Politecnico di Torino

DIPARTIMENTO DI SCIENZE MATEMATICHE
M.Sc. in Engineering Mathematics

Final dissertation

A Machine-Learning Approach to Parametric Option Pricing

Supervisor: Prof. Kathrin Glau
Co-supervisor: Prof. Paolo Brandimarte
Co-supervisor: Prof. Matthjis Breugem
Co-supervisor: Prof. Marcello Restelli

Candidate: Paolo Colusso

ACADEMIC YEAR 2018/2019

Summary

This work explores a deep-learning approach to the problem of Parametric Option Pricing. In a first phase, neural networks are used to learn a pricing function starting from a set of prices computed by means of a suitable benchmark method. In a second phase, the method is coupled with a complexity reduction technique in order for it to be scaled up to higher dimensions.
The contributions of this work are multiple. On the one hand, it shows the applicability of the neural-network approach to the parametric pricing problem. While few recent works have tackled a similar problem, this thesis shows that solutions can be found to more complex financial products, such as American and basket options. The pricer resulting from this method is fast and accurate, thus comparing favourably with the traditional Monte Carlo or PDE approaches. In addition, it provides results which are comparable to those obtained by recent developments in the use of Chebychev polynomial approximations, but without the constraints imposed by the need of having a fixed grid of Chebychev points.
On the other hand, this work shows how to make the neural-net approach scalable to higher-dimensional problems. Indeed, the high number of parameters which can enter the pricing function (model and option parameters and underlying assets) can lead to problems which are not tractable by standard machines due to memory constraints. This thesis proposes to exploit a tensor-train (TT) decomposition which significantly compresses the tensor of prices used to train the neural network. As well as ensuring an accurate representation of the tensor entries, the TT-decomposition allows to retrieve the entries by means of simple products of three-dimensional tensors. For this reason, one does not need to store the whole training tensor, but can easily compute only the few entries of the small batch of samples which are needed for stochastic gradient-based methods used in the training of the neural net.
The proposed methodology is tested on two practical cases: basket of call options, with up to 25 underlying assets, and American put options. In the first example the training data is computed by Monte Carlo methods, while finite differences for PDE are used to generate American option prices.

Sommario

Il lavoro sviluppa una metodologia basata sul deep learning per affrontare il problema del pricing parametrico di opzioni. In una prima fase si costruiscono architetture di reti neurali per apprendere funzioni di prezzo a partire da un dataset di prezzi ottenuto tramite il metodo di riferimento per lo specifico prodotto finanziario. In una seconda fase, il metodo è associato ad una tecnica di riduzione della dimensionalità in modo che sia reso applicabile a problemi di ordine maggiore.
Il contributo di questo lavoro riguarda più aspetti. Da un lato si mostra come le reti neurali abbiano notevole potenziale nel problema di pricing parametrico. Sebbene esista qualche lavoro che affronta il problema in modo simile, questa ricerca mostra come sia possibile affrontare il pricing di derivati più complessi, per esempio opzioni basket o di tipo americano, attraverso il machine learning. Il pricer ottenuto risulta veloce ed accurato, e pertanto preferibile rispetto ai tradizionali metodi Monte Carlo o di differenze finite per equazioni differenziali alle derivate parziali. Inoltre, i risultati presentano metriche in linea con quelli ottenuti da recenti sviluppi nell'uso di polinomi interpolanti di Chebychev, che impongono tuttavia il vincolo di una griglia di punti ben definita e fissa per il passaggio di interpolazione, e quindi inadatti, per esempio, all'apprendimento a partire da dati reali di mercato.
Un ulteriore contributo di questa tesi deriva dal rendere l'approccio tramite reti neurali scalabile a dimensioni maggiori. Infatti, l'elevato numero di parametri da cui può dipendere la funzione di pricing (parametri del modello, dell'opzione o degli asset sottostanti) può portare ad un problema non trattabile con macchine standard. Questa ricerca propone di sfruttare la decomposizione tensor-train, che può condurre ad una notevole compressione del tensore usato per il training delle reti neurali e quindi delle dimensioni del problema. Oltre a garantire una rappresentazione accurata delle entrate del tensore, la decomposizione tensor-train consente di ottenere le entrate tramite prodotto di tensori tridimensionali. Questo permette di non dover salvare l'intero tensore dei dati di training, ma di calcolare i pochi dati necessari per ogni fase del training della rete neurale.
Il metodo proposto è testato su due casi concreti, quello di opzioni basket con un numero di asset sottostanti fino a venticinque, e di opzioni put americane. Nel primo caso, i prezzi che costituiscono il dataset di training sono calcolati con il metodo Monte Carlo, mentre differenze finite sono usate per generare i prezzi di derivati di tipo americano.

Acknowledgements

This thesis is the outcome of five years spent at Politecnico di Torino, to which I cannot but be grateful. In a context of general suffering of the Italian university system, Politecnico has managed to engage and support motivated students in their higher education. This was the case of the program for Young Talents in the bachelor years and of Alta Scuola Politecnica in the master years. They were the occasion to join forces with other students, other backgrounds and other fields of specialisation and allowed me to grow in a nurturing environment.
I owe a lot to the challenges I was made to take on in these years. And there were many, on a daily basis, constantly putting my motivation to the test. But there were rewards as well. Three exchange programs, in Sweden and Switzerland, which could not have been more different and yet more enriching, as they gave me the confidence to be here as a global citizen, while always in a different context. But they also gave me a new lease of life each time I left, which helped.
The thesis itself was written at EPF Lausanne during my last exchange program, under the invaluable supervision of Kathrin. Always available for clarifications and last-minute meetings, she went to great lengths to provide fresh stimuli and encourage the development of new solutions, so that I could not have asked for a better mentor.
I would also like to thank Professor Brandimarte, Professor Breugem and Professor Restelli, my co-supervisors in Italy, who contributed to this work with insightful discussions and sincere enthusiasm.
Many thanks as well to Francesco, whose introduction to the low-rank part was precious.
Directly or indirectly, this thesis came to light thanks to the people I was surrounded by. Hence, thanks to my family, who had the patience to undergo non-negligible amounts of stress on my side, but encouraged me every single day. And thanks to my dad for the rides in Switzerland.
Thanks to my friends at the faculty of engineering. Matteo, who's been there when he was there, but especially when he wasn't. To Francesco, and to his company in Sweden, Norway, Denmark, the Baltics and Paris – it was always fun. To Luca, and to his – our – endurance, as things invariably seem to go wrong, but they eventually fall into place. Thanks to Alessio, who's been loyally cultivating talent with me in the past five years, and to Francesco, who's been doing so in the last two. Thanks

to those who were there in my Swiss year. Gianluca, for becoming a friend in such a short amount of time; Nora, for she's been a source of norwegian-quebecoise positivity, and Pierre, for it takes patience to listen to my French.
This thesis also marks the end of five years at Collegio Carlo Alberto, whose rooms and corridors have witnessed years of relentless study days, to say the least. And thankfully of good friendships. Thanks to Marcello, as he was the most trustworthy consultant and the most rigorous analyst whenever academic and non-academic decisions had to be made, as well as simply a friend. Thanks to Francesco, for the study days, the study breaks and the table tennis, when we still had the chance to practice it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Quantitative finance and financial engineering are active areas of research, with practitioners and academics striving to find methods which strike a balance between computational costs and accuracy. These methods and their applicability heavily rely on the ease with which one can work with the related stochastic models (Black-Scholes, Heston, SABR) for the underlying financial assets. Indeed, the modern financial industry demands speed and precision across a wide range of tasks: pricing, model calibration, hedging, risk assessment, risk management and high-frequency trading and uncertainty evaluation. This is all the more true given recent regulations and capital requirements introduced in the aftermath of the financial crisis.

In calibration, for instance, one needs to find the good model parameters matching available market prices for the financial products of interest.
Consider a set of m model parameters q taking values in a parameter space $\mathcal{Q} \subset \mathbb{R}^m$ and a set of μ derivative parameters ζ taking values in a space $\mathcal{Z} \subset \mathbb{R}^\mu$ (think of ζ, for simplicity, as the maturity and strike price of the derivative). Given market prices $\hat{P}(\zeta)$, the calibration problem can be formulated as looking for the parameters which minimise the distance between market prices and the prices obtained by a reference stochastic model for the underlying financial assets. That is:

$$\underset{q \in \mathcal{Q}}{\operatorname{argmin}}\, dist\left((\hat{P}(\zeta_i))_{i=1,\dots N}, (P(\zeta_i, q))_{i=1,\dots N} \right),$$

for a distance function which can be for instance the mean squared error.
The calibration problem is mainly tackled by heavily-traded option prices rather than by historical asset prices: it is easy to see that solving the minimisation problem requires a large number of prices for the option in order to single out the best model parameters, and these have to be computed efficiently. Indeed, the characteristics of the markets change rapidly, making the calibration process of utmost importance and requiring frequent adjustment of the parameters.
The previous motivating example illustrates the need for a method which allows to compute many prices in a fast and efficient way. One might wonder why not to use

the stochastic model for the asset dynamics to derive prices for each combination of model and derivative parameters in the first place. The answer lies at different levels.
First of all, exact solutions often do not exist, so that to obtain the future price of the underlying from a given model is only possible in few cases. One of these, the Black-Scholes model, matches computational ease with practical limitations, primarily that of its constant volatility.
Even though analytical solutions may not exist, numerical methods can, and are, used to obtain approximated ones. Methods have recently been developed, based for instance on the Fourier transform, Monte Carlo, moments or PDE discretisation. This comes with a catch, though. Although sufficiently accurate, these methods are in most cases expensive and hence hardly ever used on a large scale, as demanded, for instance, in model calibration.
For this reason, one in practice turns to a second-level approximation, by approximating approximate numerical solutions.

This work tackles a parametric derivative pricing problem. In doing so, it deals with both orders of approximation: at first, numerical solutions are found by existing methods in the literature for two categories of financial products. Later in the pipeline, these prices are generalised in an efficient and accurate way to the entire desired range of parameters' values.
The cases considered in this work are basket options, which are discussed in Section 2.1 and American put options, whose description is provided in Section 2.2: for each of these, a numerical solution is computed to the pricing problem under the reference stochastic model and with the reference method.
Subsequently, those solutions are extended to an entire range of parameters via a neural network approach, in what is called the Parametric Option Pricing (POP) problem. This involves the fast computation of option prices for a large set of both option and model parameters performed by a learning approach: in other words, after computing the price with a reference method for some grid of parameters, one also learns the price for the desired whole range of parameter values on which the price depends.
In the literature, the last step, that of learning, has mainly been done by polynomial interpolation, so as to obtain a continuous function starting from a set discrete points. Chebychev interpolation, in particular, has proven successful because of good convergence properties under some mild assumptions.

This thesis shows that deep learning via neural networks can be competitive in handling tasks of the POP type and it does so by considering the two examples mentioned above, those of American and basket options. The thesis also makes the claim that the described methodology is not always without flaws in some cases and proposes a solution which attempts at overcoming the limitations which can arise. Specifically, the deep-learning approach might fail as the training grid of points rises

in dimensionality: for parametric problems in high dimensions the size of the training data shows an exponential growth, which hinders the success of neural networks due to memory constraints. This issue motivates the introduction of a dimensionality reduction technique which allows to store the training tensor in a compact form and retrieve the data points of the tensors only when really needed during the training process, in an efficient way.

1.1 Framework and Problem Description

The general framework for financial pricing problems can be generalised as follows:

- a stochastic model is used to describe the dynamics of one or more assets of interest;
- derivatives are constructed based on these assets, which are thus called underlyings;
- the characteristics of the asset and of the financial product determine a price for the derivative;
- the suitable price has to be proposed for the financial contract.

It is clear that the stochastic model has to provide an adequate description of the dynamics of the asset over time: a trade-off is needed between the quality of the model and the ease with which it can be applied. In addition, a fair pricing mechanism has to be devised so that it is not possible to make profits without any risk whatsoever.
This work considers a two-step pricing methodology and applies it to two particular cases, basket options and American put options. While the last phase of the method is general, the first-one is problem-specific, although widely studied in the literature. Initially, we want to compute prices for the financial product of interest via a benchmark numerical method: this is Monte Carlo for the basket case, and finite differences for partial differential equations for American derivatives. The model for the underlyings also differs in the two cases, being of multi-dimensional Black-Scholes type in the former example and of Heston type in the latter.
Once the prices have been derived on a grid accounting for the parameters of interest, the second phase allows to extend the method by learning a pricing function of the parameters, for a suitable range of these.

$$f : \mathcal{Q} \times \mathcal{Z} \to \mathbb{R}^+$$

$$f : (q, \zeta) \mapsto f\,(q, \zeta) = \text{price}.$$

This last phase is performed via neural networks, whose major highlight is flexibility: neural nets are exploited to learn a general pricing function from a limited set of points, in a fast and accurate way, as prescribed by the applications requirements.

Table 1.1: **Structure of the Work**

	Basket Options		American Options	
	Type	Section	Type	Section
Stochastic Model	Black-Scholes		Heston	1.3
Numerical Method	Monte Carlo	2.1	Finite Differences for PDE	2.2
POP	Neural Nets	3	Neural Nets	3

Table 1.1 shows a comparison of the methodology for the two application cases, basket and American options.
The structure of the work follows the pipeline of the methodology.
Section 1.3 describes and motivates the use of the Heston model and related stochastic processes, namely the Ornstein-Uhlenbeck process and the Cox-Ingersoll-Ross process, employed in the American context. For these stochastic processes, some key moments are studied.
Chapter 2 describes basket options (Section 2.1) and American options (Section 2.2), as well as presenting the numerical methods used to derive their prices. Monte Carlo, coupled with a variance-reduction technique, is used in the former case, while finite differences with alternating direction implicit (ADI) method is adopted to solve the Heston PDE numerically.
Chapter 3 describes neural networks, providing a synthetic tractation of their structure and the optimisation methods used to train them.
The implementation of the pricing methodology is presented in Chapter 5, together with performance results.
The remaining chapters propose an extension of the methodology, motivated by some issues which can occur throughout the pipeline. Specifically, the goal is to deal with the curse of dimensionality which arises as options become more complex, making the problem challenging both memory-wise and computationally. This can happen as more parameters intervene in the POP problem or when large portfolios of assets are considered – in the framework of the Basket case, for instance. In these cases, even to obtain a smaller grid by reducing the number of values per parameter can prove a thorny task, demanding for a further degree of generalisation. The approach proposed consists of computing only few of the grid elements and approximating the remaining entries via low-rank techniques: by resorting to the completion algorithm, it is possible to obtain a full tensor from a subset of its entries. Furthermore, exploiting a compressed representation of the same tensor allows to save storing space. While this approach was successfully employed to Chebychev grids in the context of polynomial interpolation, we show that it is a viable option

for more general grids, claiming that it can find application in other contexts than the financial one. In addition, although the higher-order problems require a series of modifications in the training process to account for the different format of the grid, which is no longer a full tensor but a compressed one, this work shows how the compressed representation can be encompassed and how it proves invaluable in the training phase of the neural network.
Chapter 4 describes low-rank tensor approximation together with the completion algorithm used to obtain a full tensor with only few of the entries. Chapter 6 shows how results are affected when working with a synthetic dataset, i.e. obtained by tensor completion rather than entirely via the reference numerical method and shifts examples to higher dimensions.
Chapter 7 draws the conclusions with an eye on related results in the recent literature and proposes further developments of this work.

1.2 Literature Review

While several computational methods exist to obtain derivative prices, the fast and accurate computation of many of them for different choices of parameters is receiving considerable attention at the moment, as the problem of the trade-off between accuracy and computational costs is widely recognised as crucial.
Existing numerical methods have been developed to tackle this issue: besides the already mentioned operator splitting technique for finite differencing of PDEs (ADI methods in Haentjens and in 't Hout (2015)) to tackle multi-variate problems, methods have been proposed to refine numerical integration, e.g. via sparse grids, as in Holtz (2011), or to extend Monte Carlo methods, with quasi Monte Carlo or multi-level Monte Carlo, for which see for instance L'Ecuyer (2009) and Giles (2015).
Complexity reduction techniques have been equally explored due to their capacity of saving run-time and storage capacity with virtually no side-effects on the accuracy. Reduced basis methods (see Hesthaven, Rozza, G. and Stamm, (2016)) have proven successful in solving parametric PDEs which require repeated evaluation: any solutions to the parametrised problem is approximated via a limited number of basis functions. Applications have been proposed in financial engineering, for instance in Burkovska, Glau, Mahlstedt and Wohlmuth (2018).
Another form of complexity reduction technique, polynomial interpolation is also frequently employed in financial engineering problems. Gaß, Glau, Mahlstedt and Mair (2018) detail Chebychev interpolation of conditional expectations, stressing good convergence properties under hypothesis which are typically verified in the financial problems of interest. Glau, Kressner and Statti (2019) exploit the same polynomial interpolation together with a low-rank structure of the tensor of Chebychev nodes to store the nodes, efficiently compute the Fourier coefficients and readily evaluate option prices via a tensor prodcut.

The application of neural network to financial engineering and quantitative finance is more recent, especially when it comes to pricing and calibration. Results on the approximating powers of neural networks mainly date back to the last years of the twentieth century, with the pioneering work of Cybenko (1989) and Barron (1993). Neural nets were initially employed in the same years for trading strategies or credit risk models, see for instance Altman, Marco and Varetto (1994). In the last decade, the development of GPUs (graphics processing units) and formal results on the universal approximating power of neural networks have given momentum to the employment of neural net architectures in learning problems. Finance and option pricing have also started to take advantage of deep learning approaches in the last five to ten years, in two directions.

- On the one hand, some research was devoted to exploring model and parameter-free pricing functions, as was first done by Hutchinson, Lo and Poggio (1994): the authors use marketed options prices as data to train the neural net and thus recover the Black-Scholes formula. This approach can be effective when the underlying asset dynamics is particular hard to assess or when traditional no-arbitrage pricing formulas prove challenging, that is under model misspecification. Conversely, this methodology is susceptible to failure for derivatives which are seldom traded or which have not been traded before; in addition, when derivative prices can be easily obtained from the asset price dynamics, the traditional parametric approach typically performs better in pricing and hedging tasks. While a solution to the first issue is proposed in the last section of this work, the second drawback is hardly avoidable. And yet, as remarked by Hutchinson, Lo and Poggio (1994) favourable conditions for the parametric approach occur rarely.
- Another, and more recent trend, is to exploit machine learning and neural nets in model-based pricing methodologies and complement them. Some of these, as for example Sirignano and Konstantinos, (2018), use machine learning as a means to solve high-dimensional PDEs; others, such as Liu, Oosterlee and Bohte (2019), Ferguson and Green (2018) and Horvath, Muguruza and Tomas (2019), tackle the problem in a similar way to this work, focusing on the parametric option pricing (POP) task or the related one of learning the greeks. The aforementioned papers cite speed and accuracy as the main advantages of a neural-network based approach.

While this work will not analyse the performance of Monte Carlo methods and finite differences, which have been extensively studied, and will only use them as the foundation of the proposed methodology, we will consider the deep-learning approach on American and Basket options, more complex than the vanilla ones which have been used in similar studies.
In addition, we will take the method a step further by exploring the use of dimensionality reduction in the framework of neural networks. This can be done in multiple ways: first of all, it is possible to extend a small training set via a low-rank tensor

approximation. On the one hand, this should take some computational burden off the benchmark method; on the other hand, it can allow to extend the market-driven and model-free approach to financial products which are less-frequently traded, by the creation of a synthetic training set. Alternatively, and probably more importantly, this work presents the low-rank compression as a precious tool when dealing with high-dimensional problems.
The results obtained confirm and extend the preliminary ones presented in similar works, and are encouraging in the novel approach proposed. The accuracy resulting from the neural-network approaches shows that they are comparable to the benchmark methods, but with a significant gain in computational time. Furthermore, the low-rank approximation for the tensor of training data demonstrates that even fewer data are needed to represent the training grid, ensuring a further reduction in the dimensionality and a viable solution to problems where the size scales up by some orders of magnitude (because of an increased number of parameters or underlying assets) or where the training data are not generated but retrieved from the markets in a limited size.

1.3 Stochastic Models

Stochastic models in finance are intended to provide a plausible description of the dynamics of stock prices. Even though it suffers from major limitations, the Black-Scholes (BS) model is to date the most used model, because of its simplicity and ease of use. Introduced in 1973 by Black, Scholes and Merton it went on to win the Nobel Prize in economics (1997). Today, alternatives have been provided to the famous Black-Scholes model: this section describes the more recent Heston model, which sets itself as an extension of Black and Scholes'. Together with this description, the Cox, Ingersoll and Ross process and the Ornstein and Uhlenbeck process are presented, as they are useful in the presentation of the Heston model.
As mentioned in the previous chapter, we will use BS to model the underlyings in the case of Basket options, but Heston for American put options.

1.3.1 The Heston Model

In order to overcome one of the main limitations of the Black Scholes model, namely the constant volatility, the Heston model was introduced [Heston (1993)]. Although some attempts were made in the 1980s to integrate stochastic volatility in the Black Scholes model, the approach proposed by Heston presents a closed-form solution and straightforward computation of the Greeks.
In the Heston model the price process of the underlying stock follows:

$$dS_t = rS_t dt + S_t\sqrt{V_t}dW_t^1, \tag{1.1}$$

where V_t is the variance stochastic process, in turn modelled as:

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t}dW_t^2. \tag{1.2}$$

W^1 and W^2 are correlated Wiener processes, with correlation ρ, which means that

$$\mathbb{E}[dW_t^1\, dW_t^2] = \rho\, dt,$$

r is a risk-free rate and κ, θ and σ are parameters of the Cox-Ingerssoll-Ross process (1.2), whose properties are presented in section 1.4.
Specifically, the parameters mentioned correspond to:

- r, the drift process of the stock;
- κ, the mean reversion speed for the variance, $\kappa > 0$;
- θ, the mean reversion level for the variance, $\theta > 0$; and
- σ, the volatility of the variance, $\sigma > 0$.

Defining $u(t, x, y)$ as the price of a financial contract at time t, when $S_t = x$ and $V_t = y$, of a European option with payoff function Ψ, we have that by risk-neutral pricing:

$$u(t, x, y) = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}_{t,x,y}[\Psi(S_T)], \tag{1.3}$$

with $\mathbb{Q}$ risk-neutral measure.
We first observe that, from (1.3), the discounted price is a martingale, since:

$$\mathbb{E}^{\mathcal{Q}}[e^{-rT}\Psi(S_T)] = \mathbb{E}^{\mathcal{Q}}[e^{-rT}u(T, X_T, Y_T)] = e^{-rt}u(t, x, y).$$

Using Itô formula we can derive the differential of the discounted price process as follows:

$$\begin{aligned} d(e^{-rt}u(t, S_t, V_t)) = {} & e^{-rt}\left(\frac{\partial u}{\partial t}\, dt + \frac{\partial u}{\partial S_t}\, dS_t + \frac{\partial u}{\partial V_t}dV_t - ru\, dt + \frac{\partial^2 u}{\partial S_t \partial V_t}d\langle S, V\rangle_t\right) + \\ & + e^{-rt}\left(\frac{1}{2}\frac{\partial^2 u}{\partial S_t^2}d\langle S\rangle_t + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}d\langle V\rangle_t\right), \end{aligned}$$

where angular brackets denote the quadratic variation:

$$\langle W\rangle_t = \lim_{n\to\infty}\sum_{t_i\in\Pi_n}(W_{t_{i+1}} - W_{t_i})^2,$$

and, similarly,

$$\langle V, S\rangle_t = \lim_{n\to\infty}\sum_{t_i\in\Pi_n}(S_{t_{i+1}} - S_{t_i})(V_{t_{i+1}} - V_{t_i}),$$

for $(\Pi_n)_n$ sequence of partitions of $[0,t]$.
Rewriting the differential we obtain:

$$\begin{aligned} d(e^{-rt}u(t,S_t,V_t)) = & e^{-rt}\left(\frac{\partial u}{\partial t}\,dt - ru\,dt\right) \\ & + e^{-rt}\frac{\partial u}{\partial V_t}\left(\kappa(\theta - V_t)dt + \sigma\sqrt{V_t}dW_t^2\right) + \\ & + e^{-rt}\frac{\partial u}{\partial S_t}\left(rS_t dt + S_t\sqrt{V_t}dW_t^1\right) + \\ & + e^{-rt}\left(\frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t\,dt\right) + \\ & + e^{-rt}\left(\frac{1}{2}\frac{\partial^2 u}{\partial S_t^2}S_t^2 V_t\,dt + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t\,dt\right). \end{aligned}$$

However, for the discounted price process to be a martingale, the drift component has to be equal to zero. We thus impose:

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial S_t}rS_t + \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) + \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t + \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2}S_t^2 V_t + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t = ru,$$

so that the desired PDE is given by:

$$\begin{cases} \frac{\partial u}{\partial t} + \frac{\partial u}{\partial S_t}rS_t + \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) + \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t + \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2}S_t^2 V_t + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t = ru \\ u(T,x,y) = \Psi(x) \end{cases}$$

The Heston model gives an interesting modelling of the variance process. In the following subsections we will further describe such a process and compute some related moments. This will be done by studying the Ornstein-Uhlenbeck process, which serves as the foundation to the Cox-Ingersoll-Ross process, which in turn is used to describe the variance dynamics in the Heston model. Figure 1.1 shows the so-called volatility smile under the Heston model: it is the implied volatility computed from the prices of a call option, whose discounted payoff is:

$$\pi = e^{-rT}\max\{e^x - e^k, 0\},$$

with k log-strike, x log-price and T maturity.

1.3.2 The Ornstein-Uhlenbeck Process

We consider a process whose differential increments are given by:

$$dX_t = \kappa(\theta - X_t)dt + \lambda dW_t, \tag{1.4}$$

with $\kappa, \lambda > 0$ and W_t standard Brownian motion.
We can find an explicit solution to (1.4) by considering the process $Y_t = e^{\kappa t}X_t$,

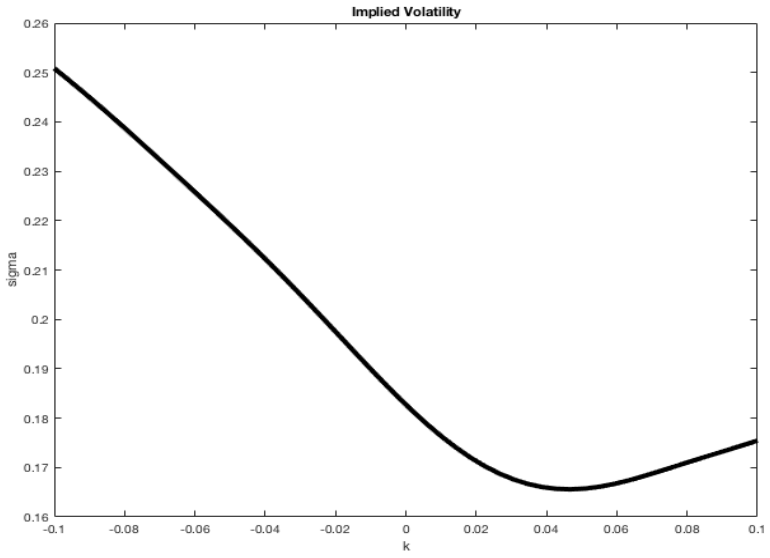


Figure 1.1: *Implied volatility under the Heston mode, exhibiting the typical volatility smile.*

whose differential can be derived as follows:

$$dY_t = \kappa Y_t dt + e^{\kappa t} dX_t = \kappa Y_t dt + e^{\kappa t}(\kappa(\theta - X_t)dt + \lambda dW_t) = \kappa\theta e^{\kappa t}\, dt + \lambda e^{\kappa t} dW_t = \\ = e^{\kappa t}\kappa\theta\, dt + \lambda e^{\kappa t} dW_t.$$

Integrating on both sides, one obtains:

$$e^{\kappa t} X_t = X_0 + (e^{\kappa t} - 1)\theta + \lambda \int_0^t e^{\kappa s} dW_s,$$

which leads to:

$$X_t = e^{-\kappa t} X_0 + (1 - e^{-\kappa t})\theta + \lambda \int_0^t e^{\kappa(s-t)} dW_s. \tag{1.5}$$

First Moment

Exploiting the solution found in the previous point, we can retrieve the expectation of X_t and its limit when t grows large.
As far as the expectation is concerned, we notice that the last term is a well-defined Itô integral, and thus its expectation is equal to zero. Hence, we have that [1]:

$$\mathbb{E}(X_t) = e^{-\kappa t} X_0 + (1 - e^{-\kappa t})\theta, \tag{1.6}$$

which, in the time limit, converges to θ. The process is said to be mean-reverting, since positive deviations from θ lead to a decrease in the value of the process and vice versa for negative ones, with a speed given by the κ parameter. In the limit and in expectation we will thus have that the process has level θ.

Second Moment

We first try to obtain the second moment of X_T in a direct way. Recalling that:

$$\mathbb{E}[X_t^2] = \mathrm{Var}(X_t) + \mathbb{E}(X_t)^2$$

As for the variance, we rely on the previous point and on Itô isometry, which holds as the negative exponential on $[0, t]$ is square-integrable. The first two components of the RHS of (1.5) are deterministic and their variance is thus equal to zero, while the last one can be computed noticing that:

$$\mathbb{E}\left[\left(\int_0^t e^{\kappa(s-t)} dW_s\right)^2\right] = \int_0^t \mathbb{E}\left[e^{2\kappa(s-t)}\right] ds = \frac{e^{-2\kappa t}}{2\kappa}\left(e^{2\kappa t} - 1\right) = \frac{1}{2\kappa}\left(1 - e^{-2\kappa t}\right)$$

Adding the terms related to the square of the expectation one has:

$$\mathbb{E}[X_T^2] = e^{-2\kappa T} X_0^2 + (1 - e^{-\kappa T})^2\theta^2 + 2e^{-\kappa T} X_0(1 - e^{-\kappa T})\theta + \frac{\lambda^2}{2\kappa}\left(1 - e^{-2\kappa T}\right) \\ = \theta^2 + e^{-2\kappa T}\left(X_0 - \theta\right)^2 + 2\theta e^{-\kappa T}(X_0 - \theta) + \frac{\lambda^2}{2\kappa}\left(1 - e^{-2\kappa T}\right)$$

[1] assuming that the sigma-algebra at time $t = 0$ is the trivial one, otherwise we should have $\mathbb{E}(X_0)$.

