Boosting financial models calibration with deep neural networks

Università degli studi di Verona

Oleksandr Honchar

March 2019

Contents

Acknowledgement

Of course, all the results behind this research are not my sole effort.

First if all I would like to thank my thesis advisor Prof. Luca Di Persio for showing me the opportunities and challenges not just in mathematical finance, but in the general research world. I was familiar with machine learning before from the purely applied point of view, but Prof. Di Persio showed me how to develop and finalize research ideas and opened my eyes to a big picture that lies behind the algorithms and their implementations.

I also am very grateful to Prof. Giandomenico Orlandi for giving me the opportunity not just to study the official program, but also share my humble experience with other students - it gave me amazing experience of teaching and sharpened my own knowledge while I was preparing the lectures.

Also, I'd like to thank all the University of Verona professors from whom I had the honor to learn. You are outstanding professionals and the knowledge you shared with me indeed brought me to the next level as a professional and as an individual.

Special warm thanks are headed to my family, my girflriend and my closest friends. During the master's degree and working on this thesis there were ups and downs, but you were always with me with support, encouragement, and love. I couldn't finish it without you all. Thank you.

Oleksandr Honchar

Introduction

This work is devoted to making a connection between classical mathematical modeling and the latest developments in machine learning. To make sense of the use of mathematical models, their parameters have to be adapted to the empirical data that is modeled. This process is also called "parameters estimation" or "inverse modeling", depending on different literature sources. For example, having a set of empirical observations of some object's motion and a supposed underlying physical model describing it, velocity being a parameter, we want to describe its behavior, exploiting the model itself as well as "historical data" of recorded speed's volumes, applying both analytical (e.g. in form of closed solutions coming from classical theory of motion) and statistical (e.g. data analytics and forecasting) approaches. Analogously, one can be interested in fitting problems related to the parameters validation for specific statistical methods and/or stochastic processes used to model, represent and/or predict phenomena arising in a numerous sets of applications ranging from Biology (see [1]) to Finance, from social sciences to renewable energy sectors (see examples of the stochastic models in [2]), etc.

In the case of financial models, the parameters need to be calibrated to the current market condition and with respect to historical prices. As an introductory example, we can consider options pricing when underlying asset is described by the Black-Scholes model (see section 2.2 for more mathematical details). This model depends on a single pricing parameter, namely, volatility and the aim of calibration is to find such a volatility value, that describes current market prices the best. This value is also often called implied volatility. In most of the cases, this problem is stated as an optimization problem: the minimization of errors between modeled data and real market ones. However, because of general non-convexity of this optimization problem and large space of parameters, it is usually solved numerically and

it takes a significant amount of time that grows with financial models parameters space or amount of data. Therefore, the calibration process for a portfolio of different financial instruments can take hours of computational time even with the use of multiprocessing on several CPUs or even GPUs running on high performance computers.

While most of today's applications of Machine Learning (ML) are related to performing classification or regression problems are given very complex data, this work shows another promising objective that deep neural networks are able to solve - the approximation of complex functions, when depending on time variable, can be even processes, also of stochastic nature. Since this work is related to the optimization problem, the ability to successfully approximate exact solutions of other optimizers is being researched. We show that for Black-Scholes model and Merton jump model for options pricing neural networks can efficiently approximate solutions from numerical optimization algorithms with a significant speedup and a negligible loss of accuracy. Although, the common fear for the use of ML in important applications is related to problems with explaining obtained results. Hence, we show the ways to interpret the parameters provided by the Neural Networks (NNs) and perform visual sensitivity analysis for easier use of human analysts and experts. Last but not least, since the market conditions are continuously changing, we provide an efficient scheme for scheduling retraining of the NNs to adapt them to the market changes.

Chapter 1

Problem overview

The goal of this chapter is to introduce the main tools and abstractions that are driving this research. It starts with a description of two of the most known and still widely used financial mathematical models, namely the Black-Scholes model (see [3] and section 2.2) and Metron jump diffusion model (see [6] and section 2.3 for further details) and how they're used in practice. Then we state the general calibration problem and main applications. In particular, we review the main current approaches to the calibration, alongside with their advantages and disadvantages, while, at the end, the ML framework for the general calibration problem is introduced.

