035921	0.034691	0.021428
PEMC	0.058983	0.046056	0.029934	0.024492	0.021839	0.020792	0.014686

Table 3.5: RMSE for variance swaps pricing under the Heston model

Method	**n = 1000**	**n = 2000**	**n = 4000**	**n = 6000**	**n = 8000**	**n = 10000**	**n = 20000**
PEMC	44.558%	32.895%	43.928%	38.848%	39.202%	40.066%	31.465%

Table 3.6: RMSE reduction for pricing variance swaps under the Heston model

Table 3.6 shows a comparable behaviour with respect to Table 3.2, with a $31 - 45\%$ reduction in root mean squared error compared to the MC estimator. Apart from a slightly larger interval of RMSE reduction ($31 - 45\%$ vs $40 - 43\%$), the fact that the performance is similar to the first example despite an increase in complexity is a sign of the versatility of the PEMC approach. Indeed, even though the Heston model adds a second stochastic process compared to the Geometric Brownian Motion case, the neural network's ability to map the non-linear relationships between the model parameters and the payoff proved effective. In addition, the RMSE reduction of the PEMC estimator is crucial in contexts like the Heston model with full truncation, where the simulations are quite costly, since it allows for reducing the number of simulated paths necessary to achieve a certain error value.

3.4. Variance Swaps Under Stochastic Local Volatility Model Results

Talking about variance swap pricing under a stochastic local volatility model, the parameters for the computation of the RMSE are the same as the Heston case. The results are in Table 3.7.

Method	**n = 1000**	**n = 2000**	**n = 4000**	**n = 6000**	**n = 8000**	**n = 10000**	**n = 20000**
MC	0.002269	0.001546	0.000989	0.000932	0.000742	0.000724	0.000477
PEMC	0.001267	0.000952	0.000667	0.000550	0.000447	0.000353	0.000303

Table 3.7: RMSE for pricing variance swaps under the stochastic local volatility model

Method	**n = 1000**	**n = 2000**	**n = 4000**	**n = 6000**	**n = 8000**	**n = 10000**	**n = 20000**
PEMC	44.164%	38.445%	32.576%	40.985%	39.790%	51.235%	36.500%

Table 3.8: RMSE reduction for pricing variance swaps under stochastic local volatility model

Despite the increased complexity of the stochastic local volatility model with respect to Heston, the PEMC estimator confirms its effectiveness, with even a slight improvement. Indeed, Table 3.8 shows that the PEMC approach produces an estimator that achieves a $33-51\%$ reduction in root mean squared error compared to MC, confirming the versatility of the method, even when dealing with a volatility surface. The result of this real-world scenario confirms the effectiveness of convolutional neural networks in handling grid-structured data [19, 32], since the model is able to learn the shape of the volatility surface even though it is perturbed by a Gaussian noise during training.

3.5. Swaptions Under Heath-Jarrow-Morton Model Results

The last and most complicated application, the pricing of a swaption under the Heath-Jarrow-Morton model, employs $5 \cdot 10^7$ MC samples for the computation of the ground truth value, while evaluating the RMSE on 200 independent experiments with sample size $n \in \{1000, 3000, 5000, 7000, 9000, 11000\}$. The obtained RMSEs are in Table 3.9.

Method	n = 1000	n = 3000	n = 5000	n = 7000	n = 9000	n = 11000
MC	0.028634	0.016532	0.013120	0.011022	0.009024	0.008439
PEMC	0.015803	0.009034	0.006583	0.005629	0.004863	0.004631

Table 3.9: RMSE for pricing swaptions under the Heath-Jarrow-Morton model

Method	n = 1000	n = 3000	n = 5000	n = 7000	n = 9000	n = 11000
PEMC	44.809%	45.352%	49.825%	48.926%	46.105%	45.123%

Table 3.10: RMSE reduction for pricing swaptions under the Heath-Jarrow-Morton model

The Heath-Jarrow-Morton framework introduces a new layer of complexity compared to the stochastic local volatility model, specifically in managing both volatility structures and forward rate curves. Despite this, the PEMC approach maintains the same level of performance, delivering a $45 - 50\%$ RMSE reduction with respect to MC, as shown in Table 3.10. This level of reduction confirms the efficiency of 1D convolutional layers in treating sequential data [19], while leveraging the proven effectiveness of 2D convolutional layers for grid-structured data.
The results in this section are additional confirmation that, as highlighted in [20], "The consistent effectiveness of PEMC across various financial instruments and modeling frameworks further underscores its versatility as a robust variance reduction technique."

4 | Extensions

One of the biggest limitations of the PEMC method is the theoretical requirement for a large number of training samples to guarantee variance reduction compared to the MC estimator, which leads to long training procedures, especially in complex pricing scenarios. In particular, Lemma 5 in [20] establishes that, in the ideal situation where the machine learning predictor g is exactly equal to $\mathbb{E}^{\mathbb{Q}}[f_\theta(Y)|\theta, X]$:

$$\frac{Var(\text{PEMC})}{Var(\text{MC})} \approx r(\rho, c).$$

Namely, in the ideal case, the variance ratio between PEMC and MC can be approximated by a function of both $\rho = corr(f, g)$ and the relative cost $c = \frac{c_g}{c_f}$ of evaluating g versus generating full samples of f. The term "cost" is intentionally left ambiguous, since it could refer to both the sample size and the wall-clock time required to generate and evaluate the sample. As shown in Figure 1 in [20], variance reduction is achieved when ρ is sufficiently large and c is sufficiently small.
In practice, however, g is just an approximation of $\mathbb{E}^{\mathbb{Q}}[f_\theta(Y)|\theta, X]$ and:

$$\epsilon_{total} \leq 2\epsilon_a^{\mathcal{G}} + 2\epsilon_e^{N_{train}},$$

with ϵ_{total} the total error between g and the ideal predictor, $\epsilon_a^{\mathcal{G}}$ the approximation error and $\epsilon_e^{N_{train}}$ the statistical error. While $\epsilon_a^{\mathcal{G}}$, which is the bias introduced by the choice of the neural network family, can be minimized by using a sufficiently complex architecture (Lemma 7 in [20]), the statistical error depends on N_{train} and the problem's complexity. According to Lemma 8 in [20], there exists a sufficiently large N_{train} to reduce this error below an arbitrary low threshold; specifically, standard generalization bounds indicate that this error typically decreases at a rate of $O(1/\sqrt{N_{train}})$. The minimization of ϵ_{total} is fundamental, as shown in the proof of Lemma 6 in [20], to make the variance ratio between PEMC and MC converge to the optimal value $r(\rho, c)$.
A solution to maintain the same level of variance reduction while decreasing N_{train} is to use more significant features. As illustrated in Section 1.5, by combining PCA and differential PCA as a preprocessing tool, it is possible to train the neural network on

lower-dimensional and noise-free data. This reduction in the problem's complexity lowers the statistical error for a fixed N_{train} [20], preserving the model's accuracy and the resulting variance reduction even with a smaller training set. The advantages are on two levels: at the lower level, training is less time-consuming, while at the higher level, the choice of X is no longer manual. The consequence of the former is that the model could be updated more frequently, hence with narrower parameter intervals, making it more adapted to the evolving market conditions, while the latter means that the selection of X is done in a more meaningful way.
The experimental results in the following sections confirm the theoretical intuition reported above. In particular, the preprocessing mechanism is implemented in three of the five real-world scenarios described in Section 2.2, with brief descriptions of the main code modifications and numerical results. Minor changes to the code described in Section 2.2 are made in order to achieve reproducibility of the results across the different implementations referring to the same real-world scenario. The code is available in this GitHub repository.