A second approach to obtaining the second moment relies on defining:

$$v(t,x) = \mathbb{E}_{t,x}(X_T^2), \tag{1.7}$$

where X_t is a Ornstein-Uhlenbeck process, by finding the solution of the PDE:

$$\begin{cases} v_t + \mathcal{G}v = 0 \\ v(T,x) = x^2 \end{cases} \tag{1.8}$$

$\mathcal{G}v$ is the generator of the process, defined as:

$$\mathcal{G}v = \kappa(\theta - x)v_x + \frac{\lambda^2}{2}v_{xx}.$$

We first observe that $v(t,x)$ is a martingale and the martingale property can be verified as follows:

$$\mathbb{E}(v(T,x_T)|\mathcal{F}_t) = \mathbb{E}[\mathbb{E}(X_T^2|\mathcal{F}_T)|\mathcal{F}_t] = \mathbb{E}(X_T^2|\mathcal{F}_t) = v(t,x_t).$$

The differential of $v(t,x)$ takes the form:

$$dv(t,x) = v_t dt + v_x dX_t + \frac{1}{2}v_{xx}d\langle X\rangle_t,$$

which, plugging in (1.4), results in:

$$\begin{aligned} dv(t,x) &= v_t dt + v_x\left(\kappa(\theta - X_t)dt + \lambda dW_t\right) + \frac{\lambda^2}{2}v_{xx}dt \\ &= (v_t + \kappa(\theta - X_t)v_x + \frac{\lambda^2}{2}v_{xx})dt + \lambda dW_t \end{aligned}$$

Since we defined $v(t,x)$ to have constant expectation (by the martingale property), and the Itô integral is zero in expectation, the drift term in the above expression must be null. However, since imposing that the drift term has to be zero corresponds exactly to the PDE (1.8), finding its solution is equivalent to finding the expression for v. The boundary condition to the PDE is obtained by noticing that X_T^2 is $\mathcal{F}_T$-measurable and can thus be taken out of the expectation.
An alternative derivation of (1.8) involves the use of the definition of generator as:

$$\lim_{h\to 0}\frac{\Gamma_h - 1}{h}u(t,x) = \mathcal{G}u(t,x),$$

where $\Gamma_{T-t}g(x) = \mathbb{E}[g(X_T)|\mathcal{F}_t]$. The derivation process is analogous.

We look for a solution of the form $v(t,x) = a(T-t) + b(T-t)x + c(T-t)x^2$. Differentiating:

$$\begin{cases} v_t = -a'(T-t) - b'(T-t)x - c'(T-t)x^2 \\ v_x = b(T-t) + 2c(T-t)x \\ v_{xx} = 2c(T-t) \end{cases}$$

Plugging the terms into the expression $v_t + \mathcal{G}v = 0$, one obtains:

$$-a'(T-t)-b'(T-t)x-c'(T-t)x^2+\kappa(\theta-x)\left[b(T-t)+2c(T-t)x\right]+\lambda^2 c(T-t)=0,$$

which means that:

$$\begin{cases} -a'(T-t)+\kappa\theta b(T-t)+\lambda^2 c(T-t)=0 \\ -b'(T-t)-\kappa b(T-t)+2\kappa\theta c(T-t)=0 \\ c'(T-t)-2\kappa c(T-t)=0 \end{cases} \tag{1.9}$$

with boundary conditions (motivated by the boundary conditions of (1.8)):

$$\begin{cases} a(0)=0 \\ b(0)=0 \\ c(0)=1 \end{cases}$$

The third equation of (1.9) implies that $c(T-t)=\exp\{-2k(T-t)\}$. The second one can then be rewritten as:

$$b'+\kappa b = 2\kappa\theta e^{-2\kappa(T-t)}$$

and has solution:

$$b(s)=\int 2\kappa\theta e^{-2\kappa s}e^{\int \kappa\, s\, ds}\, ds\, e^{-\int \kappa\, s\, ds} = 2\theta[e^{-ks}-e^{-2ks}]$$

As for the first equation:

$$-a'(T-t)+\kappa\theta\left(2\theta e^{-\kappa(T-t)}-2\theta e^{-2\kappa(T-t)}\right)+\lambda^2 e^{-2k(T-t)}=0,$$

we get a solution by integrating:

$$a(T-t)=a(0)+\theta^2+\frac{\lambda^2}{2k}-2\theta^2 e^{-\kappa(T-t)}+\left(\theta^2-\frac{\lambda^2}{2k}\right)e^{-2\kappa(T-t)}$$

Combining the components we obtain:

$$v(t,x)=\theta^2+\frac{\lambda^2}{2k}-2\theta^2 e^{-\kappa(T-t)}+\left(\theta^2-\frac{\lambda^2}{2k}\right)e^{-2\kappa(T-t)}+2\theta[e^{-k(T-t)}-e^{-2k(T-t)}]x+e^{-2\kappa(T-t)}x^2.$$

To attain $\mathbb{E}(X_T^2)$, we still need to take the expectation of the result just found:

$$\mathbb{E}(X_T^2)=\mathbb{E}[\mathbb{E}(X_T^2|\mathcal{F}_t)]=\mathbb{E}[\mathbb{E}(X_T^2|\mathcal{F}_0)]=\mathbb{E}(v(0,x))=v(0,x),$$

where the second equality follows from the martingale property of $v(t,x)$. The expressions obtained for $v(0,X_0)$ and $\mathbb{E}(X_T^2)$ eventually coincide.

1.3.3 CIR Process

The Cox, Ingersoll and Ross process for the variance in the Heston model can be readily derived from a Ornstein and Uhlenbeck process for the volatility. Indeed, consider $h_t = \sqrt{v_t}$ and a dynamics of OU type:

$$dh_t = -\beta h_t \, dt + \delta \, dW_t.$$

By Ito's lemma, with $v_t = h_t^2$, we can retrieve the dynamics of the variance as:

$$dv_t = 2h_t \, dh_t + dh_t^2 = (\delta^2 - 2\beta v_t)dt + \delta\sqrt{v_t}dW_t.$$

The stochastic differential equation defining the CIR process is hence obtained by setting:

- $\kappa = 2\beta$;
- $\theta = \frac{\delta^2}{2\beta}$;
- $\sigma = 2\delta$.

Three paths of the process are simulated by a forward discretisation scheme, and are presented in the following Figure 1.2, where the choice of parameters is $\kappa = 0.5$, $\theta = 0.04$, $\sigma = 2$, $v_0 = 0.04$ and $T = 1/12$.

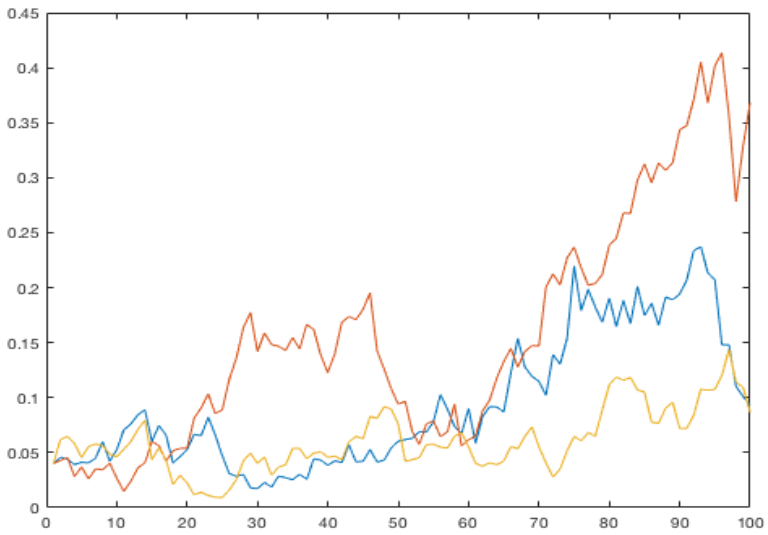


Figure 1.2: *Simulation of three paths of the CIR process.*

As there is no explicit solution to the SDE of the CIR process, we exploit the method of moments to retrieve the mean and the variance of the process.

The main features of the method are summed up in the following paragraph, preceded by some preliminary general definitions.
Consider a diffusion of type:

$$dX_t = b(X_t)dt + \sigma(X_t)dW_t$$

and a diffusion function:

$$a = \sigma\sigma^T.$$

Its generator is given by

$$\mathcal{G}f(x) = \frac{1}{2}\mathrm{Tr}(a(x)\nabla^2 f(x)) + b(x)^T\nabla f(x),$$

and is such that, by Ito's Lemma,

$$df(X_t) = \mathcal{G}f(X_t)dt + \nabla f(X_t)^T\sigma(X_t)dW_t$$

Definition 1.3.1. (Polynomial Generator)
The infinitesimal generator $\mathcal{G}$ of a diffusion X_t is polynomial if:

$$\mathcal{G}\mathrm{Pol}_n(\mathbb{R}^d) \subset \mathrm{Pol}_n(\mathbb{R}^d) \quad \forall n \in \mathbb{N}.$$

The diffusion X_t is said in this case to be a polynomial diffusion. □

Lemma 1.3.2. (Characterisation of Polynomial Generators)
Given an infinitesimal generator $\mathcal{G}$, the following are equivalent:

1. $\mathcal{G}$ is polynomial;
2. $a(x)$ and $b(x)$ are such that:

$$b \in \mathrm{Pol}_1(\mathbb{R}^d) \quad \text{and} \quad a \in \mathrm{Pol}_2(\mathbb{R}^d).$$

For scalar polynomial diffusions the requirement is hence that dX_t be of the form:

$$dX_t = (b + \beta X_t)dt + \sqrt{a + \alpha X_t + AX_T^2}dW_t.$$

Now consider $\{1, x, x^2, ..., x^N\}$ as a basis of $\mathrm{Pol}_N(\mathbb{R})$ and denote it as $H_N(x)$. This induces the representation of:

$$p(x) = \sum_{k=0}^{N} p_k x^k$$

as $\bar{p} = (p_0, ..., p_N)^T$.
Denote further as G the matrix representing the polynomial generator, i.e.:

$$\mathcal{G}H_N(x) = H_N(x)G_N.$$

Under the above notation the following theorem allows for the computation of moments in the case of polynomial diffusion.

Theorem 1.3.3. *For any $p \in Pol_N(\mathbb{R}^d)$, the moment formula holds:*

$$\mathbb{E}[p(X_T)|X_t] = H_N(X_t)e^{(T-t)G_N}\bar{p}$$

The result presented above can be used to retrieve the moments of the CIR process, which falls into the class of the polynomial diffusions. Note that there is no explicit solution to the SDE defining the CIR process, which is the reason why Theorem 1.3.3 is of use in this context.

First Moment

The generator associated to the CIR process can be written in matrix notation as:

$$G_N = \begin{pmatrix} 0 & \kappa\theta & 0 & 0 & \dots & 0 \\ 0 & -\kappa & 2\kappa\theta+\sigma^2 & 0 & & \\ 0 & 0 & -2\kappa & 3\kappa\theta+3\sigma^2 & \dots & 0 \\ \vdots & \vdots & 0 & & \ddots & \\ \vdots & \vdots & \vdots & & & n\kappa\theta + n(n-1)\frac{\sigma^2}{2} \\ 0 & 0 & 0 & \dots & & -n\kappa \end{pmatrix}.$$

The moment formula for the first moment can be written as:

$$\mathbb{E}[X_t|X_t] = (1, X_t)e^{(T-t)G}\begin{pmatrix}0\\1\end{pmatrix} = (1, X_t)e^{(T-t)\begin{pmatrix}0 & \kappa\theta\\ 0 & -\kappa\end{pmatrix}}\begin{pmatrix}0\\1\end{pmatrix}.$$

Rewriting

$$e^X = e^{\begin{pmatrix}0 & a\\ 0 & b\end{pmatrix}} = \sum_{j=0}^{\infty}\frac{X^j}{j!} = \begin{pmatrix}0 & \sum_{j=1}^{\infty}\frac{ab^{j-1}}{j!}\\ 0 & \sum_{j=0}^{\infty}\frac{b^j}{j!}\end{pmatrix} = \begin{pmatrix}0 & \frac{a}{b}(e^b-1)\\ 0 & e^b\end{pmatrix},$$

so thta plugging in the suitable values for a and b one obtains:

$$\mathbb{E}[X_t|X_t] = (1, X_t)\begin{pmatrix}0 & \frac{\kappa\theta}{-\kappa}(e^{-\kappa(T-t)}-1)\\ 0 & e^{-\kappa(T-t)}\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = \theta + (X_t-\theta)e^{-\kappa(T-t)}.$$

Going back to the original problem, one can then write the first moment of the variance following a Cox, Ingersoll and Ross model as:

$$\mathbb{E}[v_t|v_s] = \theta + e^{-\kappa(t-s)}(v_s-\theta).$$

We see that as κ goes to infinity, the first moment tends to the long-term mean θ, while for a very small κ it tends to stay at the current level of the variance.

Second Moment

Relying again on the moment formula one has:

$$\mathbb{E}[X_t|X_t] = (1, X_t, X_t^2)e^{(T-t)G}\begin{pmatrix}0\\0\\1\end{pmatrix} = (1, X_t)e^{(T-t)\begin{pmatrix}0 & \kappa\theta & 0\\ 0 & -\kappa & 2\kappa\theta+\sigma^2\\ 0 & 0 & -2\kappa\end{pmatrix}}\begin{pmatrix}0\\0\\1\end{pmatrix}.$$

Because of the polynomial representation, we are interested in the third column of the powers of the matrix in the exponent. For ease of computation, we shift to the notation:

$$e^X = e^{\begin{pmatrix}0 & d & e\\ 0 & a & b\\ 0 & 0 & c\end{pmatrix}} = \sum_{j=0}^{\infty}\frac{X^j}{j!}.$$

Starting from the d_j entry of the j-th power of X, one has:

$$d_j = d_{j-1}a = d_{j-2}a^2 = ... = d_1 a^{j-1},$$

while as far as the e_j entry of the j-th power of X is concerned we can write:

$$\begin{aligned}
e_j &= d_{j-1}b + e_{j-1}c = d_{j-1}b + (d_{j-2}b + e_{j-2}c)c\\
&= bd_{j-1} + bcd_{j-2} + bc^2d_{d-3} + ... + bc^{j-2}d_1 + c^{j-1}e_1\\
&= b\sum_{l=0}^{j-2} d_{j-l-1}c^l\\
&= b\sum_{l=0}^{j-2} da^{j-l-2}c^l = bda^{j-2}\sum_{l=0}^{j-2}\left(\frac{c}{a}\right)^l\\
&= bda^{j-2}\frac{1-\left(\frac{c}{a}\right)^{j-1}}{1-\left(\frac{c}{a}\right)} = \frac{dba}{a-c}\left(1-\left(\frac{c}{a}\right)^{j-1}\right)a^{j-2}.
\end{aligned}$$

In order to retrieve the entry relative to the sum of the powers of the matrix, we compute:

$$\begin{aligned}
\sum_{j=2}^{\infty}\frac{e_j}{j!} &= \sum_{j=2}^{\infty}\frac{\frac{dba}{a-c}\left(1-\left(\frac{c}{a}\right)^{j-1}\right)a^{j-2}}{j!}\\
&= \frac{dba}{a-c}\left(\frac{1}{a^2}(e^a-1-a) - \frac{1}{ac}(e^c-1-c)\right)\\
&= \frac{db}{a(a-c)}(e^a-1-a) - \frac{db}{c(a-c)}(e^c-1-c),
\end{aligned}$$

which, plugging in the value of the matrix G_2, results in:

$$\begin{aligned}\sum_{j=2}^{\infty}\frac{e_j}{j!} &= \frac{2\kappa^2\theta^2+\sigma^2\kappa\theta}{-\kappa^2}(e^{-\kappa(T-t)}-1+\kappa)-\frac{2\kappa^2\theta^2+\sigma^2\kappa\theta}{-2\kappa^2}(e^{-2\kappa(T-t)}-1+2\kappa)\\ &= (1-e^{-\kappa(T-t)})\left(2\theta^2+\frac{\sigma^2\theta}{\kappa}\right)-\theta^2(1-e^{-2\kappa(T-t)})-\frac{\sigma^2\theta}{2\kappa}(1-e^{-2\kappa(T-t)})\\ &= \theta^2(1-2e^{-\kappa(T-t)}+e^{-2\kappa(T-t)})+\frac{\sigma^2\theta}{2\kappa}(1-2e^{-\kappa(T-t)}+e^{-2\kappa(T-t)}),\end{aligned}$$

which corresponds to the constant term of the conditional second moment. One proceeds similarly to obtain the degree-one term.

$$\begin{aligned}b_j &= a_{j-1}b+b_{j-1}c = ba^{j-1}+(ba_{j-2}+b_{j-2}c)c\\ &= ba_{j-1}+bca_{j-2}+bc^2a_{j-3}+...+bc^{j-1}\\ &= b\sum_{l=0}^{j-1}c^la^{j-l-1}\\ &= ba^{j-1}\sum_{l=0}^{j-1}\left(\frac{c}{a}\right)^l = ba^{j-1}\frac{1-\left(\frac{c}{a}\right)^j}{1-\left(\frac{c}{a}\right)} = \frac{ba^j}{a-c}\left(1-\left(\frac{c}{a}\right)^j\right),\end{aligned}$$

which, plugging in the value of the matrix G_2, results in:

$$\begin{aligned}\sum_{j=1}^{\infty}\frac{b_j}{j!} &= \sum_{j=1}^{\infty}\frac{\frac{ba^j}{a-c}\left(1-\left(\frac{c}{a}\right)^j\right)}{j!} = \frac{2\kappa\theta+\sigma^2}{\kappa}\left(\sum_{j=1}^{\infty}\frac{a^j}{j!}-\sum_{j=1}^{\infty}\frac{c^j}{j!}\right)\\ &= \frac{2\kappa\theta+\sigma^2}{\kappa}(e^{-\kappa(T-t)}-1)-\frac{2\kappa\theta+\sigma^2}{\kappa}(e^{-2\kappa(T-t)}-1)\\ &= 2\theta(e^{-\kappa(T-t)}-e^{-2\kappa(T-t)})+\frac{\sigma^2}{\kappa}(e^{-\kappa(T-t)}-e^{-2\kappa(T-t)}).\end{aligned}$$

Lastly, considering the degree-two term one can see that:

$$c_j = c^j,$$

meaning that:

$$\sum_{j=0}^{\infty}\frac{c^j}{j!} = e^{-2\kappa(T-t)}.$$

Summing all the contributions, the result is that:

$$\begin{aligned}\mathbb{E}[X_T^2|X_t] = X_t^2e^{-2\kappa(T-t)}&+\\ &+X_t\left(2\theta+\frac{\sigma^2}{\kappa}\right)\left(e^{-\kappa(T-t)}-e^{-2\kappa(T-t)}\right)+\\ &+\left(\theta^2+\frac{\sigma^2\theta}{2\kappa}\right)\left(1-2e^{-\kappa(T-t)}+e^{-2\kappa(T-t)}\right).\end{aligned}$$

Lastly, combining the second and (the square of) the first moment, we can obtain an expression for the variance of the solution of the CIR stochastic differential equation.

$$\mathrm{Var}[v_t|v_s] = v_s\frac{\sigma^2}{\kappa}\left(e^{-\kappa(t-s)} - e^{-2\kappa(t-s)}\right) + \frac{\theta\sigma^2}{2\kappa}\left(1 - e^{-\kappa}(t-s)\right)^2.$$

We can conclude that as κ tends to infinity, the variance of the variance tends to zero. In addition, we had seen that in the same case the mean approached the long-term mean θ. This means that the uncertainty of the long-term value of the mean vanishes.
Lastly, for κ approaching zero, the variance of the variance tends to $\sigma^2 v_s(t-s)$.

Chapter 2

Option Pricing

This chapter covers the numerical methods used to generate the grid of prices which will be later generalised via a learning technique. The first section, 2.1 presents Monte Carlo methods, which are used to derive the price of Basket options for a set of values in the parameter space. Monte Carlo is here coupled with the variance reduction technique of the control variate. Section 2.2 is instead devoted to the presentation of American put options. Among the possible choices for the pricing of this financial product, this thesis describes finite differences: this approach is accurate and can be adapted to a typical formulation of the American options, the partial differential complementarity problem. After deriving the Heston partial differential equation in the previous chapter, we now encompass it in the framework of American options and describe its discretisation together with related numerical methods.

2.1 Basket Options

The first case study on which the methodology is tested involves basket European call options. As the name suggests, basket options are derivatives which depend on a set of underlying assets. For these financial products the payoff is given by:

$$\Psi(S(T)) = \max\left\{0, \sum_{i=1}^{d} \omega_i S_i(T) - K\right\},$$

where $S_i(T)$ is the price of stock i at maturity T. In our case, the number of stocks considered is $d = 5$. However, this thesis proposes, in the last chapters, an extension of the method to higher dimensions, with orders up to $d = 25$ considered.
The prices of this type of options can be computed by Monte Carlo (see Section 2.1.1 for a description of this method): by simulating the path of each of the underlying assets a large number of times one can estimate the expected future value of the payoff at maturity as the mean over all the simulation paths. By the no-arbitrage pricing formula, the price is then obtained as the discounted (estimated, in the case

of Monte Carlo) expectation:

$$v(t,x) = e^{-r(T-t)}\mathbb{E}^{t,x}\left[\Psi(S(T))\right] = e^{-r(T-t)}\mathbb{E}\left[\Psi(S(T))|S(t)=x\right].$$

In our practical case, t is taken to be equal to zero and $S(t=0)$ is given.

2.1.1 Monte Carlo Methods

Derivative pricing relies heavily on Monte Carlo methods, as prices of financial products can be represented as expectations, which can in turn be estimated by means of Monte Carlo techniques.
The use of Monte Carlo methods dates back to the Second World War, when methods relying on statistical sampling started being used in Los Alamos in order to forecast the explosive behaviour of fission arms by estimating the multiplication rate of neutrons. In general, Monte Carlo refers to all those methods which are based on stochastic simulations: the idea is to calculate the volume of a set by interpreting it as a probability.
The simplest example regards the estimate of π: by sampling N elements uniformly on the square $[-1,1]\times[-1,1]$, the probability of them being in the circle inscribed in the square is $\pi/4$. Hence, if we denote as $Z \sim Bin(N,\pi/4)$ the random variable describing the number of points falling into the circle, we have that, by the Law of Large Numbers:

$$\lim_{N\to\infty} \mathbb{P}(|\pi - 4Z/N| \geq \epsilon) = 0, \quad \forall \epsilon > 0.$$

By the same token, we can move from volumes to integrals, as follows. Consider

$$I_g = \int_0^1 g(x)\,dx :$$

this can be seen as the expectation of the random variable $g(X)$, with $X \sim Unif(0,1)$. If we can draw independently points on $[0,1]$ and evaluate g at such points, by taking the sample mean

$$\bar{I}_g = \frac{1}{N}\sum_{i=1}^{N} g(X_i), \quad \text{with} \quad \mathbb{E}(\bar{I}_g) = I_g,$$

integrability of g and the Law of Large Numbers imply that $\bar{I}_g$ converges to I_g with probability one as $N \to \infty$.
In addition, assuming square integrability of g and setting

$$\sigma_g^2 = \int_0^1 (g(x) - I_g)^2\,dx,$$

we can say that the error of the Monte Carlo estimate is is approximately normal with mean zero and standard deviation equal to $\sigma_g/\sqrt{n}$. An estimate of the quantity

σ_g, which is typically unknown, is given by the sample standard deviation:

$$s_g = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(g(X_i) - \bar{I}_g\right)^2}.$$

Notice that Slutsky's lemma implies that even with the sample estimate of the variance, the limiting distribution is still normal. As can be seen, the size of the error crucially depends on the sample size used the estimate the quantity of interest.
Convergence is of the order of $1/2$, which becomes competitive in higher dimensions. Indeed, while for dimension $d = 1$ it does not compare favourably against a trapezoidal rule ($O(n^{-2})$, $d = 1$), the latter becomes $O(n^{-2/d})$ in higher dimensions.

The Monte Carlo method finds frequent application in financial engineering, where it is used to estimate the price of derivative securities. This can be done by simulating sample paths of the stochastic process modelling the dynamics of the underlying asset price and hence obtain an estimate of the expectation of quantities which depend on the price paths, such as the price of a derivative.

Variance Reduction and Control Variates

Together with antithetic variates and stratified sampling, control variates are among the main techniques by which it is possible to increase the efficiency of Monte Carlo simulation. The idea of control variates is to make use of information about errors of known quantities in order to reduce the error made when estimating an unknown quantity.
Consider $Y_1, .., Y_n$ outputs from n independent replications of a simulation. If, parallel to each Y_i, we can compute X_i, whose expected value $\mathbb{E}[X]$ is known and such that the pairs $(X_i, Y_i), i = 1, ..., n$ are i.i.d, we can, for any fixed b, obtain the following:

$$Y_i(b) = Y_i - b(X_i - \mathbb{E}[X]).$$

The sample mean

$$\bar{Y}(b) = \frac{1}{n}\sum_{i=1}^{n}\{Y_i - b(X_i - \mathbb{E}[X])\}$$

is a control variate estimator of $\mathbb{E}[Y]$. It is unbiased since $\mathbb{E}[Y(b)] = \mathbb{E}[Y]$ and consistent, as:

$$\begin{aligned}\lim_{n\to\infty}\frac{1}{n}\sum_{i=1}^{n}Y_i(b) &= \lim_{n\to\infty}\frac{1}{n}\sum_{i=1}\{Y_i - b(X_i - \mathbb{E}[X])\}\\ &= \mathbb{E}[Y - b(X - \mathbb{E}[X])] = \mathbb{E}[Y],\end{aligned}$$

with probability one.
The variance of $Y(b)$ is

$$\mathrm{Var}[Y(b)] = \sigma_Y^2 + b^2\sigma_X^2 - 2b\sigma_Y\sigma_X\rho_{XY} := \sigma^2(b), \tag{2.1}$$

having denoted as σ_Y^2, σ_X^2 the variances of Y and X respectively and with ρ_{XY} their correlation. Hence, the variance of the control variate estimator is $\sigma^2(b)/n$, which has to be compared to σ_Y^2, if we want the new estimator to feature a higher efficiency with respect to the traditional one. In other words, we need to establish when $\sigma^2(b)$ is minimised with respect to the choice of b. The optimal value is given by:

$$b^* = \frac{\sigma_Y}{\sigma_X}\rho_{XY} = \frac{\mathrm{Cov}[X,Y]}{\mathrm{Var}[X]}.$$

Plugging back the result into (2.1), we obtain that the ratio of the variance of the control variate estimator to that of the traditional estimator follows:

$$\frac{\mathrm{Var}[\bar{Y}(b)]}{\mathrm{Var}[\bar{Y}]} = \frac{\sigma_Y^2 + \sigma_y^2\rho^2 - 2\sigma_Y^2\rho^2}{\sigma_Y^2} = 1 - \rho_{XY}^2.$$

The previous expression conveys the fact the variance reduction effect is the strongest as the control X is most correlated with Y. Furthermore, assuming that the computational time per replication does not change between X and Y, the speed-up offered by the control variate is, in terms of number of replications needed to obtain the same variance as n replications of the variable of interest, $(1-\rho^2)/n$.
As one can imagine, the value of b is most likely unknown, which means that one could replace it with its sample counterpart:

$$\hat{b}_n = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^n (X_i - \bar{X})}.$$

2.1.2 Basket Option Prices

The following section describes the computation of the price of a basket option of d correlated assets under the Black and Scholes model.
The first step is presented in Algorithm 1 and consists of the simulation of a number $NSim$ of random variables. Notice that `for` loops are shown for clarity, but the code can be implemented faster exploiting the ability of `Matlab` and `numpy` of handling matrices, so that loops become unnecessary.
The Choleski factor of the correlation matrix is used to introduce dependency between i.i.d $\mathcal{N}(0,1)$ random variables. Then, the algorithm returns the simulated prices (up to the multiplicative constant initial price) of the underlying assets stored in a matrix M, i.e.:

$$M_{i,j} = e^{\left(r - \frac{\sigma(j)^2}{2}\right)T + \sigma(j)x(j)\sqrt{T}},$$

where each row i refers to the simulation index and columns j to the asset in the basket.

Next, the price is obtained by means of a Monte Carlo technique as described in the following Algorithm 2.