1.1 Financial models basics and examples

Broadly speaking, financial mathematical models aim to describe the behavior of some objects of interest in the financial world, e.g. assets, commodities, their derivatives, structured portfolios, etc. No matter about the particular setting, such models are characterized by parameters, that define the properties of the process that is modeled. In Finance, this is the case of, e.g., volatility of the underlying asset, probability of price jumps, mean reversion coefficient etc. Usually, these parameters are parts of Stochastic Differential Equations (SDEs). Generally, the SDEs describe a change of a price of an instrument S_t on a time t with respect to economical constraints and assumptions about the stochastic process that describes evolution of prices over time. In this work, we focus on the derivatives as main instruments, beacuse of their preminent relevance within main financial scenarios.

1.2 Introduction to model calibration in finance

Any mathematical model is useless in practice unless the "right" parameters are chosen. A simple example is the Black-Scholes formula for call option pricing (the full Black-Scholes model is to be introduced in section 2.2) which takes as input six variables: initial price level of the asset S_t, the its volatility σ, the strike price K of the option, T is the expiration date or the horizon, time-to-maturity $T - t$, the bank account risk rate r and in some scenarios the dividends paid by the underlying, the following being its output

$$C(S_t, t) = N(d_1)S_t - N(d_2)PV(K) \tag{1.1}$$

which gives the price of an call option and where:

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}(ln(T) + (\frac{\sigma^2}{2} + r)(T-t)), \tag{1.2}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}, \tag{1.3}$$

$$PV(K) = Ke^{-r(T-t)}, \tag{1.4}$$

and N is the cumulative distribution function of the standard normal distribution. If you put in numerical values for these variables, the formula 1.1 returns a value for the option at the time t. Using a model with incorrect parameters, e.g. underestimated volatility σ likely leads to losses.

Therefore, pricing financial instruments is a fundamental issue that often leads out to very complex computational processes, often involving Monte Carlo simulations (see [5]). In some cases, optimal parameters can be found analytically, but in most of the cases, this problem is stated as an optimization one. Then, it becomes necessary to define appropriate cost functions as functions, measuring the difference between modeled behavior and actual market behavior that needs to minimize. Different choices of this function will be shown in the next chapters (in particular, in section 3.1), like mean

square error, mean average error and the Huber loss.

When the loss function is convex, i.e. with a single extremum and even if the analytical solution is not available, numerical optimization methods, like gradient descent, Newton method, conjugate gradients, etc can be applied. But, in most of the cases, the cost function is neither convex, nor differentiable. Hence, heuristic optimization methods are applied, for example, brute force searches [9], genetic algorithms [12], Bayesian optimization [13] (see section 3.2 for further details).

1.3 Machine learning for calibration basics

The main disadvantage of above-mentioned algorithms is their execution speed. Depending on the problem size, they take a significant amount of time that grows with specific financial models, because of large parameters space or amount of data. For example, the length of the option chain or degrees of freedom of specific financial model can be such complications. ML can be helpful here to approximate the final result of the optimization process. There are a lot of theoretical results (like the approximation theorem, see [14]) and empirical evidences that machine learning algorithms, like deep neural networks (see, e.g., [15]), are able to approximate very complex functions with high accuracy after being suitably trained on historical data.

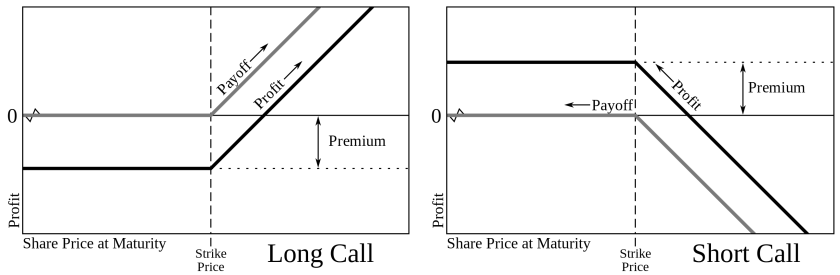
This work is devoted to such kind of approach to financial models calibration: having a dataset of a financial model optimized on historical trades, we train a machine learning model to derive the optimal parameters for the future calibrations. Instead of current calibration routine that involves iterative optimization algorithms, that on each iteration have to evaulate the function, that is based on Monte-Carlo simulations, we will perform a single forward pass of a ML model, that is, in the case of NNs, just a set of matrices multiplications. The expected speedup supposed to be drastic. In fact, most of ML models perform prediction stop in milliseconds, while optimization process with genetic algorithms can last for hours, when applied for structured portfolios.