4.1. Differential PEMC Variance Reduction

The pricing of an arithmetic Asian call option under the Black&Scholes model is the first application case of the preprocessing pipeline. In particular, a single value is compared to the two estimators (one with dim(X) = 1, the other with dim(X) = 14) described in Subsection 2.3.1. Two versions of the preprocessing procedure described in [15] are employed: the first ("Split" preprocessing) applies the method to θ and the Brownian increments separately, feeding the architecture in Figure 2.1 with the resulting features; the second one ("Full" preprocessing) concatenates θ and the Brownian increments and applies the data preparation algorithm to the concatenated matrix, feeding the "Combined Network" in Figure 2.1 with the transformed features.

4.1.1. "Split" preprocessing

The main innovation in the code of both versions of the preprocessing procedure is the `DiffPCA` class, which follows exactly the procedure at pages $32 - 34$ in [15]. The transformation matrices are fitted once at the beginning of the code on a simulated dataset with $1 \cdot 10^4$ samples using `setup_global_pca`:

```
def setup_global_pca(...):
    theta = torch.zeros((N_calibration, len(intervals)), device=
       device)
```

```
    for i, (low, high) in enumerate(intervals):
        theta[:, i].uniform_(low, high)

    W_dt = torch.normal(0.0, float(np.sqrt(dt)), size=(
       N_calibration, sampling_freq), device=device)

    theta.requires_grad_(True)
    W_dt.requires_grad_(True)

    payoff = simulate_arithmetic_asian_option_payoff(...)

    grads_raw = torch.autograd.grad(outputs=payoff, inputs=[theta,
        W_dt], grad_outputs=torch.ones_like(payoff))
    grads_theta = grads_raw[0]
    grads_W_dt = grads_raw[1]

    transformer = DiffPCA(n_components_pca=1-1e-10,
       n_components_diff_pca=1-1e-3, device=device)
    transformer.fit(theta.detach(), W_dt.detach(), payoff.detach()
       , grads_theta, grads_W_dt, intervals)

    return transformer
```

Listing 4.1: `setup_global_pca` for "split" preprocessing in the PEMC variance reduction example

In the code above, the `requires_grad_(True)` method is crucial as it enables gradient tracking on `theta` and `W_dt` (which are leaf tensors of the computational graph), allowing the computation of partial derivatives with respect to them. The gradients are computed using `torch.autograd.grad`, where the argument `grad_outputs` is set to a tensor of ones because the computational graph's root is `payoff` and the gradients to compute are the payoff's ones (`outputs=payoff`). Moreover, the `detach()` method of the first three arguments of `transformer.fit` is necessary to avoid useless memory consumption, since it excludes `theta`, `W_dt` and `payoff` from the computational graph after the gradients' computation.
Once the transformation matrices are fitted, they are stored in the `transformer` object, hence it is enough to call its `transform` method to preprocess `theta` and `W_dt` and feed the neural network with the transformed features.
The benefits of performing differential PCA on top of PCA are significant: 2 components

explain at least 99.9% of the market parameters' relevance and at least the same amount of the Brownian increments' relevance is explained by 1 component in the best-performing version of the data preparation method. Thanks to this optimization, similar results to PEMC (dim(X) = 14) in Table 3.2 are obtained with a training dataset of just 128 samples, instead of the $1.28 \cdot 10^6$ of the original problem. The results are in Tables 4.1 and 4.2.

Method	**n = 1000**	**n = 4000**	**n = 9000**
Monte Carlo (MC)	0.264495	0.129612	0.080627
Differential PEMC ("split")	0.076544	0.039799	0.027417
Geometric CV	0.010705	0.005068	0.003387

Table 4.1: RMSE for Differential PEMC variance reduction example with "split" preprocessing

Method	**n = 1000**	**n = 4000**	**n = 9000**
Differential PEMC ("split")	71.060%	69.294%	65.995%

Table 4.2: RMSE reduction of the Differential PEMC estimators with respect to MC, in case of "split" preprocessing in the PEMC variance reduction example

As a further comparison, Tables 4.3 and 4.4 report the values of RMSE and variance reduction compared to the MC estimator for the original problem, considering $N_{train} = 128$.

Method	**n = 1000**	**n = 4000**	**n = 9000**
Monte Carlo (MC)	0.264495	0.129612	0.080627
PEMC (dim(X) = 1)	0.155030	0.080227	0.051291
PEMC (dim(X) = 14)	0.148931	0.074485	0.047779
Geometric CV	0.010705	0.005068	0.003387

Table 4.3: RMSE for PEMC variance reduction example with $N_{train} = 128$

Method	**n = 1000**	**n = 4000**	**n = 9000**
PEMC (dim(X) = 1)	41.386%	38.102%	36.386%
PEMC (dim(X) = 14)	43.693%	42.533%	40.741%

Table 4.4: RMSE reduction of the PEMC estimators with respect to MC with $N_{train} = 128$ in the PEMC variance reduction example

Interestingly, the RMSE of PEMC ($\dim(X) = 1$) is not significantly affected by the reduction in N_{train}, whereas the performance of the estimator with $\dim(X) = 14$ gets worse compared to the results in Table 3.2. This is likely due to the fact that 128 samples are sufficient for the model architecture to learn the simple relationship in the 1-dimensional case, but represent an insufficient amount to map the dependencies of the 14-dimensional case, given the noise introduced by the high statistical error. In addition, the fact that with just 128 training samples the PEMC estimator delivers $36 - 44\%$ RMSE reduction compared to MC shows how powerful the method is in rather simple pricing examples.
Moreover, the comparison between the "split" preprocessing results and the standard PEMC confirms the effectiveness of the preliminary step in condensing the information into a low-dimensional input: the former achieves an RMSE reduction of $66 - 71\%$, outperforming the estimators with $N_{train} = 128$ and matching the performance of the original PEMC ($\dim(X) = 14$) with $N_{train} = 1.28 \cdot 10^6$.
A direct consequence of the reduction of N_{train} is a shorter training time for the same RMSE reduction level and a similar number of training epochs, as reported in Table 4.5. As highlighted before, it is a significant advantage since it allows for a model that quickly adapts to the current market conditions.

Method	Training time [s/epoch]
Differential PEMC ("split")	0.046
PEMC ($\dim(X) = 14$)	3.84

Table 4.5: Training time of the Differential PEMC estimator with "split" preprocessing compared to standard PEMC with $N_{train} = 1.28 \cdot 10^6$ in the PEMC variance reduction example

4.1.2. "Full" preprocessing

The implementation of the "full" preprocessing requires a slight modification of the `DiffPCA` class with respect to the "split" case: the `fit` method begins with a call to the `build_X` method, which concatenates θ and the Brownian increments into the input matrix for the preprocessing pipeline. In this case, 3 components are identified as the ones that explain at least 99.9% of the total relevance.