Algorithm 1: Simulation of correlated geometric Brownian motions

Input: model and payoff parameters σ, Σ, T, r; number of simulations $NSim$.
Output: matrix M of size $(NSim, d)$ of simulated random variables.
Initialise
  $M \leftarrow$ `zeros`$(Nsim, d)$
  $L \leftarrow$ `choleski`(Σ)
end
For $iSim = 1, \ldots, NSim$
  $\epsilon \leftarrow$ vector of d independent normal random variables
  $x \leftarrow L\epsilon$
  For $iStock = 1, \ldots, d$
    $M(iSim, iStock) \leftarrow \exp\left\{\left(r - \frac{\sigma(iStock)^2}{2}\right)T + \sigma(iStock)x(iStock)\sqrt{T}\right\}$
  end
end
Return M

Algorithm 2: Computation of the basket option price

Input: M, $\mathbf{S}_0$, strike K, weights ω, r, T.
Output: price of the basket option.
Initialise
  $\pi \leftarrow$ `zeros`$(NSim)$
  $ctr \leftarrow$ `zeros`$(NSim)$
end
For $iSim = 1, \ldots, NSim$
  $R \leftarrow$ M(i,:)
  $S \leftarrow \mathbf{S}_0 \circ R$
  $\pi(iSim) \leftarrow \max\left\{\sum_{i=1}^{d} \omega_i S_i - K, 0\right\}$
  $ctr(iSim) \leftarrow \max\left\{\exp\left\{\left(\sum_{i=1}^{d} \omega_i \log S_i\right)\right\} - K, 0\right\}$
end
$\mu \leftarrow$ mean of the control
$sum \leftarrow \pi - (ctr - \mu)$
$p \leftarrow$ mean of sum
$P \leftarrow e^{-rT}p$
Return P

In the algorithm description the symbol $\circ$ was used to denote the Hadamard product.
Again, some steps of the algorithm, namely those inside the `for` loop, can be made more efficient by using matrix operations.
The algorithm uses a Monte Carlo estimation to compute the expectation of the payoff and introduces a control variable in order to implement the variance reduction technique discussed in the previous part.
The choice of the control variate stems from the similarity between basket and Asian

call options, to which this technique was first applied by Kemna and Vorst (1990). The latter type of derivative broadly consists of call options on the arithmetic average of the asset value over until maturity:

$$\pi = e^{-rT}\mathbb{E}\left[\max\left\{\sum_{i=1}^{d}\omega_{t_i}S_{t_i} - K, 0\right\}\right].$$

It is easy to see that a basket option has the same form as an Asian one when replacing the asset value over time with the value of the assets in the basket.
While the expectation of the arithmetic (weighted) average of log-normally-distributed random variables has no closed form, the geometric average is more easily tractable. Indeed, consider:

$$G = \left(\prod_{i=1}^{d} S_i^{\omega_i}\right)^{\frac{1}{\sum_{i=1}^{d}\omega_i}} = \exp\left\{\frac{1}{\sum_{i=1}^{d}\omega_i}\sum_{i=1}^{d}\omega_i \log S_i\right\} = \exp\left\{\sum_{i=1}^{d}\omega_i \log S_i\right\},$$

where the last equality refers to the case of interest, with weights adding up to one. In the special case where all the underlying assets have the same price the previous equation can be simplified further to obtain the arithmetic average. In general, however, geometric and average mean are close, making it sensible to use the geometric average as a control variate, since its expectation is known and the price can be computed, as is shown in the following.
The price is given by $P = \mathbb{E}[(G-K)^+]$, where K denotes the strike price and G can be rewritten as

$$\begin{aligned} G &= \exp\left\{\sum_{i=1}^{d}\omega_i \log S_i\right\} = \exp\left\{\sum_{i=1}^{d}\omega_i\left(\log S_{i0} + \left(r - \frac{\sigma_i^2}{2}\right)T + \sigma_i\sqrt{T}Z_i\right)\right\} \\ &= \exp\left\{\sum_{i=1}^{d}\omega_i\left(\tilde{\mu}_i + \tilde{\sigma}_i Z_i\right)\right\} = \exp\left\{\sum_{i=1}^{d}\omega_i W_i\right\}, \end{aligned}$$

where

$$\begin{gathered} \tilde{\mu}_i = \log S_{i0} + \left(r - \frac{\sigma_i^2}{2}\right)T; \\ \tilde{\sigma}_i = \sigma_i\sqrt{T}; \\ Z \sim \mathcal{N}(0,1) \quad i = 1, ..., d; \\ W_i \sim \mathcal{N}(\tilde{\mu}_i, \tilde{\sigma}_i^2) \quad i = 1, ..., d. \end{gathered}$$

As product of d lognormally distributed random variable, G is itself lognormal, and it can be seen as:

$$G = \exp\{W\}, \quad W \sim \mathcal{N}(\tilde{\mu}, \tilde{\sigma}^2),$$

with

$$\tilde{\mu} = \sum_{i=1}^{d}\omega_i\tilde{\mu}_i$$

$$\tilde{\sigma}^2 = \sum_{i=1}^{d}\sum_{j=1}^{d} \omega_i \tilde{\sigma}_i \rho_{i,j} \tilde{\sigma}_j \omega_j.$$

Having defined these quantities, one is ready to compute the price of the Asian geometric call option. One possible way to proceed is provided hereunder.

$$\begin{aligned}
P = \mathbb{E}[(G-K)^+] &= \int_{\mathbb{R}} (e^w - K)^+ \frac{1}{2\sqrt{\tilde{\sigma}^2}} e^{-\frac{(w-\tilde{\mu})}{2\tilde{\sigma}^2}}\, dw \\
&= \int_{\log K}^{\infty} (e^w - K) \frac{1}{2\sqrt{\tilde{\sigma}^2}} e^{-\frac{(w-\tilde{\mu})}{2\tilde{\sigma}^2}}\, dw \\
&= \int_{\log K}^{\infty} e^w \frac{1}{2\sqrt{\tilde{\sigma}^2}} e^{-\frac{(w-\tilde{\mu})}{2\tilde{\sigma}^2}}\, dw - K \int_{\log K}^{\infty} \frac{1}{2\sqrt{\tilde{\sigma}^2}} e^{-\frac{(w-\tilde{\mu})}{2\tilde{\sigma}^2}}\, dw \\
&= I_1 + I_2
\end{aligned}$$

The first integral can be computed as:

$$\begin{aligned}
I_1 &= \int_{\log K}^{\infty} e^w \frac{1}{2\sqrt{\tilde{\sigma}^2}} e^{-\frac{(w-\tilde{\mu})^2}{2\tilde{\sigma}^2}}\, dw = \frac{1}{2\sqrt{\tilde{\sigma}^2}} \int_{\log K}^{\infty} e^{-\frac{w^2 - 2w\tilde{\mu} + \tilde{\mu}^2 - 2\tilde{\sigma}^2 w}{2\tilde{\sigma}^2}}\, dw \\
&= \frac{1}{2\sqrt{\tilde{\sigma}^2}} \int_{\log K}^{\infty} e^{-\frac{w^2 - 2w(\tilde{\sigma}^2+\tilde{\mu}) + \tilde{\mu}^2}{2\tilde{\sigma}^2}}\, dw \\
&= \frac{1}{2\sqrt{\tilde{\sigma}^2}} \int_{\log K}^{\infty} e^{-\frac{(w-(\tilde{\sigma}^2+\tilde{\mu}))^2}{2\tilde{\sigma}^2}}\, dw\, e^{\frac{\tilde{\sigma}^2}{2}}\, e^{\frac{2\tilde{\sigma}^2\tilde{\mu}}{2\tilde{\sigma}^2}} \\
&= e^{\frac{\tilde{\sigma}^2}{2}+\tilde{\mu}} \mathbb{P}(W \geq \log K) = e^{\tilde{\mu}+\frac{\tilde{\sigma}^2}{2}} \phi\left(\frac{-\log K + (\tilde{\sigma}^2 + \tilde{\mu})}{\tilde{\sigma}}\right) \\
&= e^{\tilde{\mu}+\frac{\tilde{\sigma}^2}{2}} \phi\left(\frac{-\log K + \tilde{\mu}}{\tilde{\sigma}} + \tilde{\sigma}\right).
\end{aligned}$$

The second one can be solved in a similar fashion leading to:

$$I_2 = -K\phi\left(\frac{-\log K + \tilde{\mu}}{\tilde{\sigma}}\right).$$

To conclude the (not discounted) price of the Asian geometric call option- and hence the value of the mean of the control- is given by:

$$\mu = e^{\tilde{\mu}+\frac{\tilde{\sigma}^2}{2}} \phi\left(\frac{-\log K + \tilde{\mu}}{\tilde{\sigma}} + \tilde{\sigma}\right) - K\phi\left(\frac{-\log K + \tilde{\mu}}{\tilde{\sigma}}\right).$$

The mean of the control enters Algorithm 2 and allows the implementation of the variance reduction technique described above with geometric call options as control variates.

We now motivate graphically the choice of the Asian geometric option as a control variate, by verifying intuitively the properties which were previously mentioned.

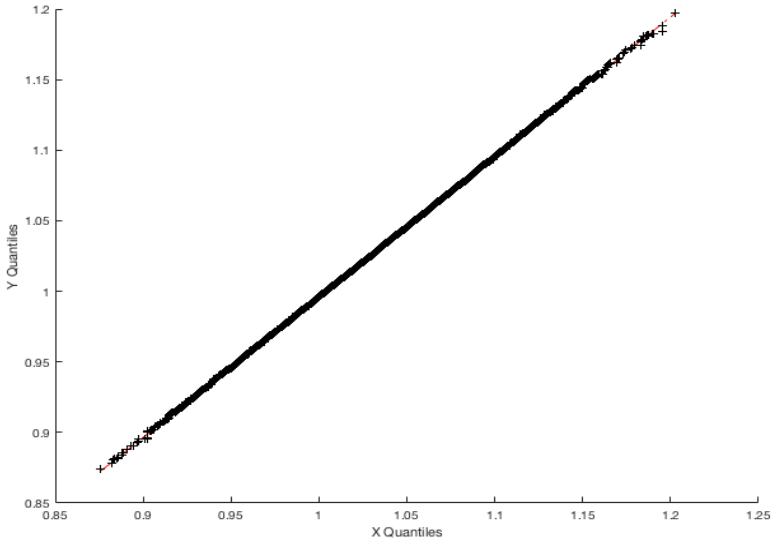


Figure 2.1: *Quantile-quantile plot of the arithmetic vs. geometric Asian options.*

Starting from Figure 2.1, the quantile-quantile plot shows that the distributions of geometric and arithmetic prices differ negligibly.

In addition, taking a look at the coefficients of interest we see that in a simulation performed with 1000 samples, both the correlation between the two prices and the b_n are close to one. Results are reported in Table 2.1.

Table 2.1: **Coefficients in the control variate experiment**

	Value
Correlation	0.9980
b_n	1.0340
Cov(P,G)	0.0012
Var(G)	0.0012

The given values allow for a simplification in the control variate formula, that of replacing b with one.

2.2 American Options

An American option differs from an European one in that it can be exercised at any time $\tau \leq T$ prior to expiry. If the holder of the option follows an optimal exercise strategy, the value of an American option at time τ is given by:

$$u(S,\tau) = \sup_{\tau \leq \tau' \leq T} \mathbb{E}\left[e^{-\int_\tau^{\tau'} r(\eta)\,d\eta}\Psi(S_{\tau'})|S_\tau = S\right],$$

i.e. the option value will be the highest among all the possible values in time before the expiry of the option.
Given the choice of an optimal stopping time, it follows that the value of an American option is never smaller than the current payoff and never smaller than the value of a corresponding European option.
To study the price of an American option consider for now the case of discrete exercise time, which is referred to as that of Bermudan options.
The value of the option at maturity offers no alternatives and is simply given by the payoff $\Psi(S_T)$. Conversely, at earlier time steps t_j, the holder can either wait, obtaining the discounted value of the option value in the next time step:

$$u(S,t_j) = \mathbb{E}\left[e^{-\int_{t_j}^{t_{j+1}} r(\eta)\,d\eta} u(S_{t_{j+1}}, t_{j+1})|S_{t_j} = S\right],$$

or exercise the option, with gain given by the current payoff $\Psi(S_{t_j})$. This implies that a rational holder of the option will pick the best among the two alternatives,

which results in:

$$u(S,t_j) = \max\left\{\Psi(S), \mathbb{E}\left[e^{-\int_{t_j}^{t_{j+1}} r(\eta)\,d\eta} u(S_{t_{j+1}}, t_{j+1})|S_{t_j} = S\right]\right\}.$$

This exercise strategy can be shown to be optimal and hence yields the Bermudan option price. This can be summed up by the following principle.

Lemma 2.2.1. *Under sufficient conditions on the model (strong Markov property) and on the payoff (integrability and regularity conditions), the Bermudan option price $u(S,t)$ satisfies the backward dynamic programming problem:*

$$\begin{cases} u(S,t_n) = \Psi(S) \quad t_n = T \\ u(S,t_j) = \max\left\{\Psi(S), \mathbb{E}\left[e^{-\int_{t_j}^{t_{j+1}} r(\eta)\,d\eta} u(S_{t_{j+1}}, t_{j+1})|S_{t_j} = S\right]\right\} \quad j < n. \end{cases}$$

Several ways have been implemented to solve this dynamic programming problem.
Longstaff and Schwartz have proposed a solution based on Monte Carlo methods, which consist of simulating paths of the underlying and approximating the value of the option via linear regression, with regressors given by basis functions of the underlying value.
An alternative is based on interpolation of the value function on a grid
A third possibility, and the one chosen in this work, involves deriving a PDE formulation and solving it by discretisation.

2.2.1 PDE Derivation

We consider the derivation of the PDE under the Heston model for the asset price dynamics. Of course, the same reasoning applies to e.g. the Black and Scholes case. As seen in Chapter 1, the Heston partial differential equation is given by:

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial S_t} rS_t + \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) + \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t + \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2} S_t^2 V_t + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t - ru = 0.$$

If one now denotes the spatial operator as $\mathcal{A}$:

$$\mathcal{A}u = \frac{\partial u}{\partial S_t} rS_t + \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) + \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t + \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2} S_t^2 V_t + \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t - ru$$

and applies a change of variable from time to time to maturity:

$$\tau = T - t$$

the PDE can be written as:

$$\frac{\partial u}{\partial \tau} - \mathcal{A}u = 0.$$

In the case of an American option we saw that

$$u(S,v,t_j) = \max\left\{\Psi(S), \mathbb{E}\left[e^{-\int_{t_j}^{t_{j+1}} r(\eta)\,d\eta} u(S_{t_{j+1}}, v_{t_{j+1}}, t_{j+1}) | S_{t_j} = S, v_{t_j} = v\right]\right\},$$

which means that:

$$u(S,v,t_j) \geq \mathbb{E}\left[e^{-\int_{t_j}^{t_{j+1}} r(\eta)\,d\eta} u(S_{t_{j+1}}, v_{t_{j+1}}, t_{j+1}) | S_{t_j} = S\right] = u^{EU}(S,v,t_j),$$

where the right-hand side corresponds to the payoff of a European option maturing at the next time instant with payoff $u(S_{t_{j+1}}, t_{j+1})$. Taking one step further we see that:

$$-\frac{u(S,v,t_{j+1}) - u(S,v,t_j)}{\Delta t} \geq -\frac{u(S,v,t_{j+1}) - u^{EU}(S,t_j)}{\Delta t},$$

and hence, in the limit for a small time step:

$$-\frac{\partial u}{\partial t}(S,v,t) \geq -\frac{\partial u^{EU}}{\partial t}(S,v,t) = \mathcal{A}u.$$

For one time step we thus have:

$$-\frac{\partial u}{\partial t}(S,v,t) - \mathcal{A}u \geq 0.$$

In addition, either the step is European, meaning that the holder of the option waits until next time, or the step is American, and the holder decides to exercise. This means that in the first case we have:

$$-\frac{\partial u}{\partial t}(S,v,t) - \mathcal{A}u = 0,$$

while in the second one the value is given by:

$$u(S,v,t) = \Psi(S).$$

The two cases give rise to the PDE

$$\left(-\frac{\partial u}{\partial t}(S,v,t) - \mathcal{A}u\right)(u(S,t) - \Psi(S)) = 0,$$

and, together with the aforementioned change of variable and a little abuse of notation (switching from τ to t):

$$\left(\frac{\partial u}{\partial t}(S,v,t) - \mathcal{A}u\right)(u(S,t) - \Psi(S)) = 0.$$

The American price hence satisfies the following conditions:

$$\begin{cases} \frac{\partial u}{\partial t}(S,v,t) - \frac{\partial u}{\partial S_t} rS_t - \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) - \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t - \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2} S_t^2 V_t - \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t + ru \geq 0 \\ u(S,t) \geq \Psi(S) \\ \left(\frac{\partial u}{\partial t}(S,t) - \mathcal{A}u\right)(u(S,t) - \Psi(S)) = 0 \\ u(S,0) = \Psi(S) \end{cases}, \tag{2.2}$$

which are sometimes referred to as the partial differential complementarity problem (PDCP) in the literature.

2.2.2 Finite Differences

Finite difference methods are aimed at finding a numerical approximation to the PDE solution on a bounded domain, obtained by means of truncation of the original space.
The starting point is the discretisation of the derivatives, which can be performed in several ways:

- forward schemes are such that

$$u'(x) \approx D_h^+ u(x) = \frac{u(x+h) - u(x)}{h};$$

- backward schemes for which:

$$u'(x) \approx D_h^- u(x) = \frac{u(x) - u(x-h)}{h};$$

- centred schemes for which:

$$u'(x) \approx D_{2h}^c u(x) = \frac{u(x+h) - u(x-h)}{2h}.$$

By taking a similar approach, one also obtains the second derivative:

$$u''(x) \approx D_h^+ D_h^- u(x) = \frac{u(x+h) - u(x) - (u(x) - u(x-h))}{h^2} = \\ = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}.$$

The PDE is hence discretised by taking small steps in space and in time, on a truncated domain: these points define a grid on which the value of the function is approximated.
Conditions are provided on the boundary of the domain in order to obtain approximated solutions: these can be of Dirichlet or Neumann type, if they are related to the unknown solution u or its spatial derivative, respectively.

2.2.3 Heston PDE Discretisation

Since the spatial domain is two-dimensional in the case of the Heston PDE (s and v), the domain considered is of the type $[0, S_{\max}] \times [0, V_{\max}]$ and four boundary conditions need to be provided, as the PDE is of order two.

Left boundary condition in S

The first Dirichlet condition can be derived by Feynman-Kac theorem, but is intuitively understood observing that for an initial value of zero, the underlying asset will always present a value equal to zero. Hence, the payoff will be equal to K at maturity for a put option.

$$u(0, v, t) = Ke^{-rt}.$$

Right boundary condition in S

$$\frac{\partial u}{\partial S}(S_{\max}, v, t) = 0.$$

The right Neumann condition follows from dividing the Heston PDE by S^2 and taking the limit for $S \to \infty$, resulting in an expression which is satisfied by the previous equation.

Left boundary condition in V

The following condition follows from inserting the boundary value $v = 0$ in the PDE.

$$rS_t \frac{\partial u}{\partial S_t}(S, 0, t) + \kappa\theta \frac{\partial u}{\partial V_t}(S, 0, t) - \frac{\partial u}{\partial t} - ru(S, 0, t) = 0$$

Right boundary condition in V

$$\frac{\partial u}{\partial v}(S, v_{\max}, t) = 0,$$

as when volatility approaches the limit, the price should be expected not to be sensitive to changes in volatility.
It has to be remarked that the boundary conditions chosen here are not the only possible ones. The Neumann conditions can for instance be replaced by Dirichlet conditions – although the latter choice can lead to poorer performances due to the act of fixing a value to the solution.

Truncated Problem

Having restricted ourselves to a truncated domain and having selected suitable conditions at the boundary, we are ready to discretise the following PDE.

$$\begin{cases} \frac{\partial u}{\partial t}(S,v,t) - \frac{\partial u}{\partial S_t} rS_t - \frac{\partial u}{\partial V_t}\kappa(\theta - V_t) - \frac{\partial^2 u}{\partial S_t \partial V_t}\rho\sigma S_t V_t - \frac{1}{2}\frac{\partial^2 u}{\partial S_t^2} S_t^2 V_t \; - \frac{1}{2}\frac{\partial^2 u}{\partial V_t^2}\sigma^2 V_t + ru = 0 \\ u(S,v,0) = \Psi(S) \\ u(0,v,t) = Ke^{-rt} \\ \frac{\partial u}{\partial S}(S_{\max},v,t) = 0 \\ rS_t \frac{\partial u}{\partial S_t}(S,0,t) + \kappa\theta \frac{\partial u}{\partial V_t}(S,0,t) - \frac{\partial u}{\partial t} - ru(S,0,t) = 0 \\ \frac{\partial u}{\partial v}(S,v_{\max},t) = 0. \end{cases}$$

Spatial Discretisation

The first direction of discretisation is the spatial one, concerning the value of the underlying asset and the volatility. We consider an evenly-spaced grid for both s and v, resulting in:

$$s_i = s_{\min} + ih_s = ih_s;$$

$$v_j = v_{\min} + jh_v = jh_v;$$

where h denotes the distance between two consecutive points on the grid. The corresponding values of the function u on the grid points (s_i, v_j) are then denoted as $u_{i,j}$.
We make use of a forward scheme for the internal (i.e. non-boundary) nodes in the S direction and of both forward and backward a scheme in the v direction. The double choice of scheme for the discretisation of the derivatives with respect to the volatility allows to encompass the boundary conditions into the finite-difference scheme.
As for the mixed derivative, the discretisation is performed by:

$$\frac{\partial^2 u}{\partial s \partial v} \approx \frac{u(s+h_s, v+h_v) - u(s+h_s, v-h_v) - u(s-h_s, v+h_v) + u(s-h_s, v-h_v)}{4h_s h_v}.$$

The result is that we obtain a system of differential equations in time, for each point in the $s \times v$ grid at time t.

$$\begin{aligned}\frac{\partial u_{i,j}}{\partial t}(t) = & rS_i \frac{u_{i+1,j} - u_{i,j}}{h_s} + \frac{1}{2} S_i^2 v_j \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_s^2} \\ & + \kappa(\theta - v_j) \frac{u_{i,j} - u_{i,j-1}}{h_v} \mathbb{1}_{v_j > \theta} + \kappa(\theta - v_j) \frac{u_{i,j+1} - u_{i,j}}{h_v} \mathbb{1}_{v_j < \theta} \\ & + \frac{1}{2} \sigma^2 v_j \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_v^2} \\ & + \rho\sigma S_i v_j \frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{4h_s h_v} \\ & - r u_{i,j}.\end{aligned}$$

Unlike the Black and Scholes case, which only presents a spatial discretisation along one variable, the Heston model features a two-dimensional grid, which makes the system of differential equation harder to be represented. Indeed, the discretised option value cannot fit in a vector, but requires a matrix, or the vectorisation of the matrix following some ordering of the variables.
We can compactly represent the system of ODEs using the notation:

$$\frac{\partial U}{\partial t} = A(t)U(t) = (A_0(t) + A_1(t) + A_2(t))U(t) + F,$$

where the subscripts relative to the matrices A account for the differentiation along different directions and F denotes the terms which are added on top of the finite differencing, due to the border effects. Specifically, we will denote in the following:

- A_0 as the operator of finite differencing relative to the mixed terms;
- A_1 as the operator of finite differencing along the S space;
- A_2 as the operator of finite differencing along the v space.

The implementation of the discretisation of the PDE is executed in `Matlab`. Notice that the A matrices are tridiagonal, as the finite differences schemes only involve the current, previous and next node on the grid. Hence, using the sparse format provided by `Matlab` allows for a significant reduction in memory usage.

Time Discretisation

After obtaining what is usually defined as a semi-discrete scheme, we can attain a fully-discrete one by performing discretisation along the temporal dimension. Typical choices are a forward or explicit Euler, a backward or implicit Euler or a θ scheme combining the two.
The result is a grid for time as well as for the underlying asset price and for volatility,

so that grid points are indexed by $u^m_{i,j}$.
The choice of the explicit Euler leads to:

$$\begin{aligned}\frac{u^{m+1}_{i,j}-u^m_{i,j}}{dt} &= rS_i\frac{u^m_{i+1,j}-u^m_{i,j}}{h_s}+\frac{1}{2}S_i^2v_j\frac{u^m_{i+1,j}-2u^m_{i,j}+u^m_{i-1,j}}{h_s^2}\\ &+\kappa(\theta-v_j)\frac{u^m_{i,j}-u^m_{i,j-1}}{h_v}\mathbb{1}_{v_j>\theta}+\kappa(\theta-v_j)\frac{u^m_{i,j+1}-u^m_{i,j}}{h_v}\mathbb{1}_{v_j<\theta}\\ &+\frac{1}{2}\sigma^2v_j\frac{u^m_{i,j+1}-2u^m_{i,j}+u^m_{i,j-1}}{h_v^2}\\ &+\rho\sigma S_iv_j\frac{u^m_{i+1,j+1}-u^m_{i+1,j-1}-u^m_{i-1,j+1}+u^m_{i-1,j-1}}{4h_sh_v}\\ &-ru^m_{i,j},\end{aligned}$$

allowing to move forward in time starting from the initial condition, which is the payoff in our case.
Collecting the values of u into the desired vector U (either one or two dimensional), the scheme can be compactly represented as:

$$U^{m+1}=\left(I+dt(A_0^m+A_1^m+A_2^m)\right)U^m+dtF^m.$$

By a similar reasoning one obtains the implicit Euler scheme, which retrieves the value of the solution in the next time node as the solution to the system:

$$\begin{aligned}\frac{u^{m+1}_{i,j}-u^m_{i,j}}{dt} &= rS_i\frac{u^{m+1}_{i+1,j}-u^{m+1}_{i,j}}{h_s}+\frac{1}{2}S_i^2v_j\frac{u^{m+1}_{i+1,j}-2u^{m+1}_{i,j}+u^{m+1}_{i-1,j}}{h_s^2}\\ &+\kappa(\theta-v_j)\frac{u^{m+1}_{i,j}-u^{m+1}_{i,j-1}}{h_v}\mathbb{1}_{v_j>\theta}+\kappa(\theta-v_j)\frac{u^{m+1}_{i,j+1}-u^{m+1}_{i,j}}{h_v}\mathbb{1}_{v_j<\theta}\\ &+\frac{1}{2}\sigma^2v_j\frac{u^{m+1}_{i,j+1}-2u^{m+1}_{i,j}+u^{m+1}_{i,j-1}}{h_v^2}\\ &+\rho\sigma S_iv_j\frac{u^{m+1}_{i+1,j+1}-u^{m+1}_{i+1,j-1}-u^{m+1}_{i-1,j+1}+u^{m+1}_{i-1,j-1}}{4h_sh_v}\\ &-ru^{m+1}_{i,j},\end{aligned}$$

Similar to what was done before, the compact representation of the scheme is:

$$(I-dtA^{m+1})U^{m+1}=U^m+dtF^{m+1},$$

which corresponds to several systems of equations, whose solution can be computed efficiently in `Matlab`.
Lastly, the θ method is commonly used and consists of a linear combination of the explicit and implicit Euler.

$$\begin{cases}U^{m+1}=U^m+dt(A_0^m+A_1^m+A_2^m)U^m+dtF^m\\ U^{m+1}=U^m+dt(A_0^{m+1}+A_1^{m+1}+A_2^{m+1})U^{m+1}+dtF^{m+1}\end{cases}$$

$$\begin{cases} U^{m+1} = U^m + dtA^m U^m + dtF^m \\ U^{m+1} = U^m + dtA^{m+1} U^{m+1} + dtF^{m+1}, \end{cases}$$

so that the solution at the next time step satisfies:

$$U^{m+1} = U^m + (1-\theta)dt(F^m + A^m U^m) + \theta dt(F^{m+1} + A^{m+1} U^{m+1}).$$

In other words, one needs to find the solution to the system:

$$(I - \theta dtA^{m+1})U^{m+1} = (I + (1-\theta)dtA^m)U^m + dt(\theta F^{m+1} + (1-\theta)F^m). \tag{2.3}$$

The good properties of the system are preserved, as all the matrices are tridiagonal. Notice that when taking $\theta = 1$ it is possible to recover the implicit Euler method, whereas $\theta = 0$ leads back to the explicit case. In addition, $\theta = 1/2$ goes by the name of Crank-Nicolson.
Having found a discretisation of the PDE, we can now go back to the original overall problem – presented in (2.2). Simplifying further the notation introduced in (2.3) we can write:

$$\begin{cases} BU - F \geq 0 \\ U - \Psi \geq 0 \\ (BU - F)^T (U - \Psi) = 0, \end{cases} \tag{2.4}$$

having set

$$\begin{cases} B = I - \theta dtA^{m+1} \\ F = (I + (1-\theta)dtA^m)U^m + dt(\theta F^{m+1} + (1-\theta)F^m) \end{cases}$$

The system (2.4) is commonly referred to as linear complementarity problem (LCP) and has to be solved for each time instant in the time grid, starting from the initial condition. Various solution procedures exist, including iterative methods such as projected Jacobi or projected SOR or direct solutions, such as Thomas algorithm for tri-diagonal systems and Brennan-Schwartz's.

2.2.4 ADI Schemes

The term ADI stands for Alternating Direction Implicit Method, as the technique allows to split the finite differencing operator so that the scheme is alternately explicit in one direction and implicit in the other one. These splitting schemes have been shown to be efficient, stable and robust when using numerical techniques to price European-style options via multi-dimensional PDE, see Haentjens and in 't Hout (2015).
The ADI schemes evolve from the θ method with the exception that the mixed derivatives are treated in an explicit way, so that the term A_0 only involves the approximation at the previous time step.

In the following derivation assume that the dt factor is already accounted for in the A matrices. In addition, we denote as $\hat{F}$ the boundary terms, i.e.:

$$\hat{F} = dt(\theta F^{m+1} + (1-\theta)F^m).$$

$$(I - \theta A_1 - \theta A_2)U^{m+1} = (I + A_0 + (1-\theta)A_1 + (1-\theta)A_2)U^m + \hat{F}$$

$$\begin{aligned}(I - \theta A_1 - \theta A_2 + \theta^2 A_1 A_2)U^{m+1} =& (I + A_0 + (1-\theta)A_1 + (1-\theta)A_2 + \theta^2 A_1 A_2)U^m + \\ &+ \theta^2 A_1 A_2(U^{m+1} - U^m) + \hat{F}\end{aligned}$$

$$\begin{aligned}(I - \theta A_1)(I - \theta A_2)U^{m+1} = \hat{A}U^{m+1} &= \\ &= (I + A_0 + (1-\theta)A_1 + (1-\theta)A_2 + \theta^2 A_1 A_2)U^m + \\ &\quad + \theta^2 A_1 A_2(U^{m+1} - U^m) + \hat{F} \\ &\approx (I + A_0 + (1-\theta)A_1 + A_2)U^m - \theta A_2(I - \theta A_1)U^m + \hat{F}\end{aligned}$$

In the last line, the approximation follows from the fact that the term

$$\theta^2 A_1 A_2(U^{m+1} - U^m)$$

is infinitesimal of higher order with respect to the other terms. Next, we rewrite the previous expression as:

$$(I - \theta A_1)[(I - \theta A_2)U^{m+1} + \theta A_2 U^m] \approx (I + A_0 + (1-\theta)A_1 + A_2)U^m + \hat{F}$$

We thus have:

$$\begin{cases}(I - \theta A_1)\, Y = (I + A_0 + (1-\theta)A_1 + A_2)\, U^m + \hat{F} \\ (I - \theta A_2)\, U^{m+1} = Y - \theta A_2 U^m,\end{cases}$$

which is best known as Douglas-Rachford method.
Further extensions of the method are shown to provide increased stability. In particular, the Hundsorfer and Verwer (HV) method proposes the following steps to compute the solution:

$$\begin{cases}(I - \theta A_1)\, Y_1 = (I + A_0 + (1-\theta)A_1 + A_2)\, U^m + \hat{F} \\ (I - \theta A_2)\, Y_2 = Y_1 - \theta A_2 U^m \\ (I - \theta A_1)\, Y_3 = (I - \theta A_1)\, Y_1 + \theta[A_0 + (1-\theta)A_1](Y_2 - U^m) \\ (I - \theta A_2)U^{m+1} = Y_3 - \theta A_2\, U^m\end{cases}$$

As can be seen, the differencing operators are here decoupled, so that one starts by solving the system implicitly in the S direction and makes an explicit step in the v direction and exchanges the roles in the following phase of the algorithm. Iterating these procedures one can show that greater stability is attained.