Chapter 2

Financial models

2.1 Options pricing basics

An (European) option is a contract which gives a holder the right, to buy or sell an asset (also called underlying) at a particular price (called strike price) on a specified date, depending on the type of the option. What's important, the trader has right to exercise the option, but has no obligation to do it. There also exist another different types of options like American option, that allows to execute in any time before the maturity date, Bermudan option that allows to exercise on a set of specific dates and others (see [9] with more examples and details). In this study we will concentrate on European style option.



(a) Profits from buying a call (b) Profits of writing a call

Figure 2.1: Call option profits

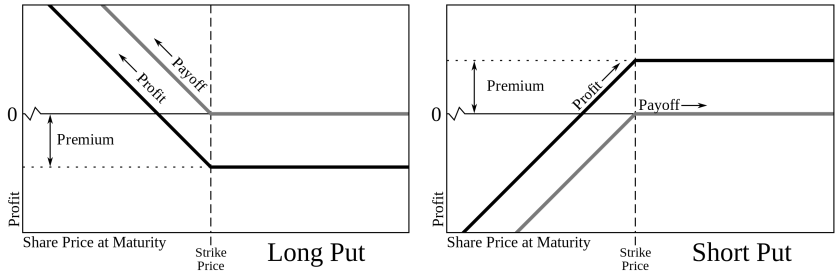
There are two main types of European options, called call and put. A

call option gives the owner of the option the right to buy an underlying asset at some particular price nevertheless the actual price of the asset is. A call option is worth money if the price of the underlying at maturity, denoted by S_T , is higher than the strike price K, otherwise it's worth zero. Hence, its payoff equals

$$C_T = max(0, S_T - K). \tag{2.1}$$

Consequently, a put option is the right to sell an underlying. A put makes money when the asset is below the strike price K at maturity, otherwise it's worth nothing as well, then we have

$$P_T = max(0, K - S_T). \tag{2.2}$$



(a) Profits from buying a put (b) Profits of writing a put

Figure 2.2: Put option profits

Figures 2.1 and 2.2 represent the payoff graphs of a put and a call at maturity T.

The unknown variable in equations 2.1, 2.2 remains S_T - value of an underlying at maturity time T. Our goal now is to compute it. Usually it's done with help of stochastic models that are meant to describe the behavior of underlying asset. In the following two subsections we will recall the basics of the most celebrated, but still actively used in practice, financial models, namely Black-Scholes model and the Merton jump model.

2.2 Black-Scholes model

The Black-Scholes model [3] was introduced to describe dynamics of a derivative price (European option in our case) over time. The core accomplishement is well-known Black-Scholes equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} = 0, \tag{2.3}$$

where S_t is the price of underlying asset at time t, $V(S,t)$ is the value of an option for an asset S at time t, K is the strike price, r - annualized risk-free interest rate. From Black-Scholes formula prices for European call and put options can be derived in a closed form [3]. The formula for call option was already introduced in equation 1.1, the formula for the put option is the following:

$$P(S_t,t) = N(-d_2)PV(K) - N(-d_1)S_t, \tag{2.4}$$

where d_1, d_2 and $PV(K)$ are obtained as in equations 1.2, 1.3 and 1.4.

Black-Scholes model assumes, that underlying prices follow the Geometric Brownian Motion (GBM, see [4]), namely:

$$\frac{dS_t}{S_t} = \mu dt + \sigma dW_t, \tag{2.5}$$

which admits following solution:

$$S_t = S_0 e^{((\mu - \frac{\sigma^2}{2})t + \sigma W_t)}, \tag{2.6}$$

where μ and σ are the expected return rate and volatility of the underlying asset, W_t is the standard Brownian motion and S_0 is the underlying price on the starting time. Example of the sample paths produced by GBM model are represented in the figure 2.3.