Method	n = 1000	n = 4000	n = 9000
Monte Carlo (MC)	0.264495	0.129612	0.080627
Differential PEMC ("full")	0.095265	0.052996	0.033198
Geometric CV	0.010705	0.005068	0.003387

Table 4.6: RMSE for Differential PEMC variance reduction example with "full" preprocessing

Method	n = 1000	n = 4000	n = 9000
Differential PEMC ("full")	63.982%	59.112%	58.825%

Table 4.7: RMSE reduction of the Differential PEMC estimators with respect to MC, in case of "full" preprocessing in the PEMC variance reduction example

The numerical results in Tables 4.6 and 4.7 highlight a superb performance of the estimator, with an RMSE reduction of $59 - 64\%$ with respect to MC. It is still a better result than the ones in Table 4.4, but it is slightly worse than the "split" preprocessing version. These results are coherent with the fact that, even though the preliminary steps produce informative features which result in a better performance compared to standard PEMC with $N_{train} = 1.28 \cdot 10^6$, in the payoff function the relation between θ and the Brownian increments is multiplicative and exponential, while the net receives linear combinations of them [16]. This means that the neural network should isolate the contribution of θ and the Brownian increments from the linear combinations in order to learn their multiplicative and exponential relationship. On the other hand, the "split" version keeps a clear distinction between the market parameters and the simulated path driven by the Brownian increments, simplifying the neural network's prediction task.
Similar to the "split" case, the "full" version has a lower training time compared to the standard PEMC with $N_{train} = 1.28 \cdot 10^6$ on a similar number of training epochs, as highlighted in Table 4.8. Moreover, the comparison with Table 4.5 highlights that the lighter architecture employed in the "full" approach reduces the training time per epoch of a factor of roughly 3.
To sum up, even though the "full" methodology's training time per epoch is lower than its "split" counterpart, they are both much faster than the standard PEMC method. Hence, since the focus is on variance reduction, the "split" procedure is preferable, achieving slightly better results in this sense.

Method	Training time [s/epoch]
Differential PEMC ("full")	0.016
PEMC (dim(X) = 14)	3.84

Table 4.8: Training time of the Differential PEMC estimator with "full" preprocessing compared to standard PEMC with $N_{train} = 1.28 \cdot 10^6$ in the PEMC variance reduction example

4.2. Differential Boost PEMC Variance Reduction

As in Subsection 2.3.2, since the geometric Asian call option is a well-known control variate for the arithmetic Asian call option, the Differential Boost PEMC estimator can be built upon it. Similar to the previous section, "split" and "full" preprocessing are implemented. The only difference between the Differential PEMC and the Differential Boost PEMC implementations (both "split" and "full") is the use of a different training label and a different estimator formula, as highlighted in Subsection 2.3.2.

4.2.1. "Split" preprocessing

Method	n = 1000	n = 4000	n = 9000
Geometric CV	0.010705	0.005068	0.003387
Differential Boost PEMC ("split")	0.004664	0.002420	0.001519

Table 4.9: RMSE for Differential Boost PEMC variance reduction example with "split" preprocessing

Method	n = 1000	n = 4000	n = 9000
Differential Boost PEMC ("split")	56.431%	52.258%	55.146%

Table 4.10: RMSE reduction of the Differential Boost PEMC estimators with respect to CV, in case of "split" preprocessing

In this case, the data preparation procedure with the best results identifies 2 components as the ones explaining at least 99% of the relevance of θ and the Brownian increments. The numerical results in Tables 4.9 and 4.10, produced with a training dataset with $1.28 \cdot 10^4$ samples, show that the Differential Boost PEMC estimator achieves $52 - 56\%$ RMSE reduction compared to the Geometric CV estimator, which is a better result than the

$36-43\%$ achieved by the standard Boost PEMC with $N_{train} = 1.28 \cdot 10^6$. Furthermore, the performance of the Differential Boost PEMC estimator is even better than the standard Boost PEMC with $N_{train} = 1.28 \cdot 10^4$, as reported in Tables 4.11 and 4.12.

Method	**n = 1000**	**n = 4000**	**n = 9000**
Geometric CV	0.010705	0.005068	0.003387
Boost PEMC	0.008429	0.004527	0.002789

Table 4.11: RMSE for Boost PEMC variance reduction example with $N_{train} = 1.28 \cdot 10^4$

Method	**n = 1000**	**n = 4000**	**n = 9000**
Boost PEMC	21.260%	10.683%	17.636%

Table 4.12: RMSE reduction of the Boost PEMC estimators with respect to CV with $N_{train} = 1.28 \cdot 10^4$

As before, the use of a smaller dataset is beneficial for the Differential Boost PEMC estimator also in terms of training time on a similar number of epochs, as shown in Table 4.13. In particular, since the model architecture is exactly the same as Subsection 2.3.1, the reduction is a direct consequence of the decrease in N_{train}.

Method	**Training time [s/epoch]**
Differential Boost PEMC ("split")	0.64
Boost PEMC ($N_{train} = 1.28 \cdot 10^6$)	2.33

Table 4.13: Training time of the Differential Boost PEMC estimator with "split" preprocessing compared to Boost PEMC with $N_{train} = 1.28 \cdot 10^6$

4.2.2. "Full" preprocessing

Method	**n = 1000**	**n = 4000**	**n = 9000**
Geometric CV	0.010705	0.005068	0.003387
Differential Boost PEMC ("full")	0.004307	0.002139	0.001427

Table 4.14: RMSE for Differential Boost PEMC variance reduction example with "full" preprocessing

Method	n = 1000	n = 4000	n = 9000
Differential Boost PEMC ("full")	59.766%	57.794%	57.868%

Table 4.15: RMSE reduction of the Differential Boost PEMC estimators with respect to CV, in case of "full" preprocessing

The "full" preprocessing identifies 5 features as the group that explains at least 99% of the total relevance. Tables 4.14 and 4.15, obtained with a training dataset of $N_{train} = 1.28 \cdot 10^4$ samples, show an RMSE reduction of $58 - 60\%$, which is slightly higher than the result obtained by the "split" version of the data preparation procedure. This observation is in contrast with the results of the previous section, where the "split" version yielded a higher RMSE reduction than the "full" preprocessing, but this is likely related to the different labels used in the two training datasets. Since the geometric Asian call option is a well-known control variate for its arithmetic counterpart, it already captures the dependencies from θ and the Brownian increments alone, so the difference between the payoffs of the two types of Asian call options (the training label in the Boost PEMC case) is likely driven by particular interactions between θ and the Brownian increments, which by construction are captured by the "full" method in an easier way than the "split" version.