2.2.5 Matrix Construction

The current section details the construction of the matrices used to solve the finite-differences problem:

$$\begin{aligned}\frac{u_{i,j}^{m+1}-u_{i,j}^{m}}{dt} &= rS_i\frac{u_{i+1,j}^{m+1}-u_{i,j}^{m+1}}{h_s}+\frac{1}{2}S_i^2v_j\frac{u_{i+1,j}^{m+1}-2u_{i,j}^{m+1}+u_{i-1,j}^{m+1}}{h_s^2}\\ &+\kappa(\theta-v_j)\frac{u_{i,j}^{m+1}-u_{i,j-1}^{m+1}}{h_v}\mathbb{1}_{v_j>\theta}+\kappa(\theta-v_j)\frac{u_{i,j+1}^{m+1}-u_{i,j}^{m+1}}{h_v}\mathbb{1}_{v_j<\theta}\\ &+\frac{1}{2}\sigma^2v_j\frac{u_{i,j+1}^{m+1}-2u_{i,j}^{m+1}+u_{i,j-1}^{m+1}}{h_v^2}\\ &+\rho\sigma S_iv_j\frac{u_{i+1,j+1}^{m+1}-u_{i+1,j-1}^{m+1}-u_{i-1,j+1}^{m+1}+u_{i-1,j-1}^{m+1}}{4h_sh_v}\\ &-ru_{i,j}^{m+1}.\end{aligned}$$

The choice made in this work consists of representing the values of the solution at each time instant on a two-dimensional grid of dimensions $n_s \times n_v$. Notice that an alternative encoding of the solution could involve the vectorisation of the solution matrix, which is simple in two dimensions. The operations are performed on an inner grid of dimension $(n_s-2)\times(n_v-1)$: the remainder of the matrix is retrieved from the boundary conditions. Rows account for the S grid, while each column corresponds to one value of v. As a result, the structure of the solution at each time step is of the form:

$$U^m=\begin{pmatrix}U^m(S=0,v=0) & U^m(S=0,v=h_v) & \dots & U^m(S=0,v=n_vh_v)\\ U^m(S=h_s,v=0) & U^m(S=h_s,v=h_v) & \dots & \vdots\\ U^m(S=2h_s,v=0) & \ddots & & \\ \vdots & & \ddots & \\ \vdots & & & \\ U^m(S=n_sh_s,v=0) & \dots & & U^m(S=n_sh_s,v=n_vh_v)\end{pmatrix}.$$

S Direction

Matrix A_1 deals with differencing along the S direction. Such matrix is tri-diagonal and can thus be stored in sparse format.
In the following, subscripts 1,2 and 3 refer to the lower co-diagonal, main diagonal and upper co-diagonal respectively; matrix B refers to terms multiplied by the volatility; the other terms are stored in C. The first and the last element of the

diagonal is treated differently, as it has to take into account the boundary terms.

$$B = \begin{pmatrix} -\frac{S_2^2}{h_s^2} & \frac{S_2^2}{2h_s^2} & 0 & \dots & & \dots & 0 \\ \frac{S_3^2}{2h_s^2} & -\frac{S_3^2}{h_s^2} & \frac{S_3^2}{2h_s^2} & 0 & \dots & \dots & 0 \\ 0 & \frac{S_4^2}{2h_s^2} & -\frac{S_4^2}{h_s^2} & \frac{S_4^2}{2h_s^2} & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \frac{S_{end-2}^2}{2h_s^2} & -\frac{S_{end-2}^2}{h_s^2} & \frac{S_{end-2}^2}{2h_s^2} \\ 0 & \dots & & & 0 & \frac{S_{end-1}^2}{2h_s^2} & -\frac{S_{end-1}^2}{2h_s^2} \end{pmatrix}.$$

$$C = \begin{pmatrix} -r\frac{S_2}{h_s} - \frac{r}{2} & r\frac{S_2}{h_s} & 0 & & \dots & & 0 \\ 0 & -r\frac{S_3}{h_s} - \frac{r}{2} & r\frac{S_3}{h_s} & 0 & \dots & \dots & 0 \\ 0 & 0 & -r\frac{S_4}{h_s} - \frac{r}{2} & r\frac{S_4}{h_s} & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & & & 0 & -r\frac{S_{end-2}}{h_s} - \frac{r}{2} & r\frac{S_{end-2}}{h_s} \\ 0 & \dots & & \dots & & 0 & -\frac{r}{2} \end{pmatrix}.$$

The matrix A_1 hence results from:

$$A_1 = dt(vB + C).$$

The matrix A_1 will then multiply each column of the matrices U^m. In other words, it will apply the finite difference operator for each value of v. However, one also has to consider the boundary conditions mentioned in the previous subsection. That is, for $v = 0$:

$$rS_t\frac{\partial u}{\partial S_t}(S, 0, t) + \kappa\theta\frac{\partial u}{\partial V_t}(S, 0, t) - \frac{\partial u}{\partial t} - ru(S, 0, t) = 0$$

the matrix A_1 has a different form:

$$A_1(v=0) = \begin{pmatrix} -r\frac{S_2}{h_s} - \frac{r}{2} & r\frac{S_2}{h_s} & 0 & & \dots & & 0 \\ 0 & -r\frac{S_3}{h_s} - \frac{r}{2} & r\frac{S_3}{h_s} & 0 & \dots & \dots & 0 \\ 0 & 0 & -r\frac{S_4}{h_s} - \frac{r}{2} & r\frac{S_4}{h_s} & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & & & 0 & -r\frac{S_{end-2}}{h_s} - \frac{r}{2} & r\frac{S_{end-2}}{h_s} \\ 0 & \dots & & \dots & & 0 & -\frac{r}{2} \end{pmatrix},$$

up to the factor dt.
Observe that the boundary condition is enforced for the term in the entry $n_s - 2$.
The S direction also contributes to the F term as follows:

$$F(2,j) = -\frac{1}{2h_s^2} K e^{-rT} S^2 v_j \quad j = 2, ..., n_v - 1.$$

This corresponds to the boundary correction relative to S_2.
Lastly, notice again that the size of the matrix A_1 is $n_s - 2 \times n_s - 2$, as the matrix only determines the non-boundary nodes. The other ones ($S = 0$ and $S = n_s$) are found from the boundary conditions.

v Direction

In order to work along the v direction one needs to transpose U^m so that the matrix acts on columns which are then the grid points v for each value of S. Matrix D accounts for the second-order differencing:

$$D = \begin{pmatrix} 0 & 0 & 0 & & \dots & & 0 \\ \frac{\sigma^2 v_2}{2h_v^2} & -\frac{\sigma^2 v_2}{h_v^2} & \frac{\sigma^2 v_2}{2h_v^2} & 0 & \dots & \dots & 0 \\ 0 & \frac{\sigma^2 v_3}{2h_v^2} & -\frac{\sigma^2 v_3}{h_v^2} & \frac{\sigma^2 v_3}{2h_v^2} & 0 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & & & \frac{\sigma^2 v_{end-2}}{2h_v^2} & -\frac{\sigma^2 v_{end-2}}{h_v^2} & \frac{\sigma^2 v_{end-2}}{2h_v^2} \\ 0 & \dots & & \dots & & \frac{\sigma^2 v_{end-1}}{2h_v^2} & -\frac{\sigma^2 v_{end-1}}{2h_v^2} \end{pmatrix}.$$

The first row of D, which consists of zeros, accounts for the boundary conditions, which imply that there is no second-order differencing for $v = 0$.
The first-order differences are instead operated by the matrix F, whose columns are split due to space constraints. $F =$

$$\begin{pmatrix} -\frac{\kappa\theta}{h_v} - \frac{r}{2} & \frac{\kappa\theta}{h_v} & 0 & \dots & & \\ -\frac{\kappa(\theta - v_2)}{h_v}\mathbb{1}_{v_2 > \theta} & \frac{\kappa(\theta - v_2)}{h_v}(\mathbb{1}_{v_2 > \theta} - \mathbb{1}_{v_2 \leq \theta}) - \frac{r}{2} & \frac{\kappa(\theta - v_2)}{h_v}\mathbb{1}_{v_2 \leq \theta} & 0 & & \\ \vdots & & & \ddots & \ddots & \\ 0 & \dots & & & & \\ 0 & \dots & & \dots & 0 & -\frac{r}{2} \end{pmatrix} \dots$$

$$\begin{pmatrix} \dots & 0 & 0 \\ \dots & 0 & 0 \\ \ddots & 0 & 0 \\ \ddots & 0 & 0 \\ -\frac{\kappa(\theta - v_{end-2})}{h_v}\mathbb{1}_{v_{end-2} > \theta} & \frac{\kappa(\theta - v_{end-2})}{h_v}(\mathbb{1}_{\dots} - \mathbb{1}_{\dots}) - \frac{r}{2} & \frac{\kappa(\theta - v_{end-2})}{h_v}\mathbb{1}_{v_{end-2} \leq \theta} \\ 0 & -\frac{\kappa(\theta - v_{end-1})}{h_v}\mathbb{1}_{v_{end-1} > \theta} & \frac{\kappa(\theta - v_{end-1})}{h_v}(\mathbb{1}_{\dots} - \mathbb{1}_{\dots}) - \frac{r}{2} \end{pmatrix}.$$

The matrix A_2 hence results from:

$$A_2 = dt(D + F).$$

Mixed terms

The mixed terms are treated differently, as the matrix representation of the solution makes it more difficult to perform differences along the two dimensions. For this reason, the mixed derivatives approximations are computed from the solution at the previous time step in a manual way, without actually constructing the matrix A_0: this is possible as the mixed terms are dealt with only explicitly, and not implicitly.

2.2.6 Solution

Starting from the initial condition, namely the payoff of a put option, the approximated solution to the differential equation is built following the ADI-DR scheme described in the previous section. In addition, the following boundary considerations have to be made:

- the last row of the solution matrix, for $S = s_{n_s h_s}$, is taken to be equal to the previous row, due to the Neumann condition on $S_{\max}$;
- the last column of the solution matrix, for $v = v_{n_v h_v}$, is taken to be equal to the previous column, due to the Neumann condition on $v_{\max}$;

- the first row of the solution matrix, for $S = 0$ is simply the discounted strike, due to the boundary condition.

Furthermore, one has to move from the plain PDE to the PDCP (2.2), i.e. implement the American step of the option. This can be done by means of the introduction of an auxiliary matrix λ_n, of size $(n_s - 2) \times (n_v - 1)$, i.e. that of the inner grid.
At each time step, we switch from:

$$(I - \theta dt A^{m+1})U^{m+1} = (I + (1-\theta)dtA^m)U^m + dt(\theta F^{m+1} + (1-\theta)F^m)$$

to:

$$(I - \theta dt A^{m+1})U^{m+1} = (I + (1-\theta)dtA^m)U^m + dt(\theta F^{m+1} + (1-\theta)F^m) + dt\lambda,$$

with the LCP becoming:

$$\begin{cases} (I - \theta dt A^{m+1})U^{m+1} = (I + (1-\theta)dtA^m)U^m + dt(\theta F^{m+1} + (1-\theta)F^m) + dt\lambda^{m+1} \\ \lambda^{m+1} \geq 0 \\ U^{m+1} \geq \Psi \\ (U^{m+1} - \Psi) \circ \lambda^{m+1} = 0 \end{cases} . \tag{2.5}$$

In the approximate numerical solution, following the approach proposed by Ikonen and Toivanen (2008), we solve:

$$\begin{cases} (I - \theta dt A^{m+1})\bar{U}^{m+1} = (I + (1-\theta)dtA^m)\hat{U}^m + dt(\theta F^{m+1} + (1-\theta)F^m) + dt\bar{\lambda}^{m+1} \\ \hat{U}^{m+1} - \bar{U}^{m+1} - dt(\hat{\lambda}^{m+1} - \bar{\lambda}^{m+1}) = 0 \\ \hat{\lambda}^{m+1} \geq 0 \\ \hat{U}^{m+1} \geq \Psi \\ (\hat{U}^{m+1} - \Psi) \circ \hat{\lambda}^{m+1} = 0 \end{cases} ,$$

where $\hat{U}^m$ and $\hat{\lambda}^m$ define successive approximations to the U^m and λ^m presented in (2.5). After obtaining $\bar{U}^m$ as the solution to the system of ODEs and $\bar{\lambda}^m = \hat{\lambda}^{m-1}$, one can retrieve the approximations $\hat{U}^m$ and $\hat{\lambda}^m$ from:

$$\hat{U}^m = \max\{\bar{U}^m - dt\bar{\lambda}^m, \Psi\}$$

and

$$\hat{\lambda}^m = \max\{0, \bar{\lambda}^m + (\Psi - \bar{U}^m)/dt\},$$

applying the maximum component-wise.
The solution to the PDPC is presented in Figure 2.2.
The algorithm for the computation of the price of an American option is summed up in 3.

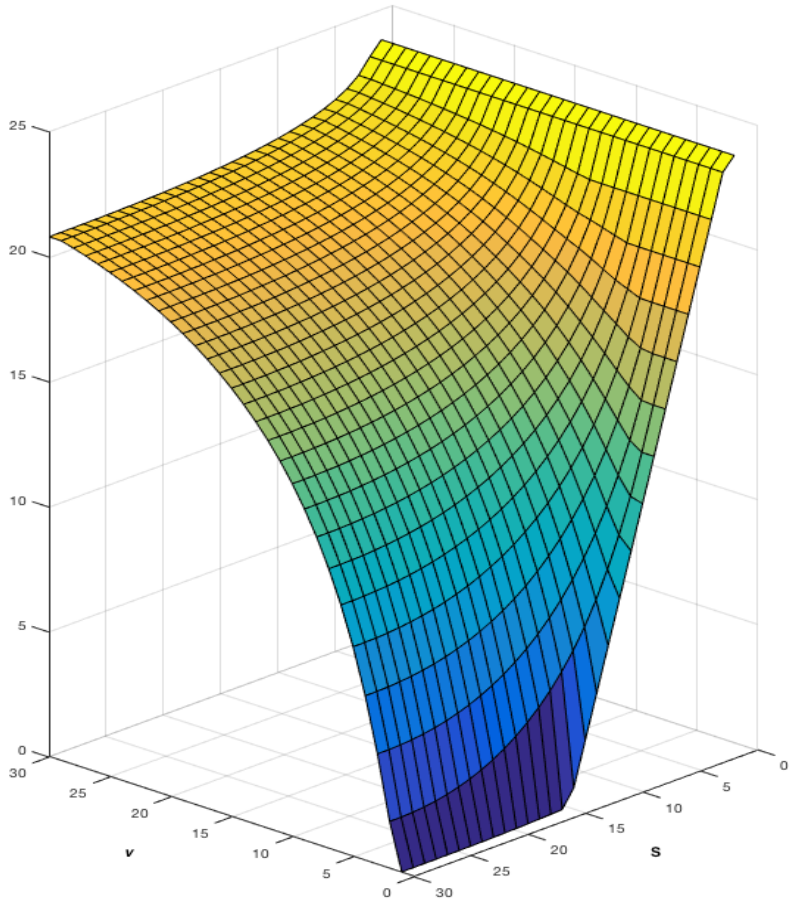


Figure 2.2: *Numerical solution to the Heston PDE.*

Algorithm 3: Approximated Solution to the Heston PDE for the American Put option problem

Input:
model parameters: σ, κ, θ, ρ, r
payoff parameters: T, K
discretisation parameters for S: n_s, $S_{\max}$
discretisation parameters for v: n_v, $v_{\max}$
discretisation parameters for time: n_t, $T_{\max}$
Output: matrix f of size (n_s, n_v) of prices on the grid.
Initialise
$s \leftarrow$ evenly-spaced points on $[0, S_{\max}]$
$sgrid \leftarrow$ inner grid of s
$v \leftarrow$ evenly-spaced points on $[0, v_{\max}]$
$vgrid \leftarrow$ inner grid of v
$u_{\text{init}} \leftarrow$ European put payoff on the inner grid
$u_0 \leftarrow u_{\text{init}}$
$\lambda \leftarrow$ `zeros` on inner grid
$A_1 \leftarrow$ differencing operator along S
$A_2 \leftarrow$ differencing operator along v
$F \leftarrow$ boundary correction
$f \leftarrow \texttt{zeros}(n_s \times n_v)$
end
For $n = 1, \ldots, n_t$
  For $j = 1, \ldots, n_v - 1$
    $f_1(:, j) = A_1(j) u_0(:, j)$
  end
  For $i = 1, \ldots, n_s - 2$
    $f_2(i, :) = (A_2 u_0(i, :)^T)^T$
  end
  $Y_0 = u_0 + A_0 u_0 + f_1 + f_2 + F dt$
  $u_1 = Y_0 - \theta f_1$
  $Y_1 \leftarrow$ solve the system $(I - \theta A_1) Y_1 = u_1$
  $u_2 = Y_1 - \theta f_2$
  $Y_2 \leftarrow$ solve the system $(I - \theta A_2) Y_2 = u_2$
  $u_0 = \max\{Y_2 - dt\lambda, u_{\text{init}}\}$
  $\lambda = \max\{0, \lambda + (u_{\text{init}} - Y_2)/dt\}$
end
f(inner grid) $\leftarrow_0$
f(boundary) $\leftarrow$ as described in 2.2.6
Return f

Chapter 3

Neural Networks

Neural networks aim to mirror the human brain and are loosely inspired by neuroscience. Although developments in this field come from mathematics and engineering rather than neuroscience, the idea is to construct a structure consisting of units, the neurons, whose function presents some analogies with the neurons of the human brain but without the claim to explain the functioning of the brain. Rather, as Ian Goodfellow puts it, the goal is to design *function approximation machines* that are capable of achieving *statistical generalization* [Goodfellow, Bengio and Courville (2016)].
The advantage of this approach is generality: as we will see, the good approximating function is learnt with the data and the engineer does not have to constrain him/herself to some specific functional form.

3.1 Structure

Neural Nets have received increasing attention in the last years, due to their representation power. Consider a standard regression task, with training set $S_t = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$: we know that a linear combination of the input features in unlikely to predict outputs well, due to a large bias. Feature augmentation is a typical solution to overcome this problem: by adding polynomials of the regressors, for instance, one might improve performance of classifiers or predictors. However, this can quickly lead to overfitting, as well as to an increase in the computational cost needed to fit the model. In addition, one does not know *a priori* what the good features of the model are and resorting to domain experts can be not feasible or too restrictive an approach. Neural networks, instead, allow to learn the features as well as the weights of the classifier at the same time.

The structure of a simple neural net is presented in Figure 3.1. Such network consists of one *input* layer, L *hidden* layers and one *output* layer. Each layer consists in turn of different nodes, which are D, K and m for the input, hidden and output nodes, respectively, in the figure. Notice that a value k is not known beforehand

and can be different for each of the hidden nodes. The number of input nodes corresponds to the size of the features space, i.e. the number of components of $\mathbf{x}$, while there are as many output nodes as the dimension of y. A last remark concerns the value of L: while simple nets can present one or few hidden layers, deep neural networks can even feature hundreds of them. Theoretically, good values for all of the hyper-parameters mentioned should be chosen via validation.
As the computation is performed from left to the right, from input nodes to output nodes in the direction of the arrows, the network is called a feedforward network. The notation adopted in this thesis is as follows: $x_j^{(i)}$ denotes node j in layer i.
The network presented in Figure 3.1 is fully-connected, meaning that each node in layer l is connected to each node in layer $l+1, l = 0, , , L-1$ and each node in the last hidden layer is connected to nodes in the output layer.
Connections take place via weighted edges, where weights from node i in layer $l-1$ to node j in node l are denoted as $w_{i,j}^{(l)}$. Specifically, we can write each node of a layer as the result of the action of a function ϕ, called *activation* function, on an affine combination of the nodes of the previous layer, i.e.:

$$x_j^{(l)} = \phi\left(\sum_{i=1}^{K} w_{i,j}^{(l)} x_i^{l-1} + b_j^{(l)}\right),$$

where $b_j^{(l)}$ is the bias term relative to the j-th node in layer l.
While the parameters w and b are part of the learning process, which occurs when training the neural network, the activation function is chosen in the design of the neural network. In this respect, it is crucial that ϕ present a nonlinearity at some point in the hidden layers, or there would be no gain in the representation power of the neural net with respect to a linear regression.
The representation power comes indeed from the hidden layers, which can be seen as a mapping:

$$\Phi : \mathbb{R}^D \to \mathbb{R}^K,$$

from the D-dimensional input space to an artificial K-dimensional one which leads to better, in some sense which has to be specified, outputs. In other words, this corresponds to the feature selection and feature augmentation phases which lead to a suitable representation of the data and would otherwise have to be performed in a regression or classification task.

Whatever the machine learning task one is interested in, how well the algorithm can learn any function $f(x)$ of the input features is of paramount importance. In particular, we claimed that neural nets are extremely powerful in representing functions, and this can be stated rigorously, under mild assumptions on the structure of the net and on the domain of interest. The following lemma is due to Barron (Barron (1993)).

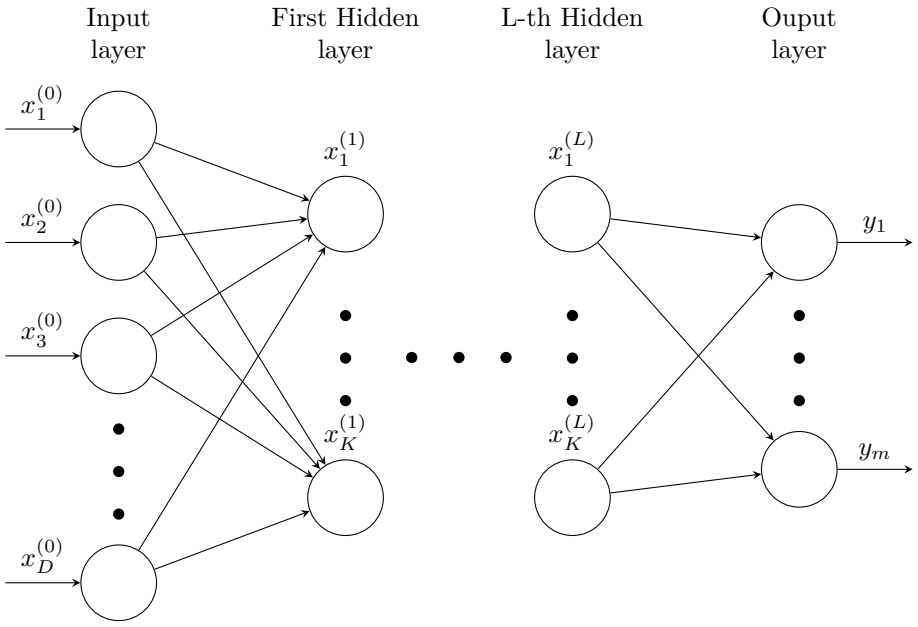


Figure 3.1: *A simple neural net, fully connected and with* L *hidden layers.*

Lemma 3.1.1. *Let $f : \mathbb{R}^D \to \mathbb{R}$ be a function such that*

$$\int_{\mathbb{R}^D} |\omega| \tilde{f}(\omega)\, d\omega \leq C,$$

where

$$\tilde{f}(\omega) = \int_{\mathbb{R}^D} f(x) e^{-i\omega x}\, dx$$

is the Fourier Transform of f.
Then, for all $n \geq 1$ there exists a function f_n of the form:

$$f_n(\boldsymbol{x}) = \sum_{j=1}^{n} c_j \phi(\boldsymbol{x}^T w_j + b_j) + c_0,$$

where ϕ is a sigmoid-like function, such that:

$$\int_{|\boldsymbol{x}| \leq r} (f(\boldsymbol{x}) - f_n(\boldsymbol{x}))^2\, dx \leq \frac{(2Cr)^2}{n}.$$

The lemma ensures that under a smoothness condition, the function f can be approximated well enough (in a L^2 sense) on a bounded domain by a neural net with one hidden layer with n nodes. As an aside, sigmoid-like refers to functions which have 0 as a left limit and 1 as the one and are sufficiently smooth.

3.2 Hidden Units

Although the design of hidden units is currently being investigated in research, there exist choices which are widely used. To determine which one to use in practice mostly comes down to trial and error.
What needs to be stressed is that many of the activation functions are not differentiable at all the points of the domain: this mostly causes no troubles, as the minimum of the loss function is hardly ever attained. Some strategies however exist to avoid this issue.

Rectified Linear Units

Rectified Linear Units (ReLU) follow the principle that linear objects are easy to optimise and consist of the activation function:

$$\phi(x) = \max\{0, x\}.$$

The fact that the second derivative is zero a.e. implies that this choice provides a useful learning direction, without second-order effects. Conversely, learning is not possible for cases in which the activation is zero. To make up for this problem generalisations exist to the ReLU so that the gradient exists everywhere.
One of these, the Leaky ReLU is defined by:

$$\phi(x) = \max\{0, x\} + \alpha \min\{0, x\},$$

with α small, for instance 0.01.
Alternatively, absolute value rectification is used in contexts where features are invariant under polarity reversal, such as image recognition:

$$\phi(x) = \max\{0, x\} - \min\{0, x\} = |x|.$$

As a last example, maxout units divide the space into groups and produce as output the maximum in each group, allowing to learn a piecewise linear function. Together with the ability of learning convex functions, maxout units also lead to fewer parameters as the following layer will present fewer parameters by a factor corresponding to the size of the groups.

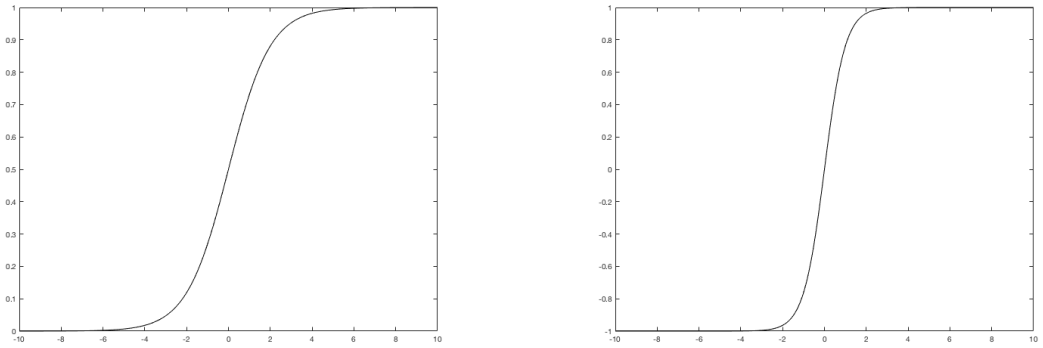
Sigmoid Function

The sigmoid function is well known in the literature and is given by:

$$\sigma(x) = \frac{e^x}{1 + e^x}.$$

The main drawback of this choice is that the sigmoidal units saturate across the domain, only being sensitive to the input when this is close to zero, and this might cause problems when using gradient-based methods.

An alternative to the sigmoid function is the hyperbolic tangent, which is related



(a) Sigmoid Function (b) Hyperbolic Tangent Function

Figure 3.2: *Plots of the sigmoid and hyperbolic tangent activation functions.*

to the sigmoid function by:

$$\tanh(x) = 2\sigma(2x) - 1.$$

Figure 3.2 highlights the similar behaviour of the two functions.

No Activation

While complete absence of activation functions will constrain the neural net to be linear, and hence to eliminate the need for the network altogether, some hidden layers can in principle be linear. Having linear nodes can lead to a significant saving in terms of the number of parameters, by introducing a low-rank structure.
Consider as an example a layer of size n with p outputs: it is represented by an $n \times p$ matrix of parameters. If that is replaced by two linear layers, the first one with q outputs, the total number of parameters will be $n \times q + q \times p = (p + n)q$, which, depending on how small q is, can be much smaller than $n \times p$.

3.3 Training

Interestingly, the previous lemma makes the learning problem that of learning a function rather than a set of parameters. The main drawback which comes with increased representation power is the loss of convexity in the most used loss functions. This means that traditional convex optimisation algorithms are of no use in the training of neural nets and one usually turns to iterative gradient-descent methods, such as stochastic gradient descent. However, it is possible to compute gradients in an efficient and exact way, which makes optimisation of the loss function feasible in practice: this is done via the back-propagation algorithm. The nodes evolve as described in the previous sections:

$$x^{(l)} = f^{(l)}(x^{(l-1)}) = \phi\left((W^{(l)})^T x^{(l-1)} + b^{(l)}\right),$$

with the activation function ϕ being applied element-wise to the vector of nodes.
As a result, the output will be produced by:

$$y = f(x^{(0)}) = f^{(L+1)} \circ f^{(L)} \circ \ldots \circ f^{(2)} \circ f^{(1)}(x^{(0)}).$$

Taking the mean-square loss as a widely-used example of loss function we will have a cost given by:

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} \left(y_n - f^{(L+1)} \circ f^{(L)} \circ \ldots \circ f^{(2)} \circ f^{(1)}(x_n^{(0)})\right)^2.$$

Now consider stochastic gradient descent: at each iteration, the weights and bias terms of a randomly-chosen term are translated by a fraction of the gradient of the loss, so as to move in the direction of its steepest descent. One thus need to compute the gradient with respect to one sample of the loss $\mathcal{L}_n$, i.e.:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} \quad \text{and} \quad \frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}}.$$

We define two quantities $z^{(l)}$ and $\delta_j^{(l)}$ and the associated *forward* and *backward passes*:

- the output of layer $l-1$ before applying the activation function and is computed by moving forward in the network in the so-called forward pass

$$z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)};$$

- the partial derivatives

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}},$$

which are obtained applying a backward pass, starting from the output and moving backward along the network. This operation corresponds to:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \delta_k^{(l+1)} W_{j,k}^{(l+1)} \phi'(z_j^{(l)}),$$

which can be written in vector form as:

$$\delta^{(l)} = (W^{(l+1)} \delta^{(l+1)}) \circ \phi'(z^{(l)}),$$

denoting with $\circ$ the Hadamard element-wise product.

Having these quantities at hand, the original derivatives result easily from:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_k \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$$

and

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)},$$

noticing that $w_{i,j}$ only affects the j-th z entry.
The back propagation algorithm follows from the steps described above and is summarised in Algorithm 4.

Computational cost

The computational cost is for the most part due to matrix-vector multiplication, both in the forward and in the backward pass. This corresponds to $\mathcal{O}(w)$ for w entries in the weight matrix. The memory cost instead refers to the need to store the input to the nonlinearity of the hidden layers, with a cost of $\mathcal{O}(mn_h)$, for m number of samples in the minibatch and n_h number of hidden units.