Method	Training time [s/epoch]
Differential Boost PEMC ("full")	0.36
Boost PEMC ($N_{train} = 1.28 \cdot 10^6$)	2.33

Table 4.16: Training time of the Differential Boost PEMC estimator with "full" preprocessing compared to Boost PEMC with $N_{train} = 1.28 \cdot 10^6$

Table 4.16 confirms that a smaller training dataset results in significantly less training time. Indeed, using the same model architecture with fewer features and training samples reduces the training time by a factor of roughly 6.5. In addition, the "full" method is faster than its "split" counterpart (Table 4.13): this is likely a consequence of the use of a lighter architecture in the "full" approach. In this scenario, a lower training time per epoch and better variance reduction compared to MC make the 'full' method preferable to the 'split' one.

4.3. Differential PEMC for Swaptions Under Heath-Jarrow-Morton Model

The significant results obtained in the previous sections lead to testing the method against a challenging real-world scenario to further validate the approach. For this reason, the data preparation algorithm is applied to the scenario described in Subsection 2.3.5. Due to the different context, "split" and "full" preprocessing are slightly different than before: the former applies the preprocessing to the concatenation of θ and the Brownian increments and feeds the "Vector Features Encoder" in Figure 2.3 with the obtained features, while the other two branches of the architecture in Figure 2.3 are fed with the same inputs of the standard case; the latter performs the data preparation procedure on the concatenation of θ, the Brownian increments and the flattened versions of the volatility surface $\sigma(\cdot, \cdot)$ and the initial forward rates grid $f(0, \cdot)$, feeding the "Synthesizer" in Figure 2.3 with the resulting group of features.

4.3.1. "Split" preprocessing

In "split" preprocessing, the `DiffPCA` class is the same as in the "full" version of the Differential PEMC variance reduction example, since θ and the Brownian increments are processed together. On the other hand, `setup_global_pca`'s structure is slightly modified with respect to the previous examples, employing a batched generation of the payoff due to its significant simulation cost:

```
def setup_global_pca(..., gen_batch_size=1024):
    ctx = HJMContext(...)

    #...initialize lists to collect results...

    for i in range(0, N_calibration, gen_batch_size):
        current_bs = min(gen_batch_size, N_calibration - i)

        #...generate chunk_theta and c_W_dt...

        chunk_theta.requires_grad_(True)
        c_W_dt.requires_grad_(True)

        _, _, _, c_payoff = opt_hjm_train(...)

```

```
        grads_raw = torch.autograd.grad(outputs=c_payoff, inputs=[
            chunk_theta, c_W_dt], ...)

        # Store results
        theta_list.append(chunk_theta.detach())
        W_dt_list.append(c_W_dt.detach())
        payoff_list.append(c_payoff.detach())
        batch_grads = torch.cat([g.reshape(current_bs, -1) for g
            in grads_raw], dim=1)
        grads_list.append(batch_grads.detach())
        # Clean cache
        del chunk_theta, c_W_dt, c_payoff, grads_raw
        torch.cuda.empty_cache()

    # Concatenate all batches
    theta = torch.cat(theta_list, dim=0)
    #...concatenate W_dt, payoff and grads batches...

    transformer = DiffPCA(n_components_pca=1-1e-10,
        n_components_diff_pca=1-1e-4, device=device)
    transformer.fit(...)

    return transformer
```

Listing 4.2: `setup_global_pca` for "split" preprocessing in the HJM example

This preliminary step identifies 5 features as the group that explains at least 99.99% of the relevance of θ and the Brownian increments concatenated. Training the architecture of Figure 2.3 with $N_{train} = 3 \cdot 10^4$ and these modified features yields the results in Tables 4.17 and 4.18.

Method	**n = 1000**	**n = 3000**	**n = 5000**	**n = 7000**	**n = 9000**	**n = 11000**
MC	0.028634	0.016532	0.013120	0.011022	0.009024	0.008439
Differential PEMC	0.014888	0.008243	0.006548	0.005302	0.004808	0.004048

Table 4.17: RMSE for Differential PEMC in the HJM example with "split" preprocessing

Method	n = 1000	n = 3000	n = 5000	n = 7000	n = 9000	n = 11000
Differential PEMC	48.005%	50.141%	50.095%	51.891%	46.718%	52.036%

Table 4.18: RMSE reduction of the Differential PEMC estimators with respect to MC, in case of "split" preprocessing in the HJM example

The Differential PEMC estimator achieves an RMSE reduction of $47-52\%$, which is very similar to the original PEMC estimator's results with the full dataset ($45-50\%$). This is further proof of the method's efficiency, which is strongly remarked by the comparison with the PEMC estimator's results with $N_{train} = 3 \cdot 10^4$ (Tables 4.19 and 4.20).

Method	n = 1000	n = 3000	n = 5000	n = 7000	n = 9000	n = 11000
MC	0.028634	0.016532	0.013120	0.011022	0.009024	0.008439
PEMC	0.030510	0.016400	0.015039	0.011364	0.009943	0.008659

Table 4.19: RMSE for PEMC estimator with $N_{train} = 3 \cdot 10^4$ in the HJM example

Method	n = 1000	n = 3000	n = 5000	n = 7000	n = 9000	n = 11000
PEMC	-6.552%	0.799%	-14.622%	-3.111%	-10.186%	-2.607%

Table 4.20: RMSE reduction of the PEMC estimators with respect to MC with $N_{train} = 3 \cdot 10^4$ in the HJM example

The PEMC estimator with $N_{train} = 3 \cdot 10^4$ achieves very similar RMSE results to MC, making the PEMC approach useless in this context. This is likely due to the combination of a high statistical error with the higher complexity of the scenario compared to the Asian option pricing under the Black&Scholes model, where the effect of the decrease in N_{train} is mitigated by the neural network's ability to track the simple dependencies. The fact that, by just introducing the preprocessing pipeline, the PEMC method shifts from useless to an advantage compared to MC shows the real potential of the data preparation method.

Method	Training time [s/epoch]
Differential PEMC ("split")	2.09
PEMC ($N_{train} = 3 \cdot 10^6$)	289.89

Table 4.21: Training time of the Differential PEMC estimator with "split" preprocessing compared to standard PEMC with $N_{train} = 3 \cdot 10^6$ in the HJM example

The training time reduction obtained by introducing the data preparation algorithm in the Asian option examples is confirmed in this context. In particular, Table 4.21 shows a huge decrease in training time, which is the consequence of the considerable simulation cost per sample in the HJM example.

4.3.2. "Full" preprocessing

The implementation of the "full" data preparation algorithm requires some modifications to the code. First of all, the simulation of the volatility surfaces and the initial forward curves is performed by `simulate_sigma_f0`, separately from the simulation of the payoff done by `simulate_hjm_swaption`. This is necessary to enable the gradient tracking of `sigma` and `f0` before the simulation of the payoff in `setup_global_pca`:

```
def setup_global_pca(..., gen_batch_size=1024):
    ctx = HJMContext(...)

    #...initialize lists to collect results...

    for i in range(0, N_calibration, gen_batch_size):
        current_bs = min(gen_batch_size, N_calibration - i)

        #...generate chunk_theta and c_W_dt...

        c_W_dt.requires_grad_(True)

        with torch.no_grad():
            # Simulate sigma and f0
            c_sigma, c_f0 = simulate_sigma_f0(...)

        c_sigma.requires_grad_(True)
        c_f0.requires_grad_(True)

        c_payoff = simulate_hjm_swaption(...)

        grads_raw = torch.autograd.grad(outputs=c_payoff, inputs=[
           c_W_dt, c_sigma, c_f0], ...)

        # Store results
        W_dt_list.append(c_W_dt.detach().cpu())
        sigma_list.append(c_sigma.detach().cpu())
```

```
        f0_list.append(c_f0.detach().cpu())
        payoff_list.append(c_payoff.detach().cpu())

        batch_grads = torch.cat([g.reshape(current_bs, -1) for g
            in grads_raw], dim=1)
        grads_list.append(batch_grads.detach().cpu())

        # Clean cache
        del chunk_theta, c_W_dt, c_sigma, c_f0, c_payoff,
            grads_raw, batch_grads
        torch.cuda.empty_cache()

    # Concatenate all batches
    sigma = torch.cat(sigma_list, dim=0)
    f0 = torch.cat(f0_list, dim=0)
    W_dt = torch.cat(W_dt_list, dim=0)
    payoff = torch.cat(payoff_list, dim=0)

    grads = torch.cat(grads_list, dim=0)

    transformer = DiffPCA(n_components_pca=1-1e-10,
       n_components_diff_pca=1-1e-4, device=device)
    transformer.fit(W_dt, sigma, f0, payoff, grads)

    return transformer
```

Listing 4.3: `setup_global_pca` for "full" preprocessing in the HJM example

Moreover, the gradients with respect to θ are not computed, since in this example θ does not contain market variables, but just parameters used in the initialization of the volatility surfaces and the initial forward curves. For this reason, the presence of θ together with the surfaces in the matrix used for the preprocessing would introduce unnecessary noise to the procedure. Moreover, the `fit` method is executed on the CPU, since the NVIDIA L4 GPU employed for the experiments does not have enough memory to contain the covariance matrices used for PCA and differential PCA. However, the fitted transformation matrices are transferred to the GPU at the end of the procedure, in order to be easily available for the transformations of the features, which are performed on the GPU. The price to pay to make the data preparation algorithm feasible is a slightly slower fit (in the order of a few minutes) compared to the one performed in the "split" case.

The enormous amount of features that the data preparation algorithm receives as input (i.e., the flattened surfaces and the Brownian increments) leads to the implementation of a further optimization in the code: the PCA for high-dimensional data introduced in Section 1.6:

```
def compute_pca(self, A, n_components):
    n_samples, n_features = A.shape
    use_dual = n_samples < n_features

    if use_dual:
        K = A @ A.T / n_samples
        d, U = torch.linalg.eigh(K)
    else:
        C = A.T @ A / n_samples
        d, P = torch.linalg.eigh(C)

    # Order descending
    d = torch.flip(d, dims=[0])
    if use_dual:
        U = torch.flip(U, dims=[1])
    else:
        P = torch.flip(P, dims=[1])

    #...computation of the number of components to keep...

    d_reduced = d[:self.n_components_]

    if use_dual:
        U_reduced = U[:, :self.n_components_]
        P_reduced = A.T @ U_reduced
        # Normalize the eigenvector to have unitary norm
        P_reduced = F.normalize(P_reduced, p=2, dim=0)
    else:
        P_reduced = P[:, :self.n_components_]
```

Listing 4.4: Integration of PCA for high-dimensional data in `compute_pca`

Since the function `compute_pca` is used to perform both PCA and differential PCA, the code implements both versions of the PCA since the dimension of the differential PCA's input is not known a priori.

The preprocessing procedure identifies 4 features as the ones that explain at least 99.9% of the total relevance. The RMSE results are in Tables 4.22 and 4.23.

Method	**n = 1000**	**n = 3000**	**n = 5000**	**n = 7000**	**n = 9000**	**n = 11000**
MC	0.028634	0.016532	0.013120	0.011022	0.009024	0.008439
Differential PEMC	0.027306	0.015405	0.012567	0.010670	0.008761	0.007346

Table 4.22: RMSE for Differential PEMC in the HJM example with "full" preprocessing

Method	**n = 1000**	**n = 3000**	**n = 5000**	**n = 7000**	**n = 9000**	**n = 11000**
Differential PEMC	4.636%	6.814%	4.220%	3.193%	2.916%	12.958%

Table 4.23: RMSE reduction of the Differential PEMC estimators with respect to MC, in case of "full" preprocessing in the HJM example

The results confirm the theoretical intuition that the data preparation procedure is beneficial for the PEMC method, since the differential estimator with "full" preprocessing achieves $3 - 13\%$ RMSE reduction compared to the MC method, against a negligible reduction for the standard estimator with $N_{train} = 3 \cdot 10^4$. In addition, as observed in the Section 4.1, the "split" method in Tables 4.17 and 4.18 achieves a better performance than the "full" one. This is likely due to the 2D CNN ("2D Function Encoder" in Figure 2.3) detecting local volatility surface patterns [32], while the 1D CNN ("1D Function Encoder" in Figure 2.3) captures the forward curve's sequential behaviour [26].

Method	**Training time [s/epoch]**
Differential PEMC ("full")	9.39
PEMC ($N_{train} = 3 \cdot 10^6$)	289.89

Table 4.24: Training time of the Differential PEMC estimator with "full" preprocessing compared to standard PEMC with $N_{train} = 3 \cdot 10^6$ in the HJM example

Finally, as shown in Table 4.24, the overall training time is lower compared to the standard PEMC approach with $N_{train} = 3 \cdot 10^6$. However, it is important to note that the average training time per epoch is approximately four times longer than that of the "split" data preparation method. This slowdown occurs even though the "full" preprocessing approach employs a simpler model architecture, since the bottleneck lies in the massive dimensionality of the inputs passed to the `transform` method: while the "split" version

only processes the Brownian increments (and the θ parameter), the "full" version applies PCA and differential PCA to the entire volatility surface and the initial forward curve, increasing the computational effort.
To conclude, the "split" dimension reduction procedure emerges as the superior approach in the HJM framework, thanks to its faster training and higher variance reduction level compared to the "full" method. Furthermore, the experiments in this chapter clarify that preserving the separation of input data (the "split" method) is often beneficial. By feeding distinct data types into specialized network branches (e.g., using 2D CNN for local volatility surfaces and 1D CNN for forward curves), the network can capture underlying dynamics more efficiently than when forced to untangle a concatenated input vector. However, the success of the "full" preprocessing in the Differential Boost PEMC example proves that concatenating features can be superior when the target label is driven heavily by the complex interactions of the variables. Regardless of the chosen preprocessing configuration, the integration of differential PCA consistently proves its worth. By filtering out non-relevant noise, it drastically reduces the required training samples and computational overhead, successfully evolving the PEMC approach into a faster pricing tool.

5 | Conclusions and Future Developments

This master's thesis investigated the effectiveness of the Prediction-Enhanced Monte Carlo framework [20] in delivering unbiased and low-variance estimators. The method employs a neural network-based predictor to act as a flexible control variate, extending the use of this variance reduction technique to scenarios where a closed-form control variate does not exist, and enhancing its efficiency when one does.