3.4 Regularisation

Regularisation deals with the importance of learning without overfitting. The goal is that the good performance of the algorithm on a training set is matched with an

Algorithm 4: Back propagation

Input: y_n, x_n for some $n \in \{1, ..., N\}$.
Output: the gradient of the loss.
Initialise
$x^{(0)} \leftarrow x_n$
end
For $l = 1, \dots, L+1$
$z^{(l)} \leftarrow (W^{(l)})^T x^{(l-1)} + b^{(l)}$
$x^{(l)} \leftarrow \phi(z^{(l)})$
end
$\delta^{(L+1)} = -2(y_n - x^{(L+1)})\phi'(z^{(L+1)})$
For $l = L, \dots, 1$
$\delta^{(l)} \leftarrow (W^{(l+1)}\delta^{(l+1)}) \circ \phi'(z^{(l)})$
end
For i, j, l
Return $\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$
Return $\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}$,
end

equally good performance on a different test set. In other words, when given fresh data as input, the algorithm should produce good results: if that were not the case, the algorithm would simply be following too closely the training data (overfitting), thus losing in generality. Regularisation can be defined as *any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error* [Goodfellow, Bengio and Courville (2016)]: this is usually implemented via constraints- soft or hard- on the parameters.
In deep learning, regularisation mainly comes down to attaining a good trade off between bias and variance, with a view to move close to the true data generating process: given the complexity of the domains deep learning applies to, a good model is often a large model with adequate regularisation.
Controlling how complex a model is can be done in several ways. The usual strategy is to limit model capacity by adding a term to the cost function:

$$\tilde{\mathcal{L}}(\theta; X, y) = \mathcal{L}(\theta; X, y) + \lambda\Omega(\theta),$$

where λ is a hyperparameter regulating the weight of the penalty. In turn, the penalty Ω is a norm term. Notice that in neural nets the bias term is left unregularised, as, unlike the wrights $W_{i,j}$, the b_is only concern one term, and are thus less subject to increased variance because one does not need to observe the interaction of two variables in different conditions. In addition, regularising the bias terms is more likely to introduce significant bias in the model.

L^2 Regularisation

Best known as ridge regression, the L^2 regularisation features a norm-two regularising term:

$$\tilde{\mathcal{L}}(\omega; X, y) = \mathcal{L}(\omega; X, y) + \frac{\lambda}{2}\omega^T\omega,$$

which induces the SGD update:

$$\omega \leftarrow \omega - \alpha(\lambda\omega + \nabla_\omega \mathcal{L}(\omega; X, y)),$$

meaning that:

$$\omega \leftarrow (1 - \alpha\lambda)\omega - \alpha\nabla_\omega \mathcal{L}(\omega; X, y), \tag{3.1}$$

where we indicate with ω the set of parameters but for the bias terms. The previous expression (3.1) motivates the phenomenon referred to as the weight decay, whereby the weight shrinks at each iteration. To study it further, we consider an approximation of the loss function by expanding it around its argmin as:

$$\hat{\mathcal{L}}(\omega) \approx \mathcal{L}(\omega^*) + \frac{1}{2}(\omega - \omega^*)^T H_{\mathcal{L}}(\omega - \omega^*),$$

where we have no first-order term due to the first-order conditions.
To find the optimum $\tilde{\omega}$ of the regularised approximate loss we take the gradient and obtain:

$$\lambda\tilde{\omega} + H(\tilde{\omega} - \omega^*) = 0,$$

$$(H + \lambda I)\tilde{\omega} = H\omega^*.$$

As H is real and symmetric, we exploit the decomposition $H = Q\Lambda Q^T$, which leads to:

$$(Q\Lambda Q^T + \lambda I)\tilde{\omega} = (Q\Lambda Q^T \lambda I)\omega^*$$

$$\tilde{\omega} = [Q(\Lambda + \lambda I)Q^T]^{-1}(Q\Lambda Q^T)\omega^*$$

$$\tilde{\omega} = Q(\Lambda + \lambda I)^{-1}\Lambda Q^T\omega^*,$$

which corresponds to rescaling ω^* along the eigenvectors v_i of H by a factor of:

$$\frac{\lambda_i}{\lambda + \lambda_i}.$$

In other words, the shrinking effect will be higher the more $\lambda > \lambda_i$ and weight decay will be stronger in the directions where the eigenvalue of the Hessian are smaller. That is, directions which contribute more to the reduction of the loss suffer less from the weight decay.

Strong and Weak Constraints

Other forms of regularisation are the L^1 regularisation, which presents a sparsity property acting as a feature selection procedure; the generalised Lagrangian, written as:

$$\tilde{\mathcal{L}}(\omega; X, y) = \mathcal{L}(\omega; X, y) + \alpha(\Omega(\omega) - k).$$

An alternative to the latter consists of explicitly having a constraint, solving the unconstrained problem and then projecting to the nearest point satisfying the constraint, thus removing the need to optimise on α as well as on ω. Removing the soft constraint can be of use when training neural nets because of the issue of *dead units*: it might be the case that non-convex optimisation methods be stuck in local minima corresponding to small ω, which in turn result in dead units of the neural net. In these cases, the function being learnt by the MLP will not be affected by these units, while it might be the case that larger weights might help to escape the local minimum. Hard constraints and projections will only enforce the weights to become smaller in cases where their magnitude is rising outside the constraint. Lastly, when coupled with high learning rates, explicit constraints and projection can allow for rapid exploration of the space of parameters without giving up on stability. This is the case as large weights lead to large gradients and to the descent becoming unstable.

Dataset Augmentation

This approach is performed where one can easily obtain new observations by modifying existing ones, for instance in image recognition. In this context, when training a classifier, generalisation is easily achieved by translating pixels, rotating or flipping images: this does not change the target class one needs to classify and provides more data. Of course this relies on the need of the classifier to be invariant with respect to some transformations.

Early Stopping

Early stopping allows to detect the point where validation error starts increasing despite a decreasing training error: parameters estimates are stored every time validation error goes down, and the algorithm stops if after a given set of iterations the validation error does not improve, returning the last stored parameters, rather than the newest ones.
As a form of hyper-parameter tuning, the training time is less costly to set as it does not require to try different values to be tried in different runs of the algorithm, but can be determined in one single run. Furthermore, the requirements in terms of memory only regard storing intermediate best parameters; in terms of overhead, the original training algorithm is not subject to modifications, but for the introduction of a validation phase which can however be performed in parallel.
It can be shown that early stopping plays the same role as weight decay in L^2

regularisation: given a number of training steps τ and a learning rate ϵ, $\epsilon\tau$ can be seen as the effective capacity, i.e. the volume of parameter space the algorithm can explore. Early stopping has the advantage that, unlike weight decay, the hyper-parameter value does not have to be selected via multiple runs of the algorithm.

Noise Robustness

Similarly, one can introduce noise in the observation inputs, but it proves less effective in the case of neural networks. Noise can equally be applied to the weights, in a Bayesian fashion attributing a distribution to each of them.

Dropout

Dropout can be seen as an alternative to bagging, which quickly becomes unfeasible due to the cost of training multiple deep neural nets. Dropout arises by training all possible subnetworks obtained by ignoring units from the original net, which can be practically done by removing the hidden layers' units by zero. Training with dropout occurs by mini-batch learning and each time a sample enters the minibatch, a mask μ is sampled for the network, whereby each neuron is assigned either a zero or a one value: if the mask marks a neuron as zero, it is not included in the training phase. Typical probabilities of accepting a node are 0.8 for inputs and 0.5 for hidden layers As dropouts tends to make weights larger, when evaluating the overall net, the weights are rescaled by the accepting probability. Alternatively, the output is a weighted average of the output with weights given by the probability of the mask. Dropout makes learning more robust by making the architecture noisy. It reduces the risk that units adapt to compensate for mistakes of other units, in a way that works for a given training set but might not for other ones.

3.5 Optimisation

Machine learning differs substantially from standard optimisation. To begin with, the goal of learning is a performance measure which is targeted indirectly, via a loss function. By reducing the cost function $\mathcal{L}$, one aims at improving the performance measure, but $\mathcal{L}$ is not a goal in itself, as is the case with optimisation.
In addition the ultimate goal of learning would be to minimise the cost over the distribution p_{data} from which the data are generated, but one settles for the empirical distribution $\hat{p}_{\text{data}}$ of the data implied by the training set.
One thus shifts from looking for the parameters minimising:

$$\mathbb{E}_{x,y\sim p_{\text{data}}}[L(f(x;\theta),y)]$$

to those relative to:

$$\mathbb{E}_{x,y\sim \hat{p}_{\text{data}}}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L\left(f(x^{(i)};\theta),y^{(i)}\right),$$

with m denoting the number of observations in the training data.
However, overfitting and the difficulties arising when trying to minimise some types of loss functions are the major reason why an objective function can still differ slightly from the empirical risk and one considers a surrogate loss function.
One last element by which learning and optimisation differ is the timing the algorithm stops. Optimisation can stop at local minima, while learning typically relies on stopping criteria.
One key aspect in learning is that empirical quantities are obtained by means of a restricted sample of the data, giving rise to mini-batch or stochastic algorithms; in the latter case, one single observation is used each time. In particular, mini-batch are favoured with respect to the batch case (entire data) or stochastic (one single datum), for a series of reasons:

- larger datasets yield more accurate results, but returns are less than linear;
- one-item batches imply under-utilisation of multi-core architectures;
- some kinds of hardwares are optimised to work with give sizes of the data (e.g. powers of 2);
- mini-batches lead to a regularising effect, in all likelihood ensured by the process of adding noise that sampling the batch provides;
- parallel architectures can exploit clever decomposition in batches such that different updates over different examples can be computed at the same time.

However, the higher variance in the gradient values demands a lower learning rate to preserve stability.
The main issues optimisation faces when dealing with neural networks are presented in the following paragraphs.

Ill conditioning of the Hessian

When making a gradient descent step, the change in the value of the loss is approximated by

$$\mathcal{L}(x) - \mathcal{L}(x_0) = \mathcal{L}(x_0 - \epsilon g) - \mathcal{L}(x_0) \approx \epsilon g^T g + \frac{1}{2}\epsilon^2 g^T H g, \tag{3.2}$$

denoting the gradient as g, the Hessian as H and the learning rate as ϵ. As ill conditioning of the Hessian is a measure of how big is the differences between second derivatives, poor conditioning means that the descent step might be occurring in the direction with largest increase of the derivative. The second-order term can cause the loss to grow if it is too large, thus hampering the learning process since the learning rate has to be kept small to prevent overshooting. What can be done to overcome this issue is to monitor the norms of the first-order and of the second-order terms.

Local Minima, Saddle Points and Cliffs

Local Minima have long been thought to represent a major issue in neural nets optimisation. Today, it seems that the case for many high-cost local minima is no longer to be made as practitioners tend to agree that local minima are typically associated with a low cost. Baldi and Hornik (1993) found for autoencoders that, if nonlinearities are absent from the model, local minima do not have higher costs than global minima, claiming that the results hold for neural networks with nonlinearities.
In multi-dimensional non-convex functions, saddle points present zero gradient besides local minima, and, interestingly, the ratio of saddle points to local minima increases with the dimension of the problem, as studied by De Spiegeleer, Madan, Reyners and Schoutens (2018). The same study claims that a significant amount of saddle points present very high cost. At the same time, however, gradient descent seems capable of escaping saddle points in empirical trials, see for instance Goodfellow, Vinyals and Saxe (2015), who study the case with an *ad hoc* example: this is also a possible reason why (stochastic) gradient descent has traditionally been preferred with respect to Newton's method, which targets a zero gradient.
Cliffs represent a third obstacle for gradient optimisation: these correspond to regions with large steepness, which can result in the parameters' point being pushed extremely far away due to the interaction of very large weights. The gradient clipping heuristic is designed so that the step size is reduced when the descent step the algorithm is attempting becomes too large.

3.5.1 Parameters Initialisation

Choosing the initial parameters for training neural nets is an active area of research. Solutions are often heuristic and intuitive as the subject is still to be completely understood: one difficulty being faced is that some points which can enhance the success of the optimisation process are of detriment when it comes to generalisation. What seems certain is the so-called breaking the symmetry principle, according to which different units with the same activation and the same incoming nodes need to have different starting points, or will result in the same updates. This principle is good practice even in the case of stochasticity in the algorithm, for instance with dropout, and motivates the use of random initialisation. The normal distribution is a typical choice, as well as the uniform one, but the magnitude of the initial points has consequences: larger weights contribute more to the symmetry-breaking principle mentioned before and avoid the dispersion of signal, but their side effect is the possibility of exploding values, linked to the so-called chaos. This phenomenon is such that weights might be so large that the behaviour of the net appears random when slightly different inputs result in entirely different outputs. Another aspect to be taken into account is regularisation, which would in principle dictate the need for smaller weights. Excessively small weights, however, may cause activations to shrink when moving forward through the neural net.
In terms of the mean, a sensible choice can be represented by zero, hinting at a

prior which assumes that units do not interact. A possible heuristic is then to pick weights uniformly in $(-1/\sqrt{m}, 1/\sqrt{m})$, with m input nodes; another one, proposed by Glorot and Bengio (2010), is related to the desired property of all layers having the same activation variance and all layers having the same gradient variance at initialisation and yields weights initialised as:

$$W_{i,j} \sim \text{Unif}\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right),$$

where m and n are input and output nodes respectively.
Interestingly, however, it is often not the case that these criteria enhance performance, due to several possible reasons. The properties imposed in the initialisation phase might not be desirable in the first place. Alternatively, they might be desirable, but they fail to be preserved throughout the training phase. Another possibility is that they do improve optimisation, but at the cost of reduced generalisation.
Biases initialisation requires fewer considerations, as biases can be without fear of undesired effects be set to zero. There are some situations where alternatives can be preferred though.
Goodfellow, Bengio and Courville (2016) suggest setting the bias of output nodes as a suitable marginal statistic of the output, such as *the inverse of the activation function applied to the marginal statistics of the output in the training set.* In addition, bias of level zero should not lead to a saturation effect, as in the case of a ReLU activation.

3.5.2 Algorithms

This section reviews the main algorithms available in the literature to learn the parameters.

Stochastic Gradient Descent

The algorithm computes an unbiased estimate of the gradient by evaluating the gradient on an i.i.d subset of the observations and averaging them. The estimate of the gradient is then used to move in the direction of steepest descent. Algorithm 5 sums up the main steps needed.
Although the learning rate ϵ is referred to as constant in the algorithm, it is in practice often modified over the course of the training. A typical choice is to have ϵ to decay linearly until iteration τ according to:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau,$$

with $\alpha = k/\tau$, and then leave it constant. Although there are no absolute rules when establishing the values for these parameters, Goodfellow, Bengio and Courville (2016) propose the following guidelines:

Algorithm 5: Stochastic Gradient Descent

Input: learning rate ϵ, initial approximation of the parameter θ_0, size of the minibatch m
Output: θ learnt parameter.
Initialise
$\theta \leftarrow \theta_0$
end
while not stopping criterion
minibatch $\leftarrow \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ sampled from the training data
$\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(f(x^{(i)}; \theta), y^{(i)}\right)$
$\theta \leftarrow \theta - \epsilon \hat{g}$
end
Return θ

- ϵ_τ should be about 0.01 ϵ_0;
- too high a learning rate will lead to an oscillating behaviour of the parameters, while if too slow the consequence might be a painfully slow learning or in the worst-case scenario even the process being stuck;
- one could find a good value by evaluating the performance on the first hundred iterations and then set an initial learning rate which is slightly higher than the value identified.

The advantage of SGD is that, although convergence is slower, updates have a cost that does not increase with the size of the training set, allowing convergence even in the case of prohibitively large datasets. In addition, slow asymptotic convergence is made up for by SGD capability of moving quick in good directions in the first iterations, due to the small amounts of computation required to perform the first steps.

Stochastic Gradient Descent with Momentum

According to momentum, the gradient moves in the direction given by a smoothed moving average of the past gradients. Smoothness is introduced by exponential decay of the old gradients, as shown in Algorithm 6.
In the algorithm, the magnitude of α relative to ϵ determines how fast the persistence of the previous gradients with respect to the most recent one.
As an aside, the physical terminology (*momentum*) refers to the modelling of the paramters' position as particles subject to forces: on the one hand there is a force which drives the particle where the loss surface decreases, while the other term is related to viscosity and is proportional to the velocity.

Algorithm 6: Stochastic Gradient Descent with Momentum

Input: learning rate ϵ, initial approximation of the parameter θ_0, initial velocity v_0 size of the minibatch m, momentum parameter α
Output: θ learnt parameter.
Initialise
  $\theta \leftarrow \theta_0$
  $v \leftarrow v_0$
end
while not stopping criterion
  minibatch $\leftarrow \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ sampled from the training data
  $\hat{g} \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L \left(f(x^{(i)}; \theta), y^{(i)} \right)$
  $v \leftarrow \alpha v - \epsilon \hat{g}$
  $\theta \leftarrow \theta + v$
end
Return θ

Algorithm 7: AdaGrad

Input: learning rate ϵ, initial approximation of the parameter θ_0, size of the minibatch m, small constnat $\delta \approx 1e-7$.
Output: θ learnt parameter.
Initialise
  $\theta \leftarrow \theta_0$
  $r \leftarrow 0$ gradient accumulation value
end
while not stopping criterion
  minibatch $\leftarrow \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ sampled from the training data
  $g \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L \left(f(x^{(i)}; \theta), y^{(i)} \right)$
  $r = r + g \circ g$
  $\Delta\theta = -\frac{\epsilon}{\delta + \sqrt{r}} \circ g$
  $\theta \leftarrow \theta + \Delta\theta$
end
Return θ

AdaGrad

In AdaGrad the learning rates are scaled by the square roots of the accumulated squared of the gradient loss. This result in parameters which contributed more to the loss seeing a larger decrease in the learning rate, with the aim of moving more in less steep regions.
The phases of AdaGrad are summed up in Algorithm 7.

Algorithm 8: Adam

Input:
learning rate ϵ suggested to be set as 1e-3,
exponential decay rates for moment estimates,
ρ_1 and ρ_2 in $[0, 1)$,
initial approximation of the parameter θ_0,
size of the minibatch m,
small constnat $\delta \approx 1e-8$.

Output:
θ learnt parameter.

Initialise
- $\theta \leftarrow \theta_0$
- $s \leftarrow 0$ first moment initialisation
- $r \leftarrow 0$ second moment initialisation
- $t \leftarrow 0$

end

while not stopping criterion
- minibatch $\leftarrow \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ sampled from the training data
- $g \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(f(x^{(i)}; \theta), y^{(i)}\right)$
- $t \leftarrow t + 1$
- $s \leftarrow \rho_1 s + (1 - \rho_1) g$
- $r \leftarrow \rho_2 r + (1 - \rho_2) g \circ g$
- $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
- $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
- $r = r + g \circ g$
- $\Delta\theta = -\frac{\epsilon \hat{s}}{\delta + \sqrt{\hat{r}}}$
- $\theta \leftarrow \theta + \Delta\theta$

end

Return θ

Adam

The algorithm is the brainchild of Kingma and Ba (2014). The name is the acronym of ADAptive Moments and can be seen as the evolution of AdaGrad combined to SGD with momentum. Adam uses the first-order moment with exponential weighting of the gradient as an estimate of momentum. In addition, Adam applies bias corrections to the estimates of first and second-order moments due to their being initialised at the origin.

Second-order Methods

We present the main ideas concerning the main second-order methods and related ones. For a more detailed description one can for instance consult Böjers (2010).

The best-known second-order method is Newton's, an iterative method with updates:

$$\theta = \theta - H^{-1}\nabla_\theta \mathcal{L}(\theta),$$

which can be derived by taking the first-order conditions of a second-order linearisation of the loss. In order to avoid the problem of negative eigenvalues, one can opt for the regularised updates:

$$\theta = \theta - (H + \alpha I)^{-1}\nabla_\theta \mathcal{L}(\theta),$$

which is acceptable for comparatively small and negative eigenvalues.
An even greater problem affecting Newton's method in the context of neural nets is the burden of computing the inverse of the Hessian matrix, which can be prohibitive due to the extremely high number of parameters the MLPs can achieve. This motivates the use of other solutions, namely conjugate gradients and Quasi-Newton methods, which aim at finding as good directions as Newton's, but without the need to compute the Hessian.
The first one, conjugate gradients, tries to overcome the typical zigzagging problem which affects steepest descent and is due to the fact that successive directions are orthogonal. For this reason, CG involves conjugate, rather than orthogonal directions. These are directions that satisfy:

$$d_t^T H d_{t-1} = 0.$$

The intuition of this condition comes from observing that for the Newton direction d_N it holds that:

$$-H(\theta_{new} - \theta_{old}) = -Hd_N = \nabla\mathcal{L} \perp d_{t-1},$$

which implies that

$$d_{t-1}^T H d_t = 0$$

is a desirable property to be satisfied when one wants to get close to Newton's properties. It can then be proved that directions defined by

$$d_t = \nabla_\theta \mathcal{L}(\theta) + \beta_t d_{t-1},$$

are conjugate and can alleviate the zigzagging problem. In the equation, β can be obtained by:

$$\beta_t = \frac{||\nabla_\theta \mathcal{L}(\theta_t)||^2}{||\nabla_\theta \mathcal{L}(\theta_{t-1})||^2},$$

which does not require knowledge of the Hessian.
Another option is BFGS (Broyden-Fletcher-Foldfarb-Shanno) algorithm, which falls into the category of quasi-Newton methods. In this algorithm, the Hessian is replaced by an approximation. Algorithm 9 sums up the main steps of the method. It must be stressed, however, that despite the fact that the computation of the Hessian is not needed, its approximation is to be stored, with a significant overhead in terms of memory due to the number of parameters, making the method of little use in many neural nets applications.

Algorithm 9: BFGS

Input:
initial approximation of the parameter θ_0,
size of the minibatch m,
small constnat $\delta \approx 1e-8$.
Output:
θ learnt parameter.
Initialise
$\theta \leftarrow \theta_0$
$k \leftarrow 0$
$H \leftarrow 0$
$s \leftarrow 1e10$
end
while not stopping criterion
minibatch $\leftarrow \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$
sampled from the training data
$g_1 \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L \left(f(x^{(i)}; \theta), y^{(i)} \right)$
$p \leftarrow -Hg_1$
$\alpha \leftarrow \text{argmin } \mathcal{L}(\theta + \alpha p)$
$s \leftarrow \alpha p$
$\theta \leftarrow \theta + s$
$g_2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L \left(f(x^{(i)}; \theta), y^{(i)} \right)$
$y \leftarrow g_2 - g_1$
$H \leftarrow \left(I - \frac{sy^T}{y^T s} \right) H \left(I - \frac{ys^T}{y^T s} \right) + \frac{ss^T}{y^T s}$
end
Return θ

3.5.3 Batch Normalisation

This recent technique is a method of adaptive reparametrisation especially adopted when training very deep nets. What happens in the case of models with several parameters is that updates can lead to unforeseen outcomes, as gradients were computed under the assumptions that the other parameters stayed constant.
Consider a mini-batch B of activations of the layer to be normalised, with the usual design-matrix arrangement such that different rows correspond to different observation and normalise it by subtracting the batch mean and dividing by the batch standard deviation:

$$\mathcal{B}' = \frac{\mathcal{B} - \mu}{\sigma},$$

obtaining σ as:

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum_i (H - \mu)_i^2},$$

where δ is a small constant.
The normalisation of a unit makes the net less powerful in terms of what it can

express. As a result, it is common practice to take hidden units as

$$y = \gamma\hat{x} + \beta, \tag{3.3}$$

where $\hat{x}$ is the normalised input belonging to the minibatch $\mathcal{B}$. This restores the original expressive power of the net while possessing an easier parametrisation which can be easily learnt by gradient descent since it does not result from many terms in the previous layer.
Ioffe and Szegedy (2015), who proposed the method, motivate its need with the phenomenon of the *internal covariate shift*, by which a change in the distribution of the inner units causes a slowdown in learning. Indeed, each hidden layer can be thought of as representing the input of a model with output given by the following hidden layer. If the output of this subnet changes during the learning process, the internal covariate shift, the parameters need to adjust accordingly so that learning becomes slower.
Notice that only the values y produced by (3.3) are taken forward to the next layers, while the normalised inputs $\hat{x}$ remain internal to the transformation in the subnet, with fixed distribution of mean zero and standard deviation one. This is shown to speed up training in most cases.

3.6 Good Practices

The following section includes some guidelines one should follow when implementing neural networks. Far from being an extensive treatment to be rigorously executed, these should be taken recalling that while most successful pipelines are alike, each dataset is problematic in its own way. In addition, what is state-of-the-art in deep learning is characterised by high volatility.
The main areas which require action on the practitioner's side are the following:

- choice of the size of the dataset, if any;
- choice of the model capacity (hidden size, depth...);
- choice of regularisation techniques, if any;
- choice of the error metric;
- choice of the optimisation model;
- underfitting or overfitting detection.

While in the model description we have limited ourselves to the description of feedforward networks due to the topology of the data in our application, other structures require different models: convolutional nets for images as inputs or gated recurrent nets for data which output a sequence.

Typical choices of the optimisation model are SGD with momentum or Adam, enriched with batch normalisation. The latter can be especially useful for convolutional nets and sigmoidal nonlinearities and can also contribute slightly to regularisation. As for regularisation, dropout is a popular choice because of its ease of use.

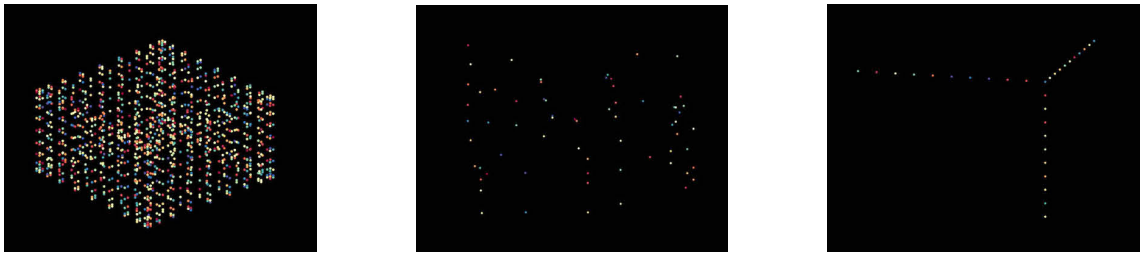
The decision of whether to extend the dataset or not is based on where performance is poor.
If training accuracy is low, one should enrich the model by adding more hidden layers or units – as it might be the case that it does not possess enough expressing power – or adapt the way the model is learning by adjusting the learning rates. If neither of these work it might be the case that the dataset suffers form bad quality. If test accuracy is instead faring worse, further data collection usually proves beneficial, although it comes at a cost and in the worst case might not even be possible. As an alternative, simplification of the model is also worth trying, as well as introducing or increasing regularisation. In the general case, a good performance of the algorithm follows from the combination of a large model and adequate regularisation, the latter probably deriving from the use of dropout.

Goodfellow, Bengio and Courville (2016) mention the brute force approach as a means to achieve virtually guaranteed success: this involves the iteration of increasing model capacity and increasing the dataset size. By subsequently performing these operations one should achieve the desired results on test set.

Chapter 4

Low-Rank Approximation

This chapter presents some concepts which are used to extend the pricing methodology proposed in this thesis.
Although more details will be presented in the next chapter, and particularly in Chapter 6, we mention for now two problems which can arise when training neural networks with standard machines. First of all, the dataset might be too large to be stored, due to memory constraints, or too cumbersome to be generated via Monte Carlo or other numerical methods. Alternatively, one might have a few data points, but not enough for the neural network to be trained.
In these cases it would be desirable to have a compressed representation of the training tensor or one which allows to retrieve more entries from the few ones available, exploiting the tensor structure of the data.
This chapter proposes a solution based on dimensionality reduction techniques which addresses – and possibly solves – the issues highlighted.
Figure 4.1 – taken from `http://www.maths.qmul.ac.uk/~kglau/` – shows the idea behind tensor compression: the goal of this chapter is to exploit a method which gives back a tensor in a compact representation after sampling a few of its entries.



(a) Full tensor (b) Sampling (c) Compression

Figure 4.1: *Scheme of the idea behind tensor compression.*

After introducing some preliminary concepts related to tensors, the chapter describes the tensor-train decomposition, which consists of a compressed representation of a

given tensor (Section 4.1) and the completion algorithm, used to obtain such a representation (Section 4.3).

Vectorisation of a Matrix

Given a matrix $A \in \mathbb{R}^{m \times n}$, the vectorisation of A is denoted as $\text{vec}(A)$ where

$$\text{vec} : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}.$$

For instance, with matrix A below, its vectorisation is obtained by stacking its columns as follows:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \quad \text{and} \quad \text{vec}(A) = \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{12} \\ a_{22} \\ a_{32} \end{pmatrix}.$$

Kroenecker Product

For a matrix $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{k \times l}$, the Kroenecker product is defined by:

$$B \otimes A = \begin{pmatrix} b_{11}A & \dots & b_{1l}A \\ \vdots & & \vdots \\ b_{k1}A & \dots & b_{kl}A \end{pmatrix} \in \mathbb{R}^{mk \times nl}.$$

It holds that:

1. $\text{vec}(AX) = (I \otimes A)X$;
2. $I_m \otimes I_n = I_{mn}$.

Definition 4.0.1. A d-th order tensor χ of size $n_1 \times n_2 \times \dots n_d$ is a d-dimensional array with entries

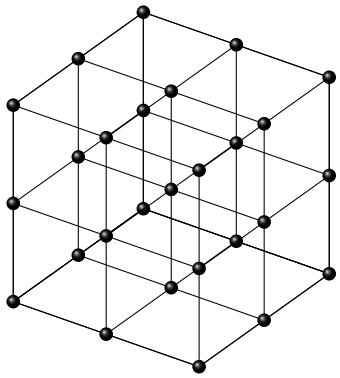
$$\chi_{i_1,\dots,i_d}, \quad i_\mu \in \{1, ..., n_\mu\} \quad \text{for} \quad \mu = 1, .., d.$$

In our case, the entries of the tensor are real, so that:

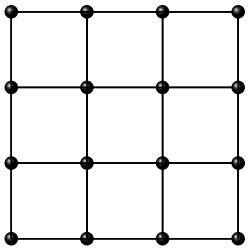
$$\chi \in \mathbb{R}^{n_1 \times \dots \times n_d}.$$

□

Simple tensors of order three (left) and two, a matrix, (right) are shown in Figure 4.2. We now introduce common operations on tensors.