The validity of the PEMC approach was demonstrated through its application to five different realistic scenarios: pricing an arithmetic Asian call option under the Black&Scholes model; pricing the same derivative using the geometric Asian call option as a baseline; pricing variance swaps under both Heston and Stochastic Local Volatility models and pricing a swaption under the Heath-Jarrow-Morton framework. The results showed a significant variance reduction of the PEMC estimator compared to standard Monte Carlo and, when available, standard control variates estimators in every test case, confirming the theoretical advantage of using the PEMC approach.

A data preparation pipeline combining scaling, PCA, and differential PCA was added to the PEMC approach, inspired by [15]. The goal of this procedure, called Differential PEMC, is to select features based on their relevance to the label rather than their variance, allowing for more aggressive filtering and significantly reducing the number of variables to a small group of highly informative ones. As a result, it is possible to use fewer training samples, considerably reducing training time without compromising the level of variance reduction.

The Asian option examples and the HJM scenario were used to test the effectiveness of this preprocessing procedure and the addition proved beneficial for the PEMC estimator's variance reduction. Two versions of the preprocessing technique were implemented: a "split" version, where features are processed separately by the neural network and then combined and fed into an additional multi-layer perceptron and a "full" version, where features are concatenated before being passed to the data preparation algorithm and the neural network. In particular, the "full" version proved effective when the estimator was

built upon an existing control variate, while the "split" version enhanced variance reduction in the absence of a known control variate.
The promising results of the PEMC and Differential PEMC approaches motivate further refinement of the methods. Among the possible directions for future research, the application of meta-learning [8] to extend the method's efficiency across a large range of parameters is particularly intriguing. Additionally, the implementation of a unified model architecture for all possible scenarios [21, 24] and the use of Quasi-Monte Carlo techniques [22, 23] to further enhance variance reduction are promising avenues for development.

Bibliography

[1] A.G.Z. Kemna and A.C.F. Vorst. A pricing method for options based on average asset values. *Journal of Banking and Finance*, 14(1):113–129, 1990.

[2] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[3] P. Blanchard, D. J. Higham, and N. J. Higham. Accurately computing the log-sum-exp and softmax functions. *IMA Journal of Numerical Analysis*, 41(4):2311–2330, 08 2020. ISSN 0272-4979. doi: 10.1093/imanum/draa038. URL `https://doi.org/10.1093/imanum/draa038`.

[4] H. Buehler, L. Gonon, J. Teichmann, and B. Wood. Deep hedging. *Quantitative Finance*, 19(8):1271–1291, 2019.

[5] R. Carmona. *HJM: A unified approach to dynamic models for fixed income, credit and equity markets*, pages 1–50. Lecture Notes in Mathematics. Springer Verlag, Germany, 2007. ISBN 3540733264. doi: 10.1007/978-3-540-73327-0_1.

[6] T. F. Coleman, Y. Li, and A. Verma. Reconstructing the unknown local volatility function. 1999. URL `https://api.semanticscholar.org/CorpusID:336822`.

[7] M. Ferguson and A. Liang. Deep learning for pricing and hedging american-style options. *arXiv preprint arXiv:1812.11033*, 2018.

[8] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[9] M. Giles and P. Glasserman. Smoking adjoints: Fast monte carlo greeks. *Risk*, 19 (1):88–92, 2006.

[10] P. Glasserman. *Monte Carlo methods in financial engineering*, volume 53. Springer, 2013.

[11] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-

level performance on imagenet classification, 2015. URL https://arxiv.org/abs/1502.01852.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[13] D. Heath, R. Jarrow, and A. Morton. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica*, 60(1):77–105, January 1992. URL https://ideas.repec.org/a/ecm/emetrp/v60y1992i1p77-105.html.

[14] B. Horvath, A. Muguruza, and M. Tomas. Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance*, 21(1):11–27, 2021.

[15] B. Huge and A. Savine. Differential machine learning. *arXiv preprint arXiv:2005.02347*, 2020.

[16] B. Huge and A. Savine. Axes that matter: Pca with a difference. *arXiv preprint arXiv:2503.06707*, 2025.

[17] J. M. Hutchinson, A. W. Lo, and T. Poggio. A nonparametric approach to pricing and hedging derivative securities via learning networks. *The journal of Finance*, 49 (3):851–889, 1994.

[18] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

[19] Y. Lecun, Y. Yere, P. Haffner, Y. Rachmad, and L. Bottou. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998. doi: 10.1109/5.726791.

[20] F. Li, H. Chen, J. Lin, A. Gupta, X. Tan, H. Zhao, G. Xu, Y. Nevmyvaka, A. Capponi, and H. Lam. Prediction-enhanced monte carlo: A machine learning view on control variate, 2025. URL https://arxiv.org/abs/2412.11257.

[21] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint arXiv:2010.08895*, 2020.

[22] H. Lin, H. Chen, K. M. Choromanski, T. Zhang, and C. Laroche. Demystifying

orthogonal monte carlo and beyond. *Advances in Neural Information Processing Systems*, 33:8030–8041, 2020.

[23] S. Liu. Langevin quasi-monte carlo. *Advances in Neural Information Processing Systems*, 36:75338–75353, 2023.

[24] L. Lu, P. Jin, and G. E. Karniadakis. Deeponet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators. *arXiv preprint arXiv:1910.03193*, 2019.

[25] M. A. Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[26] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL `https://arxiv.org/abs/1912.01703`.

[28] PyTorch. Pytorch documentation: A gentle introduction to torch.autograd, 2017. URL `https://docs.pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html`.

[29] PyTorch. Pytorch documentation: Autograd mechanics, 2017. URL `https://docs.pytorch.org/docs/stable/notes/autograd.html`.

[30] PyTorch. Pytorch documentation: Automatic differentiation with torch.autograd, 2021. URL `https://docs.pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html`.

[31] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[32] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. URL `https://arxiv.org/abs/1409.1556`.

[33] W. Zheng and Y. K. Kwok. Closed form pricing formulas for discretely sampled generalized variance swaps. *Mathematical Finance*, 24(4):855–881, 2014.

A | Appendix A

Derivation of the optimal α for PEMC(θ, α)

The formula of PEMC(θ, α) is:

$$\begin{aligned}\text{PEMC}(\theta,\alpha) &= \frac{1}{n}\sum_{i=1}^{n}\big(f_\theta(Y_i) - \alpha g(\theta, X_i)\big) + \frac{1}{N}\sum_{j=1}^{N}\alpha g(\theta,\tilde{X}_j) = \\ &= \frac{1}{n}\sum_{i=1}^{n} f_\theta(Y_i) - \alpha\Big(\frac{1}{n}\sum_{i=1}^{n} g(\theta,X_i) - \frac{1}{N}\sum_{j=1}^{N} g(\theta,\tilde{X}_j)\Big).\end{aligned}$$