(a) Tensor of order 3



(b) Tensor of order 2

Figure 4.2: *Simple three-d and two-d tensors.*

- Vectorisation is one of the possible ways to stack the entries of a tensor in a long column vector and is denoted as *vec*:

$$\text{vec} : \mathbb{R}^{n_1 \times \ldots \times n_d} \to \mathbb{R}^{n_1 \ldots n_d}.$$

  Vectorisation stacks the entries of the tensor in reverse lexicographical order into a vector. For instance, in three dimensions and with $n_1 = 3$, $n_2 = 2$, $n_3 = 3$ one has:

$$\text{vec}(\chi) = \begin{pmatrix} \chi_{111} \\ \chi_{211} \\ \chi_{311} \\ \chi_{121} \\ \chi_{221} \\ \vdots \\ \chi_{123} \\ \chi_{223} \\ \chi_{323} \end{pmatrix}.$$

- Matricisation consists of making a dth-order tensor χ a matrix. One says that for $\mu = 1, ..., d$, the μ-mode matricisation of χ is the matrix

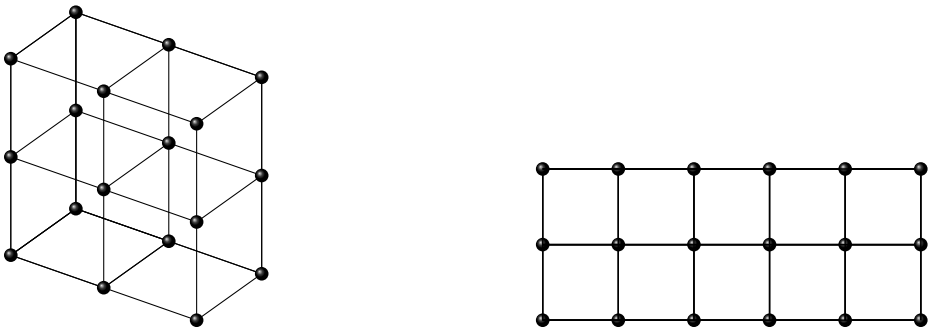
$$X^{(\mu)} \in \mathbb{R}^{n_\mu \times n_1 \ldots n_{\mu-1} n_{\mu+1} \ldots n_d}.$$

  Figure 4.3 shows the one-matricisation operation on a simple $3 \times 3 \times 2$ tensor turned into a $3 \times (3 \cdot 2)$ matrix.

  As an example, notice that for a matrix one has $A^{(1)} = A$ and $A^{(2)} = A^T$.

- The 1-mode matrix multiplication takes a matrix A of size $m \times n_1$ and multiplies it by the 1-mode matricisation of χ and is denoted as:

$$\mathcal{Y} = A \circ_1 \chi \iff Y^{(1)} = A X^{(1)},$$



(a) Tensor of order 3 (b) Matricisation of the tensor

Figure 4.3: *Matricisation.*

which of course can be generalised to other modes.

- By definition, one has that:

$$\mathrm{vec}(\chi) = \mathrm{vec}(X^{(1)}) \quad \text{and} \quad \mathrm{vec}(A \circ_1 \chi) = \mathrm{vec}(AX^{(1)}).$$

This implies, recalling the aforementioned properties:

$$\begin{aligned}\mathrm{vec}(A \circ_1 \chi) &= (I \otimes A)\mathrm{vec}(X^{(1)}) = (I_{n_d \dots n_2} \otimes A)\mathrm{vec}(\chi) \\ &= (I_{n_d} \otimes \dots \otimes I_{n_2} \otimes A)\mathrm{vec}(\chi),\end{aligned}$$

where the first equality follows from the definition of the operations of Kroenecker product and vectorisation, the second one from the previous observation and the third one from the second property of the Kroenecker product.

4.1 Tensor-Train Decomposition

We now introduce the dimensionality reduction technique which is used in the second part of this work. The tensor-train (TT) decomposition serves both purposes of compactly storing a high-dimensional tensor and obtaining such a representation. A tensor χ is in tensor-train decomposition if it can be written as:

$$\chi(i_1, ..., i_d) = \sum_{k_1=1}^{r_1} \dots \sum_{k_{d-1}=1}^{r_{d-1}} \mathcal{U}_1(1, i_1, k_1)\mathcal{U}_2(k_1, i_2, k_2) \dots \mathcal{U}_d(k_{d-1}, i_d, 1). \tag{4.1}$$

The smallest possible tuple $r_1, ..., r_{d-1}$ is called the TT rank of χ, while

$$\mathcal{U}_\mu \in \mathbb{R}^{(r_{\mu-1} \times n_\mu \times r_\mu)}$$

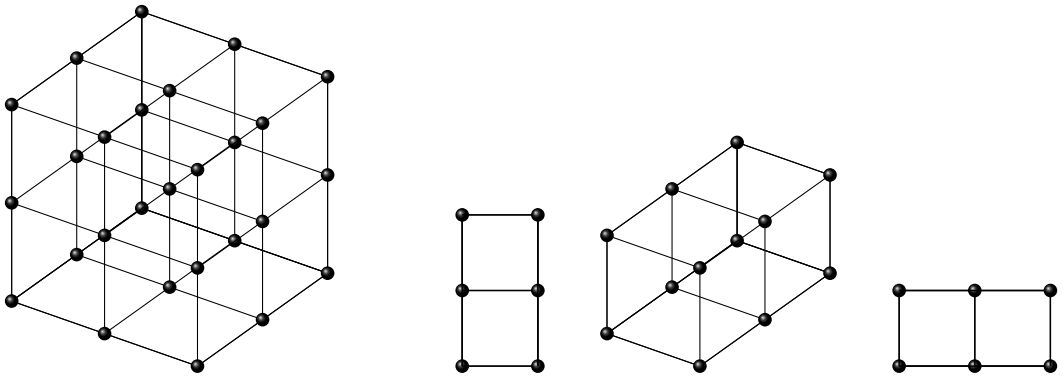
are called the TT cores for $\mu = 1, ..., d$, setting $r_0 = r_d = 1$. Compression is high when the TT ranks are small. Indeed, if we denote as

$$r = \max\{r_0, \dots, r_d\}$$

and as

$$n = \max\{n_1, \ldots, n_d\},$$

the memory needed with the compressed representation is $\mathcal{O}(dnr^2)$ rather than $\mathcal{O}(n^d)$ achieved in the full case.
Figure 4.4 shows the original tensor, on the left, and a possible decomposition into lower-order tensors, on the right. The gain in space saved is of course larger as the order of the full tensor increases.



(a) Tensor of order 3 (b) Possible TT decomposition with tensors $\mathcal{U}_1$, $\mathcal{U}_2$ and $\mathcal{U}_3$

Figure 4.4: *Decomposition example of a full tensor (left) into the product of lower-order tensors (right).*

Denote now a slice of the core as:

$$U_\mu(i_\mu) = \mathcal{U}_\mu(:, i_\mu, :) \in \mathbb{R}^{r_{\mu-1} \times r_\mu};$$

then:

$$\chi(i_1, ..., i_d) = U_1(i_1)U_2(i_2)\ldots U_d(i_d),$$

which is the more familiar matrix product.
If we now group the first μ factors together:

$$\chi(i_1, ..., i_\mu, i_{\mu+1}, \ldots, i_d) = \sum_{k_\mu=1}^{r_\mu} \left(\sum_{k_1,\ldots,k_{\mu-1}} \mathcal{U}_1(1, i_1, k_1)\ldots\mathcal{U}_\mu(k_{\mu-1}, i_\mu, k_\mu) \right) \left(\sum_{k_{\mu+1}\ldots k_{d-1}} \mathcal{U}_{\mu+1}(k_\mu, i_{\mu+1}, k_\mu)\ldots\mathcal{U}_d(k_{d-1}, i_d, 1) \right),$$

which turns out to be the product of two matrices.
Now define the μ-th unfolding of $\chi \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ by arranging the entries in a matrix:

$$X^{<\mu>} \in \mathbb{R}^{(n_1 n_2 \ldots n_\mu) \times (n_{\mu+1} \ldots n_d)}. \tag{4.2}$$

This can be obtained easily in `Matlab`, via the `reshape` function, specifying the number of rows and the number of columns.

Similarly, take the interface matrices:

$$X_{\leq\mu} \in \mathbb{R}^{(n_1 n_2 \ldots n_\mu)\times r_\mu}$$

and

$$X_{\geq\mu+1} \in \mathbb{R}^{r_\mu \times (n_{\mu+1} n_{\mu+2} \ldots n_d)}$$

as:

$$X_{\leq\mu}(i_{\text{row}}, j) = \sum_{k_1,\ldots,k_{\mu-1}} \mathcal{U}_1(1, i_1, k_1)\ldots\mathcal{U}_\mu(k_{\mu-1}, i_\mu, j)$$

and

$$X_{\geq\mu+1}(j, i_{\text{column}}) = \sum_{k_{\mu+1}\ldots k_{d-1}} \mathcal{U}_{\mu+1}(j, i_{\mu+1}, k_\mu)\ldots\mathcal{U}_d(k_{d-1}, i_d, 1),$$

where i_{row} and i_{column} are given by:

$$i_{\text{row}} = 1 + \sum_{\nu=1}^{\mu}(i_\nu - 1)\prod_{\tau=1}^{\nu-1} n_\tau$$

$$i_{\text{col}} = 1 + \sum_{\nu=\mu+1}^{d}(i_\nu - 1)\prod_{\tau=\mu+1}^{\nu-1} n_\tau$$

As a result,

$$X^{<\mu>} = X_{\leq\mu}X_{\geq\mu+1}, \quad \mu = 1, ..., d-1.$$

Notice that these matrix factorisations are nested, so that, for instance:

$$X_{\leq\mu} = (I_{n_\mu} \otimes X_{\leq\mu-1})U_\mu^L, \quad \text{with} \quad U_\mu^L = U_\mu^{<2>},$$

and

$$X_{\geq\mu}^T = U_\mu^R(X_{\geq\mu-1}^T \otimes I_{n_\mu}), \quad \text{with} \quad U_\mu^R = U_\mu^{<1>}.$$

One thus has:

$$\begin{aligned}
\text{vec}(\chi) = X_{\leq d} &= (I_{n_d} \otimes X_{\leq d-1})U_d^L \\
&= (I_{n_d} \otimes ((I_{n_{d-1}} \otimes X_{\leq d-2})U_{d-1}^L))U_d^L \\
&= (I_{n_d} \otimes I_{n_{d-1}} \otimes X_{\leq d-2})(I_{n_d} \otimes U_{d-1}^L)U_d^L \\
&= \ldots \\
&= (I_{n_d} \otimes I_{n_{d-1}} \otimes \ldots \otimes I_{n_2} \otimes U_1^L)\ldots(I_{n_d} \otimes U_{d-1}^L)U_d^L.
\end{aligned}$$

By construction and definition of tensor-train rank, the TT rank of the tensor is bounded from below by the ranks r_μ of the unfolding matrices $X^{<\mu>}$, $\mu = 1, \ldots, d$. In addition it can be shown, Oseledets (2011), that TT-ranks not higher than r_μ can be achieved with TT decomposition (4.1).

Theorem 4.1.1 (Theorem 2.1 in Oseledets (2011))**.** *If for each unfolding matrix $X^{<k>}$ of form* (4.2) *of a d-dimensional tensor χ*

$$rank\, X^{<k>} = r_k,$$

then there exists a decomposition (4.1) *with TT-ranks not higher than r_k.*

The proof allows for the construction of an algorithm (described in Algorithm 10) implementing truncation in TT format and is reported hereunder.

Proof. The first unfolding matrix has rank r_1 and can hence be decomposed as:

$$X^{<1>} = U_1 \tilde{X}^{<1>}$$

with $U_1 \in \mathbb{R}^{n_1 \times r_1}$ and $\tilde{X}^{<1>} \in \mathbb{R}^{r_1 \times n_2 \dots n_d}$.
As a result

$$\tilde{X}^{<1>} = (U_1^T U_1)^{-1} U_1^T X^{<1>}$$

and U_1 is the first TT core $\mathcal{U}_1$. In addition, one has $X_{\geq 2} = \tilde{X}^{<1>}$.
Replicating the reasoning presented a few lines above we have:

$$X_{\leq 2} = (I_{n_2} \otimes X_{\leq 1}) U_2^{<2>},$$

so that:

$$\begin{aligned} X^{<2>} &= X_{\leq 2} X_{\geq 3} = (I_{n_2} \otimes X_{\leq 1}) U_2^{<2>} X_{\geq 3} \\ &= (I_{n_2} \otimes X_{\leq 1}) \tilde{X}^{<2>}. \end{aligned}$$

The second TT core is given by $U_2^{<2>}$ via a suitable reshaping. In turn, $U_2^{<2>}$ can be derived by decomposing $\tilde{X}^{<2>}$ as done for $\tilde{X}^{<1>}$. Notice that:

$$\operatorname{rank}(\tilde{X}^{<2>}) = \operatorname{rank}(X^{<2>}),$$

since by assumption $\operatorname{rank}(X^{<2>}) = r_2$ and $X_{\leq 1} = U_1$ is full column rank.
The reasoning can be iterated to obtain all the cores as well as:

$$\operatorname{vec}(\chi) = (I_{n_d} \otimes I_{n_{d-1}} \otimes \dots \otimes I_{n_2} \otimes U_1^L) \dots (I_{n_d} \otimes U_{d-1}^L) U_d^L.$$

The proof can be implemented as an algorithm constructing the TT decomposition in order to approximate a given tensor in TT format: it is shown in Algorithm 10.

4.2 Manifold Optimization

One aims at solving the problem:

$$\min_{X \in \mathcal{M}_r} f(X)$$

where $\mathcal{M}_r$ is a manifold of rank-r tensors. In order for the iterates to remain on the manifold one has to modify traditional optimisation algorithms such as line search or Newton's.

Algorithm 10: Truncation in TT Format

Input:
$\chi \in \mathbb{R}^{n_1 \times \ldots \times n_d}$
target TT rank $r_1, ..., r_{d-1}$.
Output:
TT cores $\mathcal{U}_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ defining the TT decomposition.
Initialise
  $r_0 \leftarrow 1$
  $r_d \leftarrow 1$
  $\chi \in \mathbb{R}^{1 \times n_1 \times \ldots \times n_d}$
end
For $k = 1, \ldots, d-1$
  $X^{<2>} \leftarrow \texttt{reshape}(\chi, r_{k-1} n_k, n_{k+1} \ldots n_d)$
  $X^{<2>} \approx U \Sigma V^T$ of rank r_k
  $\mathcal{U}_k \leftarrow \texttt{reshape}(U, r_{k-1}, n_k, r_k)$
  χ updated via $X^{<2>} \leftarrow U^T X^{<2>} = \Sigma V^T$
end
$\mathcal{U}_d \leftarrow \chi$
Return $\mathcal{U}$

Manifold

We say that a subset $\mathcal{M}$ of $\mathbb{R}^n$ is a manifold of dimension m if every point $X \in \mathcal{M}$ is contained in an open subset U of $\mathcal{M}$ that can be parametrised by some function $\varphi : \Omega \subset \mathbb{R}^m \to U$, φ being a homeomorphism and with injective first derivative (see Gallier (2018) for a more rigorous definition). We say that φ is a local parametrisation of $\mathcal{M}$ at X and that $\varphi^{-1} : U \to \Omega$ is the local chart.
As an example, $GL(n, \mathbb{R})$ is a manifold: indeed, it is an open subset of $\mathbb{R}^{n^2}$, as it is the inverse image of the determinant mapping, which is continuous.

Tangent Space

Given a smooth curve

$$\gamma : \mathbb{R} \to \mathcal{M},$$

with $x = \gamma(0)$, $\gamma'(0)$ is called a tangent vector at x.
The tangent space $T_x\mathcal{M}$ is the set of all tangent vectors at x and we call the tangent bundle $T\mathcal{M}$ the disjoint union of all tangent spaces.

Retraction

A mapping R:

$$R : \bigcup_{x \in \mathcal{M}} T_x\mathcal{M} \to \mathcal{M},$$

where $R : (x, \xi) \mapsto R_x(\xi)$, is called a retraction on $\mathcal{M}$ if for every $X_0 \in \mathcal{M}$ there exists a neighbourhood $\mathcal{U}$ around $(X_0, 0) \in T\mathcal{M}$ such that:

- $\mathcal{U}$ is in the domain of R and R restricted to $\mathcal{U}$ is smooth;
- the retraction on a point on the manifold is the point itself; restricted to the zero section is the identity on the first variable.
- the retraction does not act on the direction of curves. Namely, if ξ is a tangent vector in $T_x\mathcal{M}$ and one takes a curve $\gamma : \beta \mapsto R_x(\beta\xi)$, then $\gamma'(0) = \xi$.

Riemannian Gradient

Consider $v \in T_x\mathcal{M}$ and $\gamma : (-\epsilon, \epsilon) \to \mathcal{M}$ such that $\gamma'(0) = v$.
In addition take a smooth function:

$$F : \mathcal{M} \to \mathbb{R}.$$

Define then:

$$DF(x)[v] = \frac{d}{dt}\bigg|_{t=0} F(\gamma(t)).$$

When each element of the tangent space is endowed with a smoothly varying inner product, the manifold $\mathcal{M}$ is said to have a Riemannian structure. In the case at hand we can restrict the canonical scalar product in $\mathbb{R}^n$ to obtain such a structure. If this is the case, the Riemannian gradient of a smooth function $f : \mathcal{M} \to \mathbb{R}$ at point $x \in \mathcal{M}$ is defined as the unique element $\nabla_R f \in T_x\mathcal{M}$ such that:

$$\langle \nabla_R f(x), \xi \rangle = Df(x)[\xi] \quad \forall \xi \in T_x\mathcal{M}.$$

Line Search and Extensions

In the simple Euclidean case, an optimisation problem of the type:

$$\min_{x \in \mathbb{R}^n} f(x)$$

can be tackled by line search, i.e. by iteratively solving:

$$\min_{\alpha} f(x_j + \alpha\eta_j)$$

for a search direction η_j and step size α, so as to set:

$$x_{j+1} = x_j + \alpha\eta_j.$$

This is problematic on a manifold as addition is not well defined. What one can do is to move along the tangent space and go back to the manifold via retraction.

4.3 Completion Algorithm

The completion problem consists of retrieving the entries of a tensor, when only a small fraction of them are known. A possible solution can involve the assumption, which is then to be verified, that the tensor exhibits a low-rank structure, namely that it can be approximated closely enough by a lower-rank tensor than the tensor itself.
Define first the projection onto a set Ω:

$$P_\Omega : \mathbb{R}^{n_1 \times \ldots \times n_d} \to \mathbb{R}^{n_1 \times \ldots \times n_d}$$

as:

$$P_\Omega X = \begin{cases} X_{i_1,\ldots,i_d} & \text{if} \quad (i_1, \ldots, i_d) \in \Omega \\ 0 & \text{else} \end{cases},$$

where Ω is a sampling set, $\Omega \subset \{1, .., n_1\} \times \ldots \times \{1, .., n_d\}$, containing the indices of the known entries of the tensor.
The completion problem can thus be formulated as:

$$\begin{aligned} \min_{\chi} \quad & f(\chi) = \frac{1}{2}||P_\Omega \chi - P_\Omega A||_F^2 \\ \text{subject to} \quad & \chi \in \mathcal{M}_r := \{\chi \in \mathbb{R}^{n_1 \times \ldots \times n_d} \,|\, \text{rank}_{TT} = \mathbf{r}\}, \end{aligned} \tag{4.3}$$

with A original tensor and χ its approximation.
Since $\mathcal{M}_r$, the set of TT tensors with given TT ranks, is a smooth manifold, Riemannian optimisation techniques can be applied. Specifically, and following Steinlechner (2016), a Riemannian conjugate gradient method (CG) is applied. Thanks to retraction, iterates produced by the algorithm at each step stay on the manifold.
The algorithm moves in directions which depend on the Riemannian gradient and reaches new points which are brought back into the manifold via retraction: it is shown in Algorithm 11.
In the algorithm, $\mathcal{T}_{x_{k-1} \to x_k}$ refers to the vector transport, which addresses the problem of bringing the the point in the previous iteration to the tangent space of the new point so as to make the two directions compatible.
In addition, the Riemannian gradient, denoted as ∇_R in the algorithm, is obtained by the projection of the Euclidean gradient onto the tangent space – for a more rigorous statement and proof, see Absil, Mahony and Sepulchre (2008). In other words, one needs to set:

$$\nabla_R f(X) = P_{T_X \mathcal{M}_r} \nabla f = P_{T_X \mathcal{M}_r}(P_\Omega X - P_\Omega A).$$

The Riemannian Conjugate Gradient Algorithm is brought to a halt when a desired level of accuracy is attained, compatibly with the TT ranks. A test set is chosen of index points Ω_C not in Ω and the error is evaluated both at these points and at Ω. One computes:

$$\epsilon_\Omega(\chi_k) = \frac{||P_\Omega A - P_\Omega \chi_k||}{||P_\Omega A||}.$$

Algorithm 11: Completion by CG

Input: Ω, $P_\Omega X$.
Output: X, completed tensor.
Initialise
  $X_0 \leftarrow$ initial guess
  $\xi_0 \leftarrow \nabla_R f(X_0)$
  $\eta_0 \leftarrow -\xi_0$
  $\alpha_0 \leftarrow \operatorname{argmin}_\alpha f(X_0 + \alpha\eta_0)$
  $X_1 \leftarrow R(X_0, \alpha_0\eta_0)$
end
For $k = 1, 2, \ldots$
  $\xi_k \leftarrow \nabla_R f(X_k)$
  $\eta_k \leftarrow -\xi_k + \beta_k \mathcal{T}_{x_{k-1} \to x_k} \eta_{k-1}$
  $\alpha_k \leftarrow \operatorname{argmin}_\alpha f(X_k + \alpha\eta_k)$
  $X_{k+1} \leftarrow R(X_k, \alpha_k\eta_k)$
end
Return X_{k+1}

When the error stagnates, i.e.,

$$\frac{|\epsilon_\Omega(\chi_k) - \epsilon_\Omega(\chi_{k+1})|}{|\epsilon_\Omega(\chi_k)|} < \delta$$

and

$$\frac{|\epsilon_{\Omega_C}(\chi_k) - \epsilon_{\Omega_C}(\chi_{k+1})|}{|\epsilon_{\Omega_C}(\chi_k)|} < \delta$$

the algorithm stops, with $\delta > 0$ small number.
The ranks which determine the optimisation problem are set adaptively: one starts with the simplest possible rank structure, namely $(r_0, \ldots, r_d) = (1, \ldots, 1)$, and. if the desired accuracy is not attained, the rank is increased to $(r_0, \ldots, r_d) = (1, 2, 1, \ldots, 1)$ and so on, until increasing the ranks does not lead to improved accuracies.
As for the test set Ω, following Glau, Kressner and Statti (2019), starting for a set of given size, this is added to the set Ω whenever the stopping criterion on the test accuracy is not satisfied. After adding the new samples, the algorithm is run again until a sampling percentage is reached, as shown in Algorithm 12.
The algorithm implementation used in this work is contained in the `Matlab` package developed by Steinlechner, for which see Steinlechner (2016), and extended by Statti, as shown in Glau, Kressner and Statti (2019).

Algorithm 12: Adaptive Sampling Strategy

Input:

$r_{\max}$ maximum rank admitted
$P_\Omega A$ sample points on the tensor
p maximal percentage of the size of Ω

Output: χ, completed tensor of TT ranks $r_\mu \leq r_{\max}$.

Initialise
  Ω_C^{new} test set
end
Perform adaptive rank strategy and obtain completed tensor χ_C.
$err_{\text{new}} \leftarrow \epsilon_{\Omega_C^{\text{new}}}(\chi_k)$
while $|\Omega|/\textbf{size}(A) < p$
  $err_{\text{old}} \leftarrow err_{\text{new}}$
  $\bar{\chi} \leftarrow \text{rank}(1, \ldots, 1)$ approximation of χ_C
  $\Omega_C^{\text{old}} \leftarrow \Omega_C^{\text{new}}$
  Create new test set Ω_C^{new} such that $\Omega_C^{\text{old}} \cap \Omega_C^{\text{new}} = \emptyset$
  $\Omega \leftarrow \Omega \bigcup \Omega_C^{\text{new}}$
  Perform adaptive rank strategy and obtain completed tensor χ_C starting from $\bar{\chi}$.
  $err_{\text{new}} \leftarrow \epsilon_{\Omega_C^{\text{new}}}(\chi_k)$
  if stopping criterion
    break
  end
end
Return $\chi \leftarrow \chi_C$

Chapter 5

Learning Methodology

This section describes the steps taken to learn prices on a hyper-rectangle of the parameters of interest. Learning is performed by means of neural nets and is supervised. This means that for the algorithm to learn a pricing function, a dataset containing inputs and outputs has to be provided. The learning process will be such that the algorithm learns how to construct a function which maps inputs – option and model parameters – into the right outputs, the price.
In order to achieve this, the following steps need to be implemented, both in the case of basket and of American options:

- construct a dataset of parameters' values and corresponding option prices as follows:
  1. generate different combinations of parameters;
  2. compute the price for each combination of parameters with the benchmark numerical method, that is Monte Carlo with a variance reduction technique for basket options and PDE for American put options;
- construct a neural net architecture;
- input the dataset to the neural net and learn the pricing function through the training of the neural net;
- test the pricing function on fresh data.

Each of the previous points poses some challenges, which are discussed in order.

First of all, the definition of the grid on which prices are computed. Our initial approach consists of picking evenly spaced points on the interval of interest for each of the parameters, so that the hyper-rectangle of parameters would look, if it were in two dimensions, similar to that represented in Figure 4.4a. This can be a possible choice for neural nets, but it usually leads to poor results when applying polynomial interpolation methods, whose accuracy depends crucially on the loci of the points. Other possibilities for the choice of the grid will be presented later: we will examine

the performance when Chebychev points are employed and then turn to random points. The latter choice is crucial when one needs to move from learning from computed prices (via a benchmark technique) to learning from real data.
Notice that the flexibility in the choice of the points – as well as flexibility in general – is an advantage of the neural-net approach compared to other methods: polynomial methods, for instance would in all likelihood not be able to handle real data points, as convergence would be poor for prices selected randomly on the space of parameters.

In addition, the construction of the dataset can prove painfully slow. On the one hand, a good number of points has to be provided to the neural net in order for the learning to be successful. On the other hand, however, computing the prices for each of the parameters' combinations is expensive. This is especially true for Monte Carlo-based approaches, which require a good number of data points, and less so for the PDE solution, which can be computed faster exploiting efficient solutions of the systems of equations and sparse representation of matrices.
While this does not make the process infeasible in our initial setting, and it will not in yet more complex ones, when striving to work in very high dimensions other solutions should be conceived. This is due to both time and memory constraints, which become binding as the dimensions of the problem scale up. These issues will be discussed in Chapter 6, where this thesis introduces a novel approach based on the construction of an approximated *synthetic* dataset. By doing so, it will be possible to speed up the data generation process and overcome the limitations posed by memory.

A further element to be discussed is the architecture of the network, which fundamentally requires trial and error. As mentioned in Chapter 3, one has to try with different combinations of values for the number of hidden layers, of hidden nodes, of learning rate and with different regularisation techniques. Choices will be presented and motivated in the following sections.

The current chapter describes the learning methodology and results for the basket option case in Section 5.1, where two different types of grids are presented as training data, namely the evenly-spaced one and the Chebychev grid. Section 5.2 shifts the focus to American option, trained on equally-spaced points. For American options, we also consider taking more nodes for those parameters which seem to be more correlated with the option price: results are shown in Paragraph 5.2.1. Final remarks on the neural networks follow in Section 5.3.

5.1 Basket Options

Basket options were discussed in Chapter 2.1. The approach followed in this thesis to obtain prices for the intervals of interest for each parameter consists of generating prices on the grid of parameters as described in the previous sections and using them to train the neural network.
The network architecture was chosen after several trials, together with cross validation techniques for some of the parameters of interest, namely the learning rate and the number of hidden nodes. Although the results of cross validation for the choice of parameters may vary sensibly depending on the network architecture, cross validation was used to form a general idea of the behaviour of the multi-layer perceptron with respect to the parameters.
We aim at learning a pricing function which takes as input the initial price of five underlying assets and yields the basket option price. The problem can be formalised as having to learn a function f such that:

$$f : [1, 1.5]^5 \to \mathbb{R}^+$$

$$f : \left(S_1^{(0)}, S_2^{(0)}, S_3^{(0)}, S_4^{(0)}, S_5^{(0)}\right) \mapsto \text{price} = f\left(S_1^{(0)}, S_2^{(0)}, S_3^{(0)}, S_4^{(0)}, S_5^{(0)}\right).$$

Notice that, in fact, the parameters r, Σ, σ, T and K are fixed, so that:

$$f = f_{r,\Sigma,\sigma,T,K},$$

r being the risk-free rate, Σ the correlation matrix between the assets, σ the vector of volatilities for the assets, T the maturity of the option and K the strike price. Hence, one can easily understand that the problem can be further extended to include explicit dependency of the pricing function f on all the parameters mentioned. This would further increase the dimensions of the problem, calling for other solutions in terms of the data generation and the storage of the training set: one such solution will be presented and discussed in Chapter 6.
The values of the parameter are summed up for the sake of clarity in Table 5.1.

Table 5.1: **Parameters in the basket option problem**

	Value or value range
$S_0^{(i)}$, $i = 1, \dots d$	[1,1.5]
d	5
ρ	[-1,1]
σ	0.2
Σ	I_d
T	0.25
r	0

In Table 5.1 we denoted as I_d the identity matrix of size $d \times d$ and as d the number of underlying assets in the basket. The assets are taken to be uncorrelated. Further experiments will present results where the correlation matrix between the assets is different from the identity.
Training was performed with the use of a validation set to monitor the behaviour of the net throughout the iterations: one typically observes that accuracy on the training set, which affects directly the training process of the net, practically always goes down during the training phase, before becoming stationary. The introduction of a validation set helps detect overfitting. Indeed, the net will try to adhere to the training points in order to improve performance, to the detriment of generality. For this reason, the training performance will improve, but evaluating the net on fresh points will reveal poorer results whenever the number of training iterations increases disproportionately. Hence, monitoring performances on a fresh validation set can suggest when to stop the training phase so as to prevent overfitting. In addition, the algorithm is written so as to store the best-performing model attained, based on validation results. By doing so, when overfitting occurs, one is able to retrieve the best model previously obtained.
Figure 5.1 shows graphically how monitoring was performed: as one can see, both the validation and the training errors are decreasing during the training phase. When this is not the case, the model might be tending to overfit.

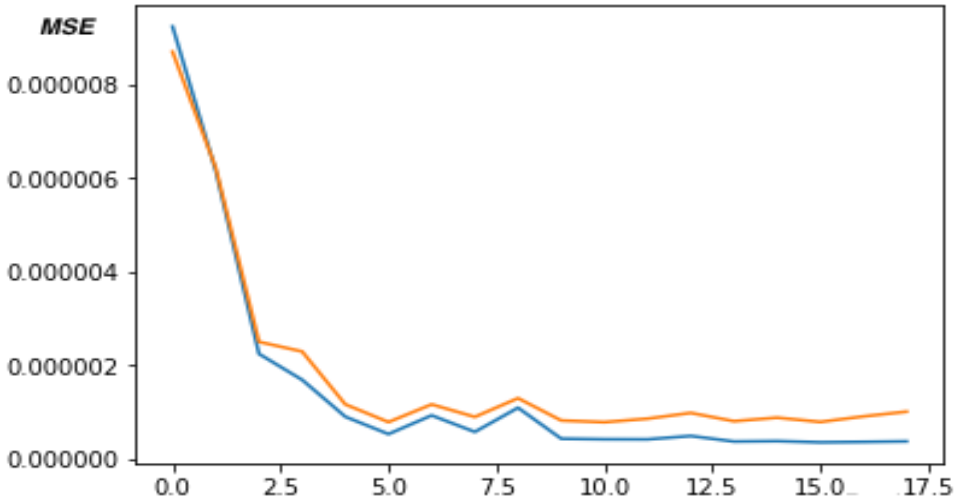


Figure 5.1: *Values for train and validation errors across the epochs of the training phase.*

The architecture of the neural net for basket options trained on an evenly-spaced grid is summarised in Table 5.2 and shown in Figure 5.2, as the small size allows for its complete representation. This is also the reason why the training of the neural net is extremely fast, as a good performance is achieved after a few seconds ($\approx 20s$).

Table 5.2: **Neural network architecture for basket options – evenly-spaced grid**

	Input Size	Output Size
Linear	5	10
Sigmoid		
Linear	10	7
Sigmoid		
Linear	7	5
Sigmoid		
Linear	5	1
ReLU		

The net consists of alternating linear and sigmoid units, with the exception of the last layer, which features a rectified linear unit: this last step is paramount in order to ensure that positive prices are attained.

Once the neural net has been trained, a test set is used to evaluate its performance. A set of 1000 points is hence generated randomly on the parameter space, which is taken slightly smaller than the parameter space itself in order to counteract border effects. The model produced by the neural net is hence applied to each of the points of the test set and the output is set against the benchmark price generated with Monte Carlo to obtain the error. Then, some statistics are computed.
Errors are presented in Figure 5.3, together with their histogram. The left-hand side panel shows the error on each sample of the test set, labelled on the horizontal axis, while the histogram on the right-hand side is constructed with 100 bins. It is easy to see that deviations from the Monte Carlo prices are heavily concentrated around zero, with very few exceptions.

Denoting the prediction obtained by the neural net as $\hat{y}$ and the true value generated by the benchmark method (Monte Carlo in the basket case), the error metrics for the neural net are computed as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2;$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|,$$

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{|\hat{y}_i - y_i|}{y_i},$$

and

$$\text{Maximum Absolute Error} = \max|\hat{y} - y|,$$

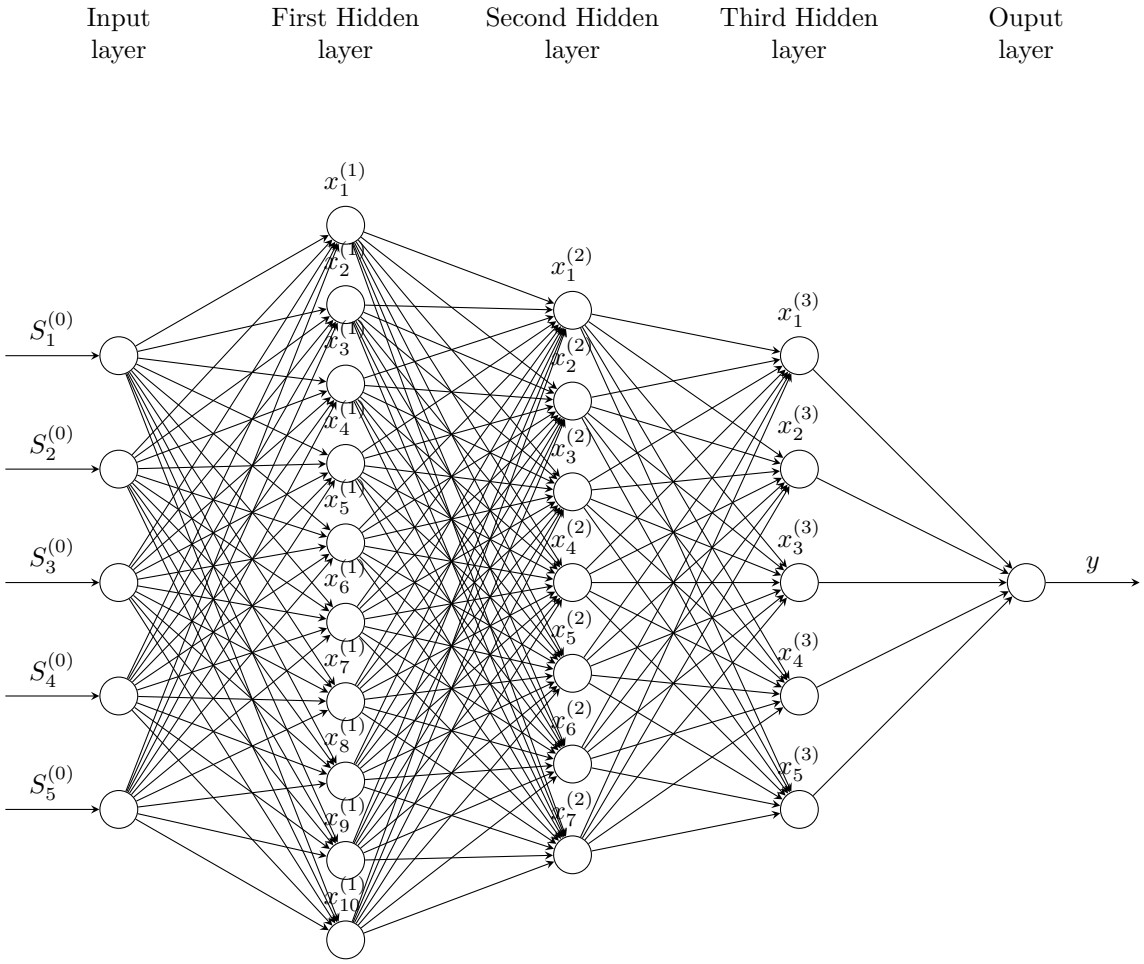
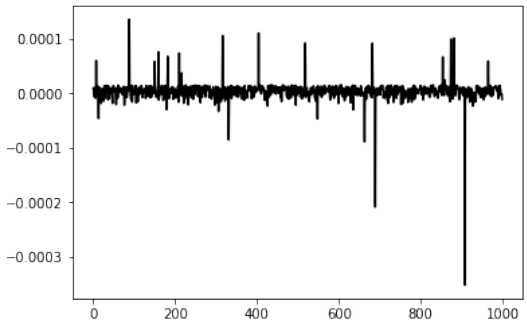
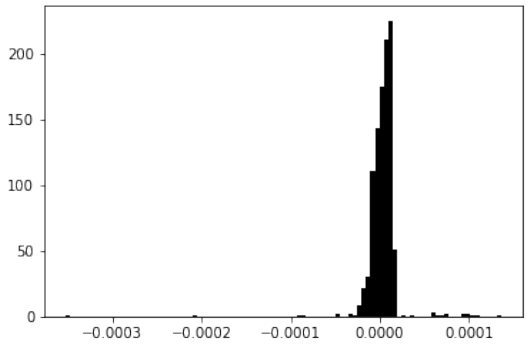


Figure 5.2: *Neural net architecture for the basket option problem.*



(a) Error for each point of the test set, indexed on the horizontal axis



(b) Histogram of the Error

Figure 5.3: *Error values and histogram for basket option prices computed on test set by a neural net trained on evenly-spaced points.*

where the maximum is taken over the entire test set.
The error metrics are presented in Table 5.3. As one can see, in the worst-case scenario the error only appears in the fourth digit. On average, the error only affects the fifth digit.

Table 5.3: **Error Metrics for basket options on evenly spaced points**

	Value
MSE	4.93e-10
Maximum Absolute Error	1.94e-4
MAE	1.68e-5
MAPE	7.85e-5

Lastly, it ought to be stressed that the deep-learning approach shows a significant improvement in terms of speed with respect to Monte Carlo. Table 5.4 compares the time needed to compute the value of one price with Monte Carlo and by means of the ANN. The gain in speed is striking and of the order of 10000.

Table 5.4: **Computation Time per basket option price in seconds**

	Neural Network	Monte Carlo
Time	1.23e-7	2.54e-3

Lastly, notice that a confidence interval for the Monte Carlo method is in the order of 10^{-4}. This means that the deep-learning approach, because of what was seen in Table 5.3, has the same degree of accuracy as the benchmark method.

5.1.1 Using Chebychev Points

We know that Chebychev points lead to good convergence properties in the polynomial approximation of functions. Hence, we want to empirically see if they present a similar behaviour when they are used as points of the training grid for a multi-layer perceptron.
In addition, Chebychev points will be used in the following Chapter 6 to experiment with tensor completion and compression of the training set, so that this serves as a preliminary exercise.
The Chebychev points on the interval $[1, 1.5]$ are given by:

$$\mathcal{P}_c = \{1.0063, 1.0545, 1.1415, 1.25, 1.3585, 1.4455, 1.4937\},$$

while an evenly spaced grid on the same interval results in:

$$\mathcal{P}_e = \{1.0000, 1.0833, 1.1667, 1.2500, 1.3333, 1.4167, 1.5000\}.$$

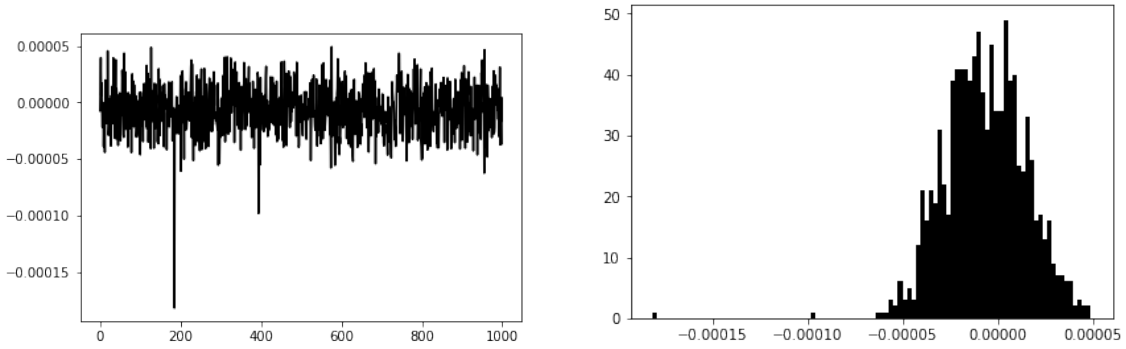
As can be seen, Chebychev points tend to be more dense when moving further from the mid-point. In addition, they cover a slightly smaller range of values.
The structure of the neural nets is in all respects identical to the previous exercise (compare Tables 5.2 and 5.5), but for the activation function of the hidden layers, which is in the Chebychev case taken as a rectified linear unit.

Table 5.5: **Neural network architecture for basket options – Chebychev grid**

	Input Size	Output Size
Linear	5	10
ReLU		
Linear	10	7
ReLU		
Linear	7	5
ReLU		
Linear	5	1
ReLU		

The errors for the Chebychev case are summarised in Table 5.6 and presented in Figure 5.4. The plots show a good behaviour of the deviations from the Monte Carlo prices, conveying the fact that the two approaches can be used equivalently and that Chebychev points can substitute evenly-spaced grids for the training phase.

A comparison between the error metrics between the different choices of the grid confirms that behaviours are similar in the two cases.



(a) Error for each point of the test set, indexed on the horizontal axis

(b) Histogram of the Error

Figure 5.4: *Error values and histogram for basket option prices computed on a test set by a neural net trained on Chebychev points.*

Table 5.6: **Error Metrics for basket options on different grids**

	Evenly Spaced	Chebychev
MSE	4.93e-10	4.98e-10
Maximum Absolute Error	1.94e-4	1.81e-4
MAE	1.68e-5	1.76e-5
MAPE	7.85e-5	7.75e-5

5.2 American Options

This section covers the practical case of American Options. The procedure is similar to that being followed in the case of the basket options, although with a different data generation process and slightly modified neural network architectures.
The goal is again to learn a pricing function for different ranges of the parameters. In our case study, the parameters consist of the Heston parameters κ, θ, σ and ρ and the option strike K. Maturity, initial price of the underlying, initial volatility and risk-free rate are fixed, but could in principle be included in the learning process: this would obviously scale up the dimensionality of the problem, and would probably require a different data-generating scheme. Please refer to the following Chapter 6 for a possible way to tackle the problem.
To formalise the framework, we need to learn a pricing function f such that:

$$f : [2, 2.4] \times [-1, 1] \times [0.2, 0.5] \times [1, 2] \times [0.05, 0.2] \to \mathbb{R}^+$$

$$f : (K, \rho, \sigma, \kappa, \theta) \mapsto \text{price} = f\,(K, \rho, \sigma, \kappa, \theta)\,.$$

Notice that, since the parameters for maturity T, initial price S_0 and initial volatility v_0 we should write in a more meaningful way:

$$f = f_{T,S_0,v_0,r}.$$

The values of the parameter are summed up for the sake of clarity in Table 5.7.

Table 5.7: **Parameters in the American option problem – Grid 1**

	Value or value range	Number of points
K	[2.0,2.4]	10
ρ	[-1,1]	10
σ	[0.2,0.5]	10
κ	[1,2]	10
θ	[0.05,0.2]	10
S_0	2	1
v_0	0.0175	1
T	0.25	1
r	0.1	1

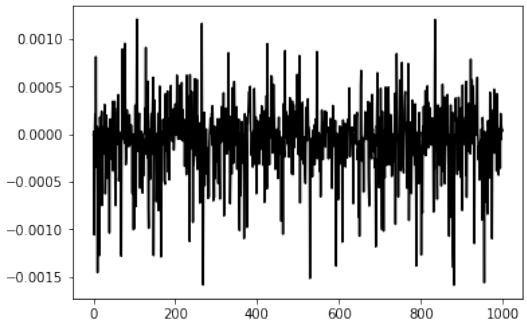
The dataset was generated so as to include 10 evenly-spaced points for each parameter, meaning that the size of the problem is 10^5. For each combination of parameters, the price was computed as the numerical solution of the Heston PDE, adjusted to encompass the American type of the derivative.
Training was performed, as in the basket option case, with a validation test to monitor the tendency of the model to overfit. As one can notice, the dimensionality is higher than for the basket option problem, and, as a result, the size of the network is bigger, as shown in Table 5.8.

Table 5.8: **Neural network architecture for American options**

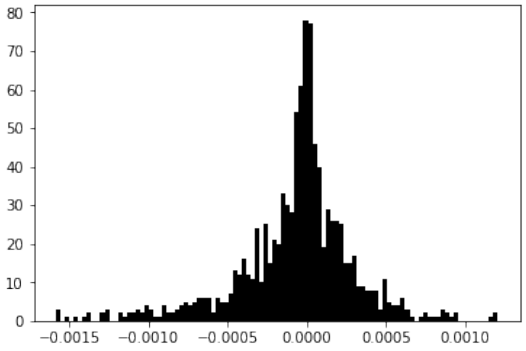
	Input Size	Output Size
Linear	5	50
ReLU		
Linear	50	55
ReLU		
Linear	55	50
ReLU		
Linear	50	45
ReLU		
Linear	45	1
ReLU		

With respect to the previous case (compare with Table 5.2), one should notice an

increase in the number of hidden nodes, as well as an extra hidden layer.
The errors, Figure 5.5a, and their distribution, Figure 5.5b, highlight a good behaviour of the neural net, although slightly weaker than in the basket case. In particular, the histogram points to a satisfactory performance of the learning process, with errors peaking around zero.



(a) Errors for each point of the test set, indexed in the horizontal axis



(b) Histogram of the Error

Figure 5.5: *Error values and histogram for American option prices computed on test set by a neural net trained on Grid 1.*

Furthermore, Table 5.9, confirms intuitions from the graphical representations of the errors, with metrics larger by approximately one order of magnitude with respect to the basket case.

Table 5.9: **Error Metrics for American put options**

	Value
MSE	1.37e-7
Maximum Absolute Error	1.55e-3
MAE	2.51e-4
MAPE	2.05e-3

Lastly, the computational time shows a significant improvement of the neural-net approach with respect to the benchmark case. Results are shown in Table 5.10.

Table 5.10: **Computation Time per American option price in seconds**

	Neural Network	PDE
Time	9.52e-7	4.8e-1

5.2.1 Other Choices for the Grid

Unlike the basket case, one might argue that in the American problem not all the parameters are equally important in affecting the option price. To inspect this issue we run a linear regression, which, however limited, can yield a first insight onto the role of the parameters. Results are shown in Table 5.11 and hint at a predominant contribution of the strike price K and of the long-term mean of the volatility θ in explaining the derivative price.

Table 5.11: **Linear regression of prices on the American option parameters**

	Dependent variable:
	Price
K	0.480^{***}
	(0.006)
ρ	-0.001
	(0.002)
σ	-0.003
	(0.016)
κ	0.008
	(0.005)
θ	0.163^{***}
	(0.031)
Constant	-0.879^{***}
	(0.016)
Observations	1,000
R^2	0.866
Adjusted R^2	0.866
Residual Std. Error	0.043 (df = 994)
F Statistic	$1{,}289.280^{***}$ (df = 5; 994)
Note:	*p<0.1; **p<0.05; ***p<0.01

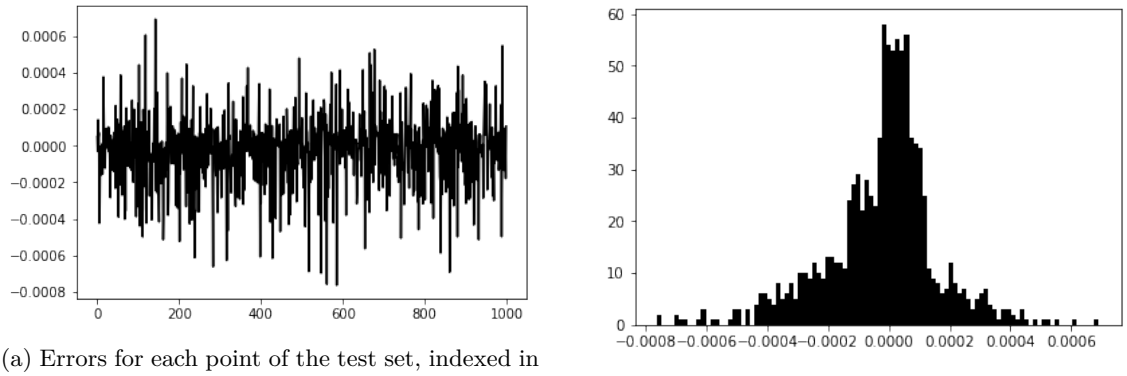
Table 5.11 motivates a different choice for the grid of points than the one proposed before now. Indeed, although still considering evenly-spaced points, we propose rebalancing the weight from the parameters which show less correlation with the option price to those which show more. This means shifting points from the grids of ρ, σ and κ to those of θ and K. For the change to be of some use, the overall

number of points should be comparable to that used in the previous case: in fact, we manage to bring the number of points down by approximately 50% (see Table 5.12).

Table 5.12: **Parameters in the American option problem – Grid 2**

	Value or value range	Number of points
K	[2.0,2.4]	24
ρ	[-1,1]	6
σ	[0.2,0.5]	6
κ	[1,2]	6
θ	[0.05,0.2]	10
S_0	2	1
v_0	0.0175	1
T	0.25	1
r	0.1	1

Results are satisfactory, with MSE down by one order of magnitude and MAE, MAPE and maximum absolute error reduced by approximately one half, as can be seen by Table 5.13.
The neural network architecture is a slight modification of that shown in Table 5.8.



(a) Errors for each point of the test set, indexed in the horizontal axis

(b) Histogram of the Error

Figure 5.6: *Error values and histogram for American option prices computed on test set by a neural net trained on Grid 2.*

Figure 5.6 shows again performances that are slightly worse than in the basket case, but with errors that are more densely distributed around zero than in the previous choice of grid for the American problem. Haentjens and in 't Hout (2015) show that the accuracy of the benchmark ADI method is of the order of $1e-3$ in the maximum norm, meaning that the new methodology is faster while confirming the

Table 5.13: **Error Metrics for American put options**

	Old Grid	New Grid
MSE	1.7e-7	3.41e-8
Maximum Absolute Error	1.55e-3	7.61e-4
MAE	2.51e-4	1.30e-4
MAPE	2.05e-3	9.83e-4

same accuracy.

5.3 Remarks on the MLP architecture

The previous paragraphs showed that deep learning can be a precious resource and a flexible tool in tackling the POP problem. Results were satisfactory in both the basket and the American problem for different choices of the training grid. The flexibility of this method, however, comes at a cost, which has to be measured in the tuning of the neural net structure. Together with cross validation, trial and error was used to obtain suitable parameters for the architecture of neural networks. This section briefly discusses a few of the choices made.

- The mean-squared error was taken as loss function: as well as being a typical choice in the literature, it is well-suited for gradient-based methods. However, other cost functions are evaluated on a test set, namely the MAE and the maximum absolute error.
- Adam was set as the optimiser.
- The learning rate was set as $1e-3$ at the beginning of training phase; then, after approximately 10000 epochs, it was decreased to $1e-4$ and subsequently to $1e-5$, for a larger number of epochs. The idea was to allow for a rapid exploration of the parameter space, and then to identify the good parameters by a more careful search in the domain. This is why the first phase is typically faster, while the second one requires more epochs in order for the mean-squared error to decrease.
- Dropout was described as a powerful resource against overfitting. However, monitoring the error on training set and validation or test set showed that the model does not tend to overfit. While the MSE goes down on the training set, even though sometimes oscillations can occur due to a poor choice of the learning rate, it does so also on the validation set. When this is not the case, either because it stagnates or starts to move upwards, the learning process is interrupted, and the best previous result is recovered from the best model that was saved.
  The lack of overfitting might be due to the fact that the dataset is large enough

for the MLP architecture. Compared to those in the literature, the architectures devised are limited in the number of hidden nodes and modest in the depth. Indeed, when introducing a dropout rate, however limited and despite it being only inserted on one or two hidden layers, the training process failed. This is a feature which is also highlighted by Liu, Oosterlee and Bohte (2019), who put it down to the sensitivity of the price to the input, which is for instance higher than that to the pixels of an image.

- Batch normalisation did not seem to contribute to better training results, as did not input standardisation.
- Regularisation by means of a penalty for the norm of the coefficients in the cost function tended to slow down the learning process, without introducing significant improvements in the loss.

Chapter 6

Learning via an Artificial Dataset

This section proposes a further extension to the neural-net approach. We discussed how neural nets are successful in tackling the parametric option pricing problem, in that they can learn a function which generalises prices on a given grid. We showed different metrics describing the behaviour of the model on fresh data and started to observe that several factors intervene in determining how successfully a model performs.
To begin with, the size of the training set proves crucial, as the multi-layer perceptron fails to generalise when not enough cases are provided for it to learn from. At the same time, inputting more data points is expensive, which is the reason why the thesis was aimed at finding a method not to entirely compute them with traditional approaches in the first place.
Furthermore, the problem is inherently multi-dimensional, which means that increasing the number of nodes implies an exponential increase of the dimensionality of the problem; not to mention the fact that we might want to consider higher-dimensional cases notwithstanding the number of points we strive to funnel into the neural net. For instance, in the case of a basket option, the finance industry can be interested in offering a derivative contract whose number of underlying assets is 25 instead of 5, which was the case described in Chapter 5.1.
These are some of the issues practitioners can face and this chapter is devoted to proposing a general and viable solution to them. Section 6.1 motivates the need for larger training sets, while the following 6.2 explains why an approximated dataset, which we label as synthetic, can be of use in this and other types of applications. Paragraphs 6.3 and 6.4 revisit the examples presented in the previous chapter in the light of the synthetic approach, showing that to work with a dataset obtained by tensor completion would not make practitioners worse off. The same approach is tested on random grids rather than on evenly-spaced ones in Section 6.5. Lastly, Section 6.6 of this thesis shows how to apply both tensor completion and tensor compression to handle problems in higher dimensions. By storing the approximated tensor in TT format, we can train neural nets from huge tensors and achieve adequate error metrics dodging the curse of dimensionality.

6.1 The case for a larger dataset

Consider the American option problem on an evenly-spaced grid consisting of the same number of points along each dimension.
Figure 6.1 shows how the mean-squared error varies with the number of points on the grid. In particular, three different datasets where generated with 6, 8 and 10 nodes for each parameter and several neural nets were trained on them. Next, the models were evaluated on a test set and the metrics of interest were computed. The figure points to a clear advantage of a larger dataset, with the mean-squared error going down by one order of magnitude from the 6 to the 10-node case. However, the difference in the number of points in the two cases is not negligible, as we move from $6^5 = 7776$, through $8^5 = 32768$ to $10^5 = 100000$, the latter demanding a good couple of hours to be generated on a standard machine.

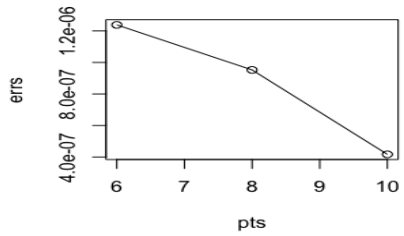


Figure 6.1: *Variation of the MSE for different sizes of the grid in the American option problem.*

In addition, the problem might grow in size even when leaving the number of nodes per parameter fixed. This occurs when one would like to expand the parameter space in the POP problem – for instance by including different values of r and/or different maturities T to the set of varying parameters – or add more assets in the basket. Under this circumstances, we would witness an exponential growth in the number of points, which would hinder the feasibility of the process.

6.2 The case for a synthetic dataset

In the cases described above it might be impractical to retrieve all the points via the benchmark method or even impossible. Indeed, one could face the problem where generating more data is too expensive in terms of time or where data are scarce. The latter case arises in particular with real data for more exotic options: in this context, prices are not always widely available, and all the more so for the different values of the many parameters of interest.

Even when one would rather have the full tensor of prices computed with the benchmark method, however lengthy its generation, it might still be important to start work with a preliminary and fast solution.
Lastly, practitioners might find themselves working with too large a dataset and impossible to store or manipulate with the use standard machines. In these cases, a possible idea can be that of working with approximated and possibly compressed tensors. Tensor completion was described in Chapter 4 and is used here as an intermediate step before learning the pricing function via a neural net. Specifically, the completion algorithm is used to approximate the tensor and store it in the compressed TT format.
The proposed pipeline is hence the following:

- starting from few data points – i.e. only few entries of the tensor – approximate the tensor of prices and obtain a completed tensor;
- use the approximated tensor to train the neural net. The completed tensor can be handled in two ways, based on the context:
  - the full tensor can be retrieved from its compressed version, if the dimension of the problem allows to store all the data;
  - alternatively, it is possible to work with the compressed tensor in TT format.

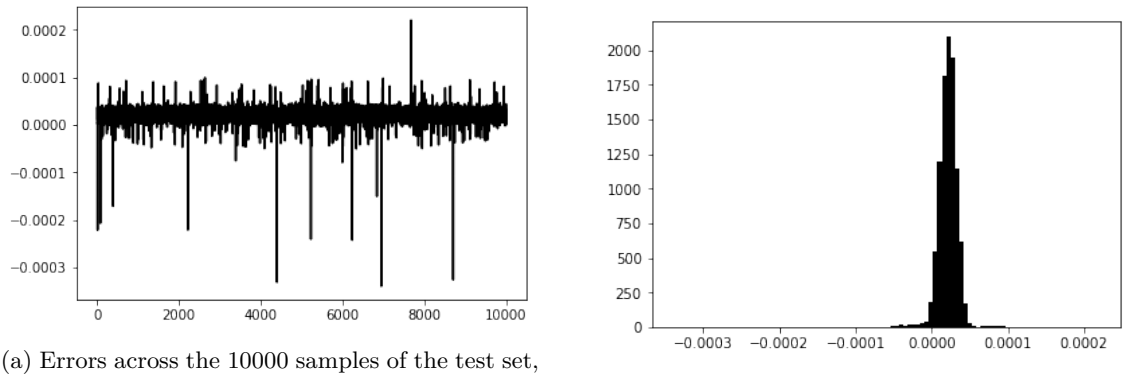
  Both possibilities are explored in the remainder of this chapter.

The process of learning the tensor before learning the pricing function can be seen as a form of bagging, as it is in some respects a combination of two learning methods: at first one tries to form an educated guess – via tensor completion – of the tensor entries, and then to extend the learning process – via a neural net – to the entire parameter space.
We know from Glau, Kressner and Statti (2019) that tensor approximation works well on a Chebychev grid: in the paper, the authors compute option prices via a compressed representation of the Chebychev tensor and perform polynomial interpolation by means of tensor-train products.
As we showed in the previous Chpater, and recalling Table 5.6, it is possible to use a Chebychev grid to train the neural net, as the equally spaced and Chebychev points showed a similar behaviour in terms of the error metrics: we will, however, extend the argument to other possibilities by considering different cases. Specifically, we want to shift the focus so as to encompass in this discussion:

- a general grid instead of a Chebychev one;
- the approximated tensor, both full and compressed, as the training set for the supervised learning process.

6.3 Basket Options

The procedure is experimented on the basket option case where an approximated tensor is used instead of the full one. The tensor in the example consists of a five-dimensional grid with 7 evenly-spaced points on each dimension. Results (see Table 6.1) are encouraging: when compared to the full tensor, a network trained on the approximated tensor shows similar error statistics.



(a) Errors across the 10000 samples of the test set, indexed on the horizontal axis.

(b) Histogram of the Error

Figure 6.2: *Error values and histogram for basket option prices computed on a test set by a neural net. The training dataset was obtained by tensor completion.*

Figure 6.2 confirms the results summarised in the previous table, with errors hardly ever reaching the fourth decimal digit. The plot of the errors highlights the fact that exceptions in the size of the error are limited and infrequent.