To simplify the notation, let $f_i = f_\theta(Y_i)$, $g_i = g(\theta, X_i)$, $\tilde{g}_j = g(\theta, \tilde{X}_j)$ and $\mathbb{E}[Z] = \mathbb{E}^{\mathbb{Q}}_\theta[Z]$ for any random variable Z.
It follows that its variance is:

$$Var\big(\text{PEMC}(\alpha)\big) = Var\Big(\frac{1}{n}\sum_{i=1}^{n} f_i - \alpha g_i\Big) + Var\Big(\frac{1}{N}\sum_{j=1}^{N}\alpha\tilde{g}_j\Big) = Var(A) + Var(B),$$

$$\begin{aligned}Var(A) &= \frac{1}{n^2}\sum_{i=1}^{n} Var\big(f_i - \alpha g_i\big) = \frac{1}{n^2}\cdot n \cdot Var\big(f - \alpha g\big) = \frac{1}{n}Var(f - \alpha g) = \\ &= \frac{1}{n}\Big(Var(f) + \alpha^2 Var(g) - 2\alpha Cov(f,g)\Big),\end{aligned}$$

$$Var(B) = \frac{\alpha^2}{N^2}\sum_{j=1}^{N} Var(\tilde{g}_j) = \frac{\alpha^2}{N^2}\cdot N \cdot Var\big(\tilde{g}\big) = \frac{\alpha^2}{N}\cdot Var(g).$$

Summing up:

$$Var\big(\text{PEMC}(\alpha)\big) = \frac{1}{n}Var(f) - \frac{2\alpha}{n}Cov(f,g) + \alpha^2 Var(g)\Big(\frac{1}{n} + \frac{1}{N}\Big).$$

The minimization of this quantity, done by imposing the partial derivative with respect to α equal to 0 leads to the optimal value:

$$\alpha^* = \frac{Cov\big(f_\theta(Y), g(\theta, X)\big)}{\big(n/N + 1\big)Var\big(g(\theta, X)\big)}.$$

In practice, $N \gg n$ and, in the ideal scenario, $g(\theta, X) = \mathbb{E}^{\mathbb{Q}}_{\theta}[f_\theta(Y)|\theta, X]$. The starting point is that:

$$Cov(f, g) = \mathbb{E}[f \cdot g] - \mathbb{E}[f] \cdot \mathbb{E}[g].$$

Then, exploiting the tower property:

$$\mathbb{E}[f \cdot g] = \mathbb{E}\big[\mathbb{E}[f \cdot g|\theta, X]\big],$$

$$\mathbb{E}[g] = \mathbb{E}\big[\mathbb{E}[f|\theta, X]\big] = \mathbb{E}[f].$$

Moreover, due to the fact that g is a deterministic function of (θ, X):

$$\mathbb{E}[f \cdot g|\theta, X] = g \cdot \mathbb{E}[f|\theta, X] = g \cdot g = g^2 \Rightarrow \mathbb{E}\big[\mathbb{E}[f \cdot g|\theta, X]\big] = \mathbb{E}[g^2].$$

Finally:

$$Cov(f, g) = \mathbb{E}[g^2] - \mathbb{E}[g]^2 = Var(g). \tag{A.1}$$

A further result is:

$$Cov(f - g, g) = Cov(f, g) - Cov(g, g) = Var(g) - Var(g) = 0,$$

that follows by (A.1) and by linearity of the covariance.

B | Appendix B

Derivation of the closed-form expected payoff of a geometric Asian option with discrete monitoring under the Black&Scholes model

According to the Black&Scholes model, the dynamics of the stock price S_t are given by:

$$dS_t = rS_t dt + \sigma S_t dW_t,$$

hence

$$S_t = S_0 \cdot e^{\left(r - \frac{\sigma^2}{2}\right)t + \sigma W_t}, \tag{B.1}$$

where W_t is a Wiener process and r and σ are constants. The goal is to find a closed-form expression for:

$$P_G^{exact} = \mathbb{E}_0^{\mathbb{Q}}[(G_T - K)^+],$$

with T the maturity of the geometric Asian call option, $G_T = \left(\prod_{i=1}^n S_{t_i}\right)^{\frac{1}{n}} = e^{\frac{1}{n}\sum_{i=1}^n \ln(S_{t_i})}$ the geometric average of the stock price and K the strike price of the option.
Because of the form of G_T:

$$\ln(G_T) = \frac{1}{n}\sum_{i=1}^{n} \ln(S_{t_i}),$$

where, according to (B.1):

$$\ln(S_{t_i}) = \ln(S_0) + \left(r - \frac{\sigma^2}{2}\right) t_i + \sigma W_{t_i},$$

With this construction, $\ln(G_T)$ is normally distributed [1] with variance:

$$\begin{aligned}
Var(\ln(G_T)) &= Var\left(\frac{1}{n}\sum_{i=1}^{n}\ln(S_{t_i})\right) = Var\left(\frac{1}{n}\sum_{i=1}^{n}\sigma W_{t_i}\right) = \frac{\sigma^2}{n^2}Var\left(\sum_{i=1}^{n}W_{t_i}\right) = \\
&= \frac{\sigma^2}{n^2}\left[\mathbb{E}\left[\left(\sum_{i=1}^{n}W_{t_i}\right)^2\right] - \mathbb{E}\left[\sum_{i=1}^{n}W_{t_i}\right]^2\right] = \frac{\sigma^2}{n^2}\mathbb{E}\left[\left(\sum_{i=1}^{n}W_{t_i}\right)^2\right] = \\
&= \frac{\sigma^2}{n^2}\mathbb{E}\left[\left(\sum_{i=1}^{n}W_{t_i}\right)\left(\sum_{j=1}^{n}W_{t_j}\right)\right] = \frac{\sigma^2}{n^2}\mathbb{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n}W_{t_i}W_{t_j}\right] = \\
&= \frac{\sigma^2}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbb{E}\left[W_{t_i}W_{t_j}\right] = \frac{\sigma^2}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\min(t_i,t_j) = \frac{\sigma^2 T}{n^3}\sum_{i=1}^{n}\sum_{j=1}^{n}\min(i,j) = \\
&= \frac{\sigma^2\cdot T}{n^3}\cdot\frac{n(n+1)(2n+1)}{6} = \frac{\sigma^2 T(n+1)(2n+1)}{6n^2} = \sigma_n^2 T,
\end{aligned}$$

and expected value:

$$\begin{aligned}
\mathbb{E}\left[\ln(G_T)\right] &= \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}\ln(S_0) + \left(r-\frac{\sigma^2}{2}\right)t_i + \sigma W_{t_i}\right] = \ln(S_0) + \left(r-\frac{\sigma^2}{2}\right)\frac{T}{n^2}\sum_{i=1}^{n}i = \\
&= \ln(S_0) + \left(r-\frac{\sigma^2}{2}\right)\frac{T}{n^2}\cdot\frac{n(n+1)}{2} = \ln(S_0) + \left(r-\frac{\sigma^2}{2}\right)\frac{T(n+1)}{2n} = \\
&= \ln(S_0) + \left(\mu_n - \frac{\sigma_n^2}{2}\right)T,
\end{aligned}$$

where:

$$\sigma_n =: \sqrt{\frac{\sigma^2(n+1)(2n+1)}{6n^2}} \qquad \mu_n =: \left(r-\frac{\sigma^2}{2}\right)\frac{n+1}{2n} + \frac{\sigma_n^2}{2},$$