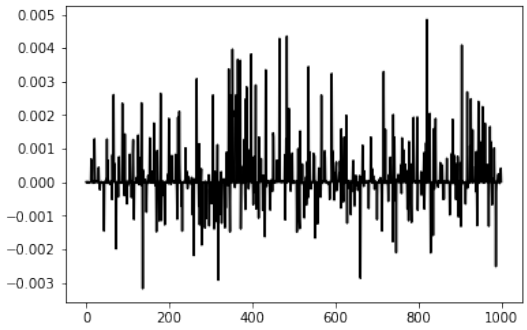
Table 6.1: **Comparison of metrics for full and approximated tensor in the basket case**

	Real Data	Artificial Data
MSE	4.93e-10	6.70e-10
Maximum Absolute Error	1.94e-4	3.39e-4
MAE	1.68e-5	2.23e-5
MAPE	7.85e-5	1.02e-4

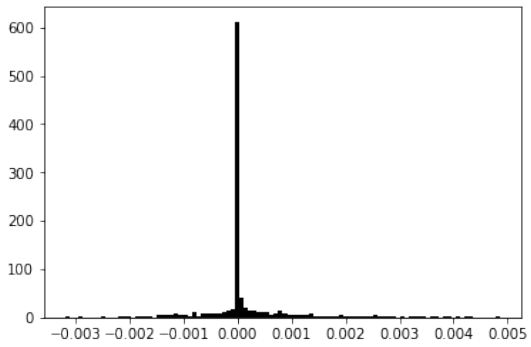
6.4 American Options

The same experiment is performed in the case of American options, with equally satisfying results. The errors are reported in Table 6.2 and suggest that the differences in the values might be due to the stochasticity of the training process rather than to the quality of the training data.
Indeed, while the values are comparable in terms of order of magnitude between real

and synthetic data, the MSE favours real data, while the MAE synthetic ones. Figures 6.3a and 6.3b are also comparable to their real analogue, for which one should go back to the previous section.



(a) Errors across the 1000 samples of the test set, indexed on the horizontal axis



(b) Histogram of the Error

Figure 6.3: *Error values and histogram for American option prices computed on a test set by a neural net. The neural net was trained on a tensor obtained by tensor completion.*

Table 6.2: **Comparison of metrics for full and approximated tensor in the American case**

	Real Data	Artificial Data
MSE	1.7e-7	6.12e-7
Maximum Absolute Error	1.55e-3	4.84e-3
MAE	2.05e-3	3.52e-3

6.5 Random Grid

The current paragraph is devoted to studying the behaviour on a random grid. The main reason for doing so is to show that a dataset can be extended when only few random observations are available: needless to say, this is not always guaranteed to lead to optimal results, but it proves fairly adequate in our case.
We remark that this approach goes beyond the mere learning of a pricing function, and can potentially be applied to different contexts where one seeks to construct a grid from the few available observations.
In our case, it is of special interest when working with real data, which are naturally scarcer for exotic options. To test the effectiveness of this idea, we consider a random grid, as we expect that parameters of real-life options are randomly distributed on the parameters space. At any rate, an application would involve few points with given parameters' values and their relative price: we model this setting by taking few random points on the parameter space and completing the tensor by means of

the completion algorithm, ensuring a prescribed accuracy is maintained.
Figure 6.4 shows that the accuracy of the approximation approaches 10^{-4} on the test set, meaning that the completion algorithm produces a reasonable result.

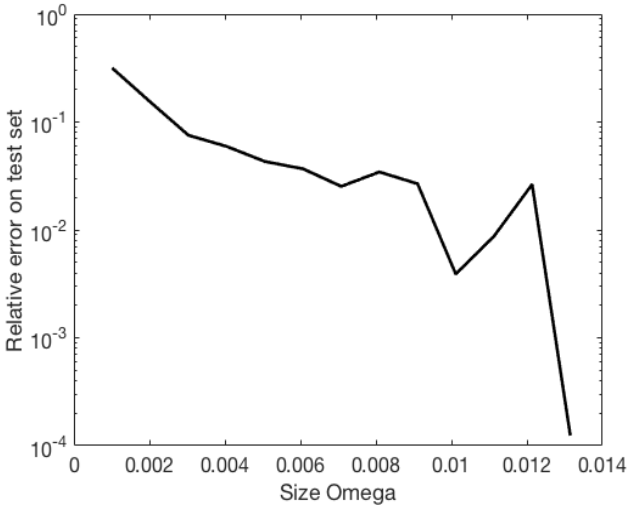
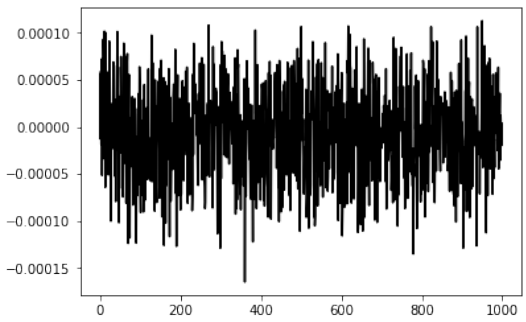


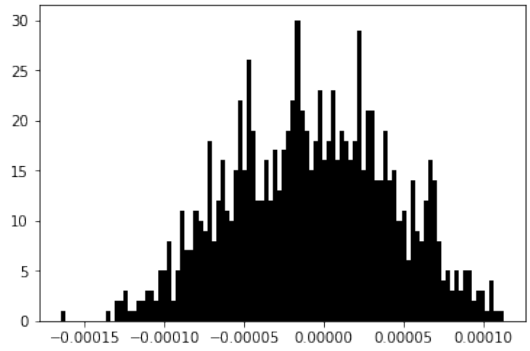
Figure 6.4: *Relative error on varying test sets for different sampling set sizes as percentage of the size of the full tensor in adaptive sampling strategy – random grid case.*

The setting proposed mimics the situation where one finds him/herself with some option prices and wants to build a full grid starting from the few ones available. The constructed full grid is then used to train the neural net, either in a full or in a compressed format.
Results are shown graphically in Figure 6.5 and in Table 6.3 and are promising, as they suggest that adopting a synthetic dataset is not to the detriment of the learning process, even when the grid has a random structure, rather than the regular one seen in the previous case studies.

Table 6.3 compares and sums up the main cases hitherto considered for the training grid: the full tensor of evenly-spaced points entirely computed via the benchmark method, the same tensor, but achieved via tensor completion, and lastly the completed tensor of a grid with a random structure.
The errors metrics are on average in line with those obtained by the tensor of real and of synthetic data, meaning that the new pricing methodology can be extended to the cases mentioned earlier.



(a) Errors across the 1000 samples of the test set, indexed on the horizontal axis



(b) Histogram of the Error

Figure 6.5: *Error values and histogram for basket option prices computed on a test set by a neural net. The neural net was trained on a random and approximated tensor.*

Table 6.3: **Error Metrics – Basket Case**

	Real Data	Artificial Data	Random Grid
MSE	4.93e-10	6.70e-10	2.91e-9
Maximum Absolute Error	1.94e-4	3.39e-4	1.52e-4
MAE	1.68e-5	2.23e-5	4.52e-5
MAPE	7.85e-5	1.02e-4	1.69e-4

6.6 Scaling Up

Until now, this work has showed that deep learning can be used to tackle the parametric option pricing problem.
In Chapter 5 we constructed a full tensor of prices which was used as a training set to a neural net which had to learn a pricing function on the hyper-rectangle of parameters.
The first part of Chapter 6 performed a similar task, with the full tensor of prices replaced by a full tensor of approximated prices obtained by tensor completion. We showed in the experiments that results were only slightly worse, or comparable, when the compressed approximated tensor was used instead of the real one.
However, the examples considered before have not fully exploited the tensor-train decomposition described in Chapter 4. The dimensions of those problems still allowed to make use of the full tensor of prices even with standard machines. For instance, the basket problem involved a tensor of order 5, with 7^5 entries, while the American-option case was based on a 5-order tensor and 100000 entries. As a result, it was still possible to vectorise the full tensor in a one-dimensional vector of prices and a matrix of parameters.
This procedure quickly becomes unfeasible when the number of parameters scales up and the remainder of this chapter is devoted to devising a solution to the curse of

dimensionality. Indeed, when the tensor is of higher orders, memory requirements become dominant and demand for a different encoding of the data. In this thesis, the TT decomposition is presented as one possible way to overcome such problem.

The example studied in the following consists of learning a pricing function for a basket option with 15 and 25 underlying assets. It is easy to understand that memory constraints do not allow to directly store the full tensor of prices. When one considers a grid of five points in the interval of interest, this corresponds to a training tensor of $5^{15} = 30517578125 = \mathcal{O}(10^{10})$ entries in the problem of order 15. The RAM requirement is of 244 GigaBytes, which makes the problem intractable on standard machines.
Solving this issue requires a slightly different formulation of the solution concept than the one which was used in dimension five.
Specifically, the key changes regard:

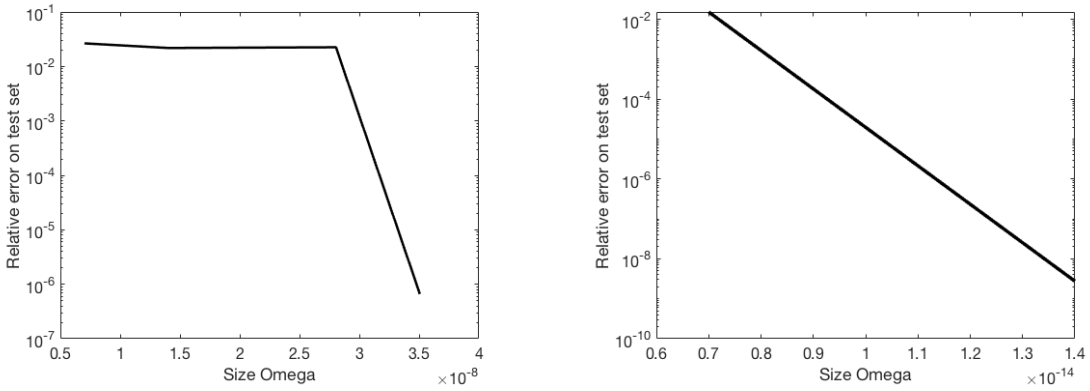
- the tensor encoding;
- the training of the neural network.

As far as the training tensor is concerned, we represent the tensor in TT-format. While previous applications involved computing the approximated tensor in TT format and then vectorising the full tensor to obtain a vector of prices, the extension of the method does not retrieve the full tensor, but relies on its approximation stored in TT-format.
The TT-format allows to have a representation of the tensor without the memory burden which having the full tensor would entail. In addition, the entries of the tensor – the prices for a combination of parameters – are obtained by the product (4.1), a matrix multiplication which can be easily parallelised.

As far as the second modification of the solution structure is concerned, the method involves inputting small batches of the tensor entries (computed via (4.1)) to the neural network. Each epoch hence consists of the training process going through a small sample of the data, and a new subset is retrieved when a new epoch begins. The memory requirements hence vanish, as few of the tensor entries are computed when the training phase needs them. In the example of order 15, the subsets of data were taken to be of size 10000, which is roughly 10^7 times less than the size of the full tensor. Within each subset of the data, mini-batches were taken to solve the problem of the minimisation of the loss via the gradient-based Adam method.

When compressing the tensor in TT format, it is of utmost importance to ensure that the quality of the approximation is adequate. This is measured by resorting to a test set, as described in the Completion Algorithm with Adaptive Sampling Strategy (see Algorithms 11 and 12).

We consider as an example the cases of basket options with 15 and 25 underlying assets. Notice that the tensors associated with these problems are huge and would be intractable without some form of dimensionality reduction. As mentioned, the tensors are compressed, and the training process exploits sampling and the TT-format of the tensor. Despite a predictable slowdown in the training process with respect to the lower-dimensional cases, we will show that the performance of the neural network is comparable and the completion is satisfactory.
Figures 6.6a and 6.6b show the evolution of the error on a varying test set across the different iterations of the algorithm for the completion of an order-15 and order-25 tensor, respectively.



(a) Order 15 (b) Order 25

Figure 6.6: *Relative error on varying test sets for different sampling set sizes in adaptive sampling strategy – tensors of order 15 and 25. The size is expressed as a percentage of the size of the full tensor.*

The Tables 6.4 and 6.5 sum up the characteristics of the completion process for the basket tensors of order 15 and 25. Notice that the completion of the tensor takes place in a reasonable amount of time, in both cases, as shown in the third column. In addition, and more importantly, the amount of memory needed in the two cases decreases strikingly. We defined the compression ratio as the ratio of the points needed for the accuracy to achieve the desired level to the size of the full tensor: the value for this quantity is in the order of 10^{-8} for the case with 15 underlying assets and 10^{-14} for that with 25 The compression ratio can be computed as the ratio of the second column to the fourth column. The second column of the tables show the final size of Ω, namely the number of points needed to achieve an adequate approximation of the full tensor. The memory requirements – in terms of the number of points to be stored – are featured in the last two columns and are computed as follows:

- full storage s_f:

$$s_f = 8n^d$$

- TT storage s_t:

$$s_t = n(r_1 r_2 + \cdots + r_{d-2} r_{d-1}) + n(r_1 + r_{d-1}),$$

  where the last term accounts for the last and the first tensors, which present one less dimension. To go from the number of nodes to the size of RAM needed it suffices to multiply the numbers by 8.

Lastly, the second row details the TT-ranks obtained in the completion process.

Table 6.4: **Completion Results for the order-15 tensor.**

Compression Ratio	Final Size	Time Completion	Full Storage	TT Storage
3.5062e-08	1070	71.53s	30517578125	410
TT Ranks: 1 3 2 2 2 3 2 3 3 3 3 2 2 2 2 1				

Table 6.5: **Completion Results for the order-25 tensor.**

Compression Ratio	Final Size	Time Completion	Full Storage	TT Storage
1.3999e-14	4172	100.51s	298023223876953152	925
TT Ranks: 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 2 2 3 3 2 2 3 1				

The accuracy of the completion measured on the test set is also satisfactory, being in the order of 10^{-7} in the 15-asset problem and in the order of 10^{-9} for the 25-asset problem. Detailed results are reported in Table 6.6, which shows the relative error attained in the last iteration for the two case studies.

Table 6.6: **Accuracy results for the basket tensors.**

	Order 15	Order 25
Relative Error	6.61e-07	2.77e-09

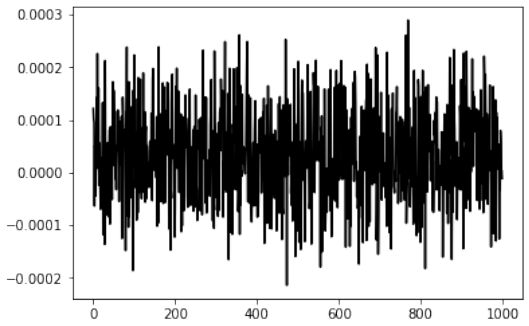
The error metrics are collected in Table 6.7, which shows a comparison of the results for the three problems considered – of orders 5, 15 and 25, although, for the sake of brevity, the completion results were not gathered in a table, as the magnitude of

the compression is less striking.

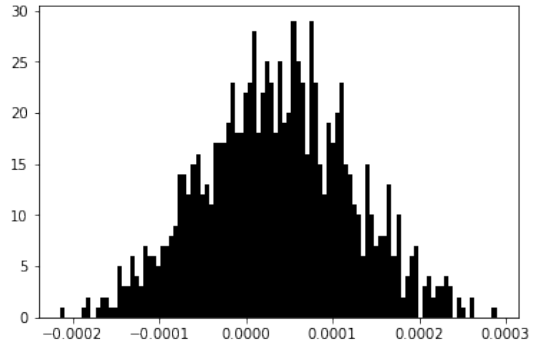
Table 6.7: **Error metrics for the higher-order basket problems.**

	Order 5	Order 15	Order 25	Order 25 Uncorrelated Assets
MSE	3.08e-7	3.41e-08	8.61e-9	1.92e-8
Maximum Absolute Error	1.65e-3	5.16e-4	2.89e-4	3.85e-4
MAE	4.5e-4	1.56e-4	7.46e-5	1.12e-4
MAPE	4.5e-4	6.36e-4	3.13e-4	4.56e-4

To begin with, observe that the dimensionality no longer represents a bottleneck in the training process. In addition, the increased number of points does not result in lower performance metrics. As seen in previous examples, the error in the worst-case scenario still enters after the fourth digit. Mean-squared error is also decreasing with the order of the problem, and is in the order of 10^{-7}, 10^{-8} and 10^{-9} for the orders 5, 15 and 25 respectively. This is due to the fact that the increased size of the data results in a better accuracy of the model rather than hindering the learning process. As a last remark, we also consider the case where the assets' dynamics exhibits a correlation. Introducing a covariance in the processes does not impact severely on the performance metrics of the neural net, as shown by the last column of Table 6.7. The neural network architecture consisted of four hidden layers and 75 hidden nodes for each of them.
The plot of the errors is shown in Figure 6.7 for the order-25 problem. Although the peak around zero is less striking than in previous plots, the absolute numbers still look adequate, highlighting that the learning process proceeded successfully.



(a) Errors for each point of the test set, indexed in the horizontal axis



(b) Histogram of the Error

Figure 6.7: *Error values and histogram for the order-25 basket option prices computed on a test set.*

Chapter 7

Conclusions

The first part of this work consisted of employing different computational methods to obtain prices of different financial products. The first application involved basket options, for which the Black-Scholes model was used. Conversely, American put options were coupled with the Heston model. In the former case we exploited the representation of prices as expectations, what is commonly referred to as the risk-neutral valuation formula, while the price as a solution to the model PDE was used in the case of American options.
The different representations of prices lead to different solution concepts, Monte Carlo and finite differences for PDE, both being widely adopted in the financial industry. Monte Carlo is certainly the reference method due to its ease of application, but presents several drawbacks, the computational burden on top. The method requires several simulations to guarantee adequate results, so that it becomes a bottleneck when it has to be employed repeatedly. Finite differences do not fare better, but recent developments have allowed for extensions in more dimensions. The application we considered is two-dimensional in space, and can be made faster with alternating direction schemes, as was shown in Section 2.2.

We discussed in the beginning how exact solutions are simply not available for more complex models and financial products, as things become more challenging when moving away from vanilla options under Black and Scholes. However, numerical solutions also present downsides which are hard to disregard when using them extensively and repeatedly.
Despite advancements and refinements of traditional numerical methods, this motivates, in the framework of parametric option pricing (POP), a further degree of approximation other than that of numerical solutions: the approach followed, which is well-established in the literature, consisted of obtaining prices for the financial product of interest on a grid, and generalising it on a domain for the parameters. As a result, after numerically computing prices on a given set of points (with Monte Carlo or finite differences), the goal is to be able to retrieve prices also for points which are not part of the grid. In other words, we have an offline phase with a view

to simplify traditional numerical pricers, and an online phase which allows to easily compute the price by means of a simplified model.
The offline one is inherently more expensive, but is performed only once and combines numerical methods for the grid generation with interpolation or learning for the generalisation. The result is that the onus is shifted from the online to the offline part, so that when prices are really needed – online – they are readily available in instants, with high accuracy.
Interpolation has been successfully employed, especially of Chebychev type. However, recent developments in computational performances of computers, made possible by GPUs and TPUs, have suggested that a viable way to go is with neural networks, whose representation power has been formally established in the last two decades.
As first proven by the pioneering works of Cybenko and Barron, and later by further results due to many others, the advantages of neural networks are certainly flexibility, speed and accuracy, all of which are highly sought after by the financial industry.
In its second part, this work showed the potential of the neural network approach, its limitation and a possible solution to it. The advantages of deep learning are summed up in the following paragraphs.

Speed is certainly central. As mentioned in the introduction to this thesis, the finance industry needs fast results in virtually all of its tasks: ever-changing market scenarios and information transmitting themselves so rapidly mean that financial products have to be updated in real time to ensure no losses occur. Calibration, which was presented as a motivating example in the first lines of Chapter 1, is only one of the many examples in which speed matters: saying that to work with a properly calibrated model is important is possibly an understatement. The same holds for hedging or for the computation of the Greeks. The neural-network approach performed approximately 10000 times faster than Monte Carlo in the Basket experiment (see Table 5.4) and similarly for the American case.

Accuracy is also a highlight of the neural-network approach. The method was tested against the benchmark methods, Monte Carlo for basket options and FD for American options. The goal is to see whether learning the pricing function from few points on the grid is fundamentally different than computing each single price via the reference method – assuming that was doable in the first place.
The basket option results, for which see for instance Table 5.3 and Figures 5.3, show that the significant gain in computational time does not make the method less competitive than Monte Carlo. Mean-squared error is in the order of 10^{-9}, mean absolute error is $\mathcal{O}(10^{-5})$ and in the worst case of the 10000-sample test set, the error only showed up in the fourth decimal digit. In addition, these results are in line with respect to Chebychev interpolation, which, however, demands a specific grid type, making it less versatile. Ferguson and Green (2018) also consider a basket-option

task with neural networks which are deeper and larger. Accuracies in their case are slightly lower in absolute terms, despite working with different magnitudes of the initial prices.
American options are traditionally more complex to deal with, and examples in the literature are harder to come by. Oosterlee and co-authors (Liu, Oosterlee and Bohte (2019)), for instance, consider vanilla options in the Heston and Black-Scholes model in a similar problem to that analysed here. The study proposes a different choice for the training set, obtained by Latin hypercube sampling and featuring a much larger size. The neural networks proposed are also significantly larger, though not deeper, and a restricted test set is taken with respect to the training set. Also in their case study, neural networks perform well, with gains in computational speed not going to the detriment of the desired accuracy.

The final section of this work has tackled potential issues and relative solutions of the neural-network approach, having discussed the theoretical background in Chapter 4 and presented the results in Chapter 6.
The main issue arises when the dimensions of the problem scale up: if a grid is constructed with n points in each dimension, it is easy to see that the size of the grid will soon reach prohibitive dimensions, $\mathcal{O}(n^d)$ for a d-dimensional problem. In this case, generating and storing the whole nodes of the grid becomes problematic, hindering the success of the machine-learning approach due to reasons of time and memory. The solution proposed in this work consists of constructing the tensor of nodes by low-rank approximation starting from a subset of points and to store it in compressed TT format. The completion algorithm described in Section 4.3 allows for the approximation of high-dimensional tensors with a prescribed accuracy: by working with an approximated tensor we are able to train a neural network with extremely limited consequences on the error metrics.
The advantages of an approximated tensor go beyond the computational gains attained when generating the training grid. As stressed by Hutchinson, Lo and Poggio (1994), neural networks have limited scope with real data in pricing financial products as prices are scarce for many options. Tensor approximation could allow for the creation of a synthetic dataset from the few available prices on the market.
In fact, the approach can be of use also beyond the financial sector. Neural nets are becoming the leading learning method across virtually any conceivable domain. For many of these, however, data are hard to retrieve. By exploiting the manifold structure of a tensor one could extend the available data via tensor completion.
Glau, Kressner and Statti (2019) showed that the approach is successful for Chebychev grids in very high dimensions (up to 25). In our work, we experimented with more general grid types – evenly spaced and random – to show that the method can be of use for most data configurations. This solution concept, which we called a third-degree approximation in the pricing process, leads to error metrics that are still adequate and can deeply extend the scope of the neural net approach to encompass very high-dimensional problems. In addition, the accuracy of the approximation

can be tailored so as to match the goals of the application, making the method extremely versatile.
Results in Table 6.3 showed that synthetic data (obtained from the approximated tensor) still lead to a mean absolute error which is in the order of 10^{-5}, hinting at the fact that the approach can be easily extended to any type of data. The random grid, indeed, mirrors the case where a few samples obtained from the real world are used to generate a full-tensor training grid.

The last experiments considered fully exploited the TT decomposition in the training process. By storing the tensor in TT format, it was possible to provide a solution to the curse of dimensionality, as the high-order tensors were replaced by more compact three-dimensional ones. By doing so, this thesis showed how high-dimensional problems (up to 25) can be solved with standard machines and how training sets which would otherwise require hundreds of GigaBytes of RAM can still be used in the learning process. Results confirmed that training is successful when one exploits the TT format of the tensors coupled with stochastic optimisation algorithms. The samples needed in the batch-based training can be easily retrieved when needed from the tensor in TT form, meaning that the memory ceases to represent an issue.

Follow-up experiments could be oriented to further scaling up the problem, so as to extend the pricing function to more parameters of the problem. This would increase the generalisation to a higher extent.
In addition, as the deep-learning approach was successful for the pricing problem of American and Basket options, we claim that it can be exploited for other exotic options which have received less attention in the literature.

Bibliography

Absil, P.-A., Mahony, R., and Sepulchre, R. (2008). *Optimization algorithms on matrix manifold*, Princeton University Press, Princeton, NJ.

Altman, E. I., Marco, G. and Varetto, F. (1994) *Corporate distress diagnosis: Comparison using linear discriminant analysis and neural networks (the Italian experience)*, Journal of Banking and Finance, 18, 505—529.

Baldi, P., Hornik, K. (1989). *Neural networks and principal component analysis: Learning from examples without local minima*, Neural Networks, 2, 53–58. 286.

Barron, A. E. (1993) *Universal approximation bounds for superpositions of a sigmoidal function*, IEEE Transactions on Information Theory 39(3):930 - 945.

Bayer, C., Stemper, B. (2018) *Deep calibration of rough stochastic volatility models*, Preprint at: `https://arxiv.org/pdf/1810.03399.pdf`.

Böjers, L. C. (2010). *Mathematical Methods of Optimization*, Studentlitteratur.

Brandimarte, P. (2014). *Handbook in Monte Carlo Simulation: Applications in Financial Engineering, Risk Management, and Economics*, John Wiley & Sons, NJ.

Brandimarte, P. (2016). *An Introduction to Financial Markets: A Quantitative Approach*, John Wiley & Sons, Hoboken, NJ.

Bühler, H., Gonon, L., Teichmann, J. and Wood, B. (2018) *Deep Hedging*, available at `arXiv:1802.03042`.

Burkovska, O., Glau, K., Mahlstedt, M. and Wohlmuth, B. (2018) *Complexity reduction for calibration of American options* Forthcoming in J. Comput. Finance, `https://arxiv.org/abs/1611.06452`.

Culkin, R. and Das, S. R. (2017). *Machine Learning in Finance: The Case of Deep Learning for Option Pricing*, Journal of Investment Management.

Cybenko, G. (1989). *Approximation by Superpositions of a Sigmoidal Function*, Math. Control Signals Systems (1989) 2:303-314, Springer-Verlag, New York.

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio Y. (2014) *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*, Advances in neural information processing systems 2933 - 2941.

De Spiegeleer, J., Madan, D. B., Reyners, S. and Schoutens, W. (2018) *Machine Learning for Quantitative Finance: Fast Derivative Pricing, Hedging and Fitting*, Available at SSRN: https://ssrn.com/abstract=3191050 or http://dx.doi.org/10.2139/ssrn.3191050

Dingeç, K. D., Hörmann, W. (2013). *Control variates and conditional Monte Carlo for basket and Asian options*, Insurance: Mathematics and Economics 52(3):421 - 434.

Ferguson, R., Green, A. D. (2018). *Deep Learning Derivatives*, Available at SSRN: https://ssrn.com/abstract=3244821 or http://dx.doi.org/10.2139/ssrn.3244821.

Gallier, J. (2004), *Manifolds, Riemannian Metrics, Lie Groups, Lie algebra, and Homogeneous Manifolds With Applications to Machine Learning, Robotics, and Computer Vision*, CIS610 at UPenn, Spring 2018, available at http://www.cis.upenn.edu/ cis610/cis610-18-sl1.pdf.

Gass, M., Glau, K., Mahlstedt, M. and Mair, M. (2018) *Chebyshev interpolation for parametric option pricing*, Finance Stoch., 22 (2018), pp. 701–731, http://dx.doi.org/10.1007/s00780-018-0361-y.

Giles, M. B. (2015) *Multilevel Monte Carlo methods*, Acta Numer., 24, pp. 259–328, http://dx.doi.org/10.1017/S096249291500001X.

Glasserman, P. (2004), *Monte Carlo Methods in Financial Engineering*, Springer, New York.

Glau, K., Kressner, D., and Statti, F. (2019), *Low-rank tensor approximation for Chebyshev interpolation in parametric option pricing*, pre-print at https://arxiv.org/abs/1902.04367.

Glorot, X. and Bengio, Y. (2010). *Understanding the difficulty of training deep feedforward neural networks*, AISTATS'2010.

Goodfellow, I. J., Vinyals, O., and Saxe, A. M. (2015). *Qualitatively characterizing neural network optimization problems.*, International Conference on Learning Representations.

Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*, The MIT Press, Cambridge, Massachusetts, London, England.

Haentjens, T. and in 't Hout, K. (2015). *ADI Schemes for Pricing American Options under the Heston Model*, Applied Mathematical Finance 22:3, 207-237.

Hesthaven, J. S., Rozza, G. and Stamm, B. (2016) *Certified reduced basis methods for parametrized partial differential equations*, SpringerBriefs in Mathematics, Springer, Cham; BCAM Basque Center for Applied Mathematics, Bilbao, http://dx.doi.org/10.1007/978-3-319-22470-1, BCAM SpringerBriefs.

Heston, S. L. (1993). *A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options*, The Review of Financial Studies, Volume 6, Issue 2, April 1993: 327-343.

Holtz, M. (2011) *Sparse grid quadrature in high dimensions with applications in finance and insurance*, vol. 77 of Lecture Notes in Computational Science and Engineering, Springer-Verlag, Berlin, http://dx.doi.org/10.1007/978-3-642-16004-2.

Horvath, B., Muguruza, A. and Tomas, M. (2019). *Deep Learning Volatility*, The Review of Financial Studies, Volume 6, Issue 2, April 1993: 327-343.

Hutchinson, J., Lo, A. W. and Poggio, T. (1994). *A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks*, The Journal of Finance, 49: 851-889, doi:10.1111/j.1540-6261.1994.tb00081.x.

Ikonen, S. and Toivanen, J. (2008). *An Operator Splitting Method for Pricing American Options*, Partial Differential Equations 10.1007/978-1-4020-8758-5 –16.

Ioffe, S. and Szegedy, C. (2015) *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, International Conference on Machine Learning (ICML).

Kemna, A., Vorst, A. (1990). *A pricing method for options based on average asset values*, Journal of Banking & Finance 14, 113–130.

Kingma, D. and Ba, J. (2014). *Adam: A method for stochastic optimization*, arXiv:1412.6980.

L'Ecuyer, P. (2009). *Quasi-Monte Carlo methods with applications in finance*, Finance Stoch., 13, pp. 307–349, http://dx.doi.org/10.1007/s00780-009-0095-y.

Liu, S., Oosterlee, C.W., Bohte, S.M. (2019) *Pricing Options and Computing Implied Volatilities using Neural Networks*. Risks 2019, 7, 16.

McGhee, W. A., (2018). *An Artificial Neural Network Representation of the SABR Stochastic Volatility Model*, Available at: http://dx.doi.org/10.2139/ssrn.3288882.

Oseledets, I. V. (2011). *Tensor-train Decomposition*, SIAM 33:5, 2295-2317.

Rouah, F. D. (2013). *The Heston Model and Its Extensions in Matlab and C*, Wiley.

Sirignano, J., and Konstantinos, S. (2018) *DGM: A deep learning algorithm for solving partial differential equations*, Journal of Computational Physics.

Steinlechner, M. (2016). *Riemannian Optimization for High-Dimensional Tensor Completion*, SIAM J. Scientific Computing, 38.