Thanks to this result, it is possible to write:

$$\ln(G_T) \overset{\text{"law"}}{\sim} \ln(S_0) + \left(\mu_n - \frac{\sigma_n^2}{2}\right)T - \sigma_n\sqrt{T}Z, \qquad Z\sim N(0,1),$$

hence:

$$G_T \overset{\text{"law"}}{\sim} S_0\cdot e^{\left(\mu_n-\frac{\sigma_n^2}{2}\right)T-\sigma_n\sqrt{T}Z}, \qquad Z\sim N(0,1). \tag{B.2}$$

Now it is possible to compute the expected payoff under the risk-neutral measure $\mathbb{Q}$ conditional on the filtration at time 0:

$$\mathbb{E}_0^{\mathbb{Q}}\left[(G_T-K)^+\right] = \underbrace{\mathbb{E}_0^{\mathbb{Q}}\left[G_T\cdot\mathbb{I}_{\{G_T\geq K\}}\right]}_{(A)} - K\cdot\underbrace{\mathbb{E}_0^{\mathbb{Q}}\left[\mathbb{I}_{\{G_T\geq K\}}\right]}_{(B)}.$$

To compute (A) and (B), it is possible to notice that, due to (B.2):

$$G_T \geqslant K \iff S_0 \exp\left[\left(\mu_n - \frac{\sigma_n^2}{2}\right)T - \sigma_n\sqrt{T}Z\right] \geqslant K$$

$$\iff \left(\mu_n - \frac{\sigma_n^2}{2}\right)T - \sigma_n\sqrt{T}Z \geqslant \ln\left(\frac{K}{S_0}\right)$$

$$\iff Z \leqslant \frac{\ln\left(\frac{S_0}{K}\right) + \left(\mu_n - \frac{\sigma_n^2}{2}\right)T}{\sigma_n\sqrt{T}} := d_2.$$

The computation of (A) and (B) follows from this result:

$$\begin{aligned}
(A) &= \mathbb{E}_0^{\mathbb{Q}}\left[G_T \cdot \mathbb{I}_{\{G_T \geq K\}}\right] = \mathbb{E}_0^{\mathbb{Q}}\left[S_0 \exp\left(\left(\left(\mu_n - \frac{\sigma_n^2}{2}\right)T - \sigma_n\sqrt{T}Z\right) \cdot \mathbb{I}_{\{G_T \geq K\}}\right]\right] = \\
&= S_0 \int_{-\infty}^{d_2} \frac{e^{-\frac{z^2}{2}}}{\sqrt{2\pi}} \exp\left(\left(\mu_n - \frac{\sigma_n^2}{2}\right)T - \sigma_n\sqrt{T}z\right)dz = \\
&= S_0 e^{\mu_n T} \int_{-\infty}^{d_2} \frac{e^{-\frac{z^2}{2}}}{\sqrt{2\pi}} \exp\left(-\frac{\sigma_n^2}{2}T - \sigma_n\sqrt{T}z\right)dz = \\
&= S_0 e^{\mu_n T} \int_{-\infty}^{d_2} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(z^2 + 2z\sigma_n\sqrt{T} + \sigma_n^2 T\right)\right)dz = \\
&= S_0 e^{\mu_n T} \int_{-\infty}^{d_2} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(z + \sigma_n\sqrt{T}\right)^2\right)dz = \\
&= S_0 e^{\mu_n T} \int_{-\infty}^{d_2+\sigma_n\sqrt{T}} \frac{1}{\sqrt{2\pi}} e^{-\frac{(z')^2}{2}} dz' = S_0 e^{\mu_n T}\mathcal{N}(d_2 + \sigma_n\sqrt{T}) = S_0 e^{\mu_n T}\mathcal{N}(d_1),
\end{aligned}$$

$$(B) = \mathbb{E}_0^{\mathbb{Q}}\left[\mathbb{I}_{\{G_T \geq K\}}\right] = \mathbb{E}_0^{\mathbb{Q}}\left[\mathbb{I}_{\{Z \leq d_2\}}\right] = \mathcal{N}(d_2),$$

where $\mathcal{N}$ is the cumulative density function of a standard Gaussian random variable.
To conclude, the closed-form expected payoff of a geometric Asian option with discrete monitoring under the Black&Scholes model is:

$$P_G^{exact} = (A) - K \cdot (B) = S_0 e^{\mu_n T}\mathcal{N}(d_1) - K\mathcal{N}(d_2)$$

$$\begin{aligned}
d_1 &= \frac{\ln\left(\frac{S_0}{K}\right) + \left(\mu_n + \frac{\sigma_n^2}{2}\right)T}{\sigma_n\sqrt{T}} \\
d_2 &= d_1 - \sigma_n\sqrt{T}
\end{aligned}$$

List of Figures

List of Tables

Acknowledgements

Per prima cosa vorrei ringraziare i miei genitori Michele e Roberta e mio fratello Filippo. Dire che siete stati fondamentali per il raggiungimento di questo traguardo può sembrare una frase fatta, ma per me è addirittura riduttivo rispetto al vostro contributo. Dal primo giorno di elementari fino all'ultimo di università mi avete sempre sostenuto e incoraggiato negli studi e grazie a voi sono riuscito a superare le difficoltà, sia nello studio che nella vita, che mi sono capitate. Festeggiare con voi i bei voti è sempre valso per me molto più del risultato numerico raggiunto. Inoltre, dal primo momento in cui ho pensato di venire a studiare a Milano, avete sostenuto questa mia scelta, anche se magari per voi comportava uno sforzo non banale dal punto di vista emotivo ed economico, e penso che questo sia uno dei segni di affetto più tangibili che avreste mai potuto darmi. Vorrei ringraziare anche i miei nonni, Fiorino, Maria, Aldo e Tosca, i miei tifosi numero 1, e tutti i miei zii e cugini, grazie per il vostro continuo supporto. Un ringraziamento speciale va anche alla mia compagnia storica, "A.T.P.", e al gruppo allargato di "Jungle Equipe's": grazie per il vostro sostegno e per avermi fatto sempre sentire a casa quando sono tornato a Pesaro, cosa per niente banale secondo me visto che da quando ho iniziato l'università abbiamo iniziato necessariamente a vederci di meno; dal mio punto di vista, questo ha rafforzato molto la nostra amicizia. Grazie anche ai miei amici di "Albero": condividere con voi il percorso universitario e la vita a Milano ha reso molto più piacevole e meno faticosa la mia esperienza universitaria. Vorrei anche ringraziare i miei coinquilini, vecchi e nuovi, e in particolare il mio storico compagno di stanza Davide, per avermi sempre fatto sentire accolto: condividere la quotidianità con voi ha reso questa esperienza indimenticabile e piena di bellissimi ricordi. Ringrazio anche tutte le altre persone con cui ho condiviso un pezzo di strada in questi 5 anni, più o meno lungo che sia, dai "Butei" di Trondheim ai "Bsares Milanes" perché, anche se magari non lo sapete, ognuno di voi ha contribuito al raggiungimento di questo traguardo e, soprattutto, a farmi crescere. Infine, vorrei anche ringraziare il mio relatore, il Prof. Marazzina, per il suo prezioso aiuto durante tutto il percorso di tesi.