Moreover, the Black-Scholes assumes that:

- there are no arbitrage opportunities. The absence of arbitrage opportunities means that all risk-free portfolios must earn the same return;
- the underlying asset will pay zero dividends during all the life of the option;

- the risk-free interest rate r and the asset volatility σ are constants over the existence of the option;
- trading is done continuously. Short selling is allowed, the assets are divisible;
- there are no transaction costs and no taxes related to hedging.

It worth to mention, that not every financial model allows a closed form solution with respect to derivative pricing, as example, within Merton jump model framework, that will be reviewed in detail in section 2.3, analytical solution can't be derived even for simple European style options. In such cases, a typical solution is permitted by adopting Monte-Carlo approach, whose main steps are:

- divide whole time interval from purchase date to expiry date into equal subintervals
- start iteration from i = 1 to I, where I is number of Monte Carlo simulations

  for every timestep t a random number $z_t(i)$ is drawn

  calculate S_T at final time T running equation 2.4 for all t

  calculate value of an option using 2.1 and 2.2
- sum up the inner values, average, and discount them back with the riskless short rate

To find S_T itself, we can use Euler discretization of the underlying process SDE. For example, when considering the GBM SDE in equation 2.5 it's the following:

$$S_t = S_{t-\Delta t} e^{(r-\sigma^2/2)\Delta t + \sigma\sqrt{\Delta t} z_t} \tag{2.7}$$

where z_t is a standard normally distributed random variable, and Δt is suitable chosen time step whose value takes into account the desired accuracy and affordable computational efforts.

Of course, the Black-Scholes model can't capture all the details of underlying process' nature, but since we are concerned with parameters calibration problem, it will serve as a proof of concept model because of its simplicity.

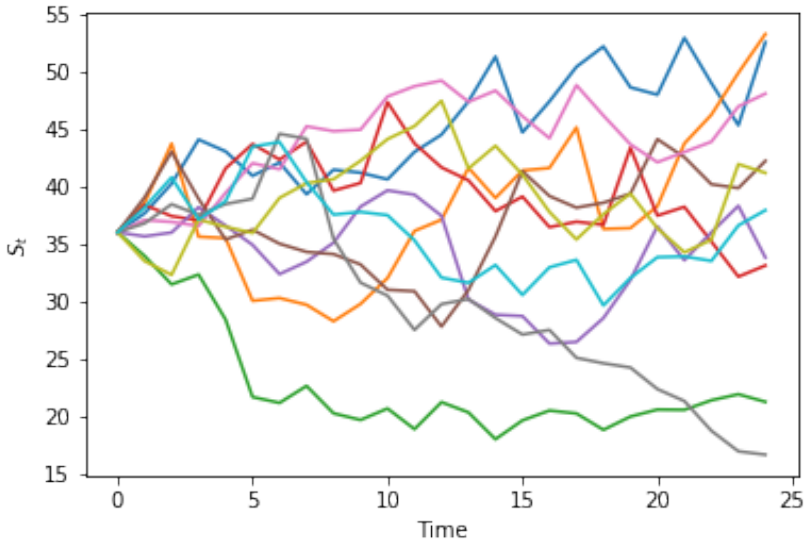


Figure 2.3: Samples of the Geometric Brownian Motion with $\sigma = 0.25$

2.3 Merton jump model

The Merton jumps model, introduced in 1976 by Robert Merton, (see [6]), is a model for stock price behavior that incorporates the idea that prices can change more strongly, opposed to what can happen with the Black-Scholes framework. In particular, these movements are supposed to have the so-called "jump" structure. The consideration of jumps allows for more realistic scenarios that are not incorporated in the standard Black-Scholes model. This causes option prices to increase compared to the Black-Scholes model and to depend on the risk aversion of investors. The SDE defining the underlying asset prices in Merton model is the following one:

$$dS_t = (r - r_J)S_t dt + S_t dZ_t + J_t S_t dN_t \tag{2.8}$$

where S_t is the underlying price at time t, r are the constant riskless short rate, r_j is the drift correction for the jump component to maintain risk neutrality (see [7] for more details),

$$r_J = \lambda(e^{\mu_J + \delta^2/2} - 1) \tag{2.9}$$

σ - is the constant volatility of S, Z_t - is a standard Brownian motion, J_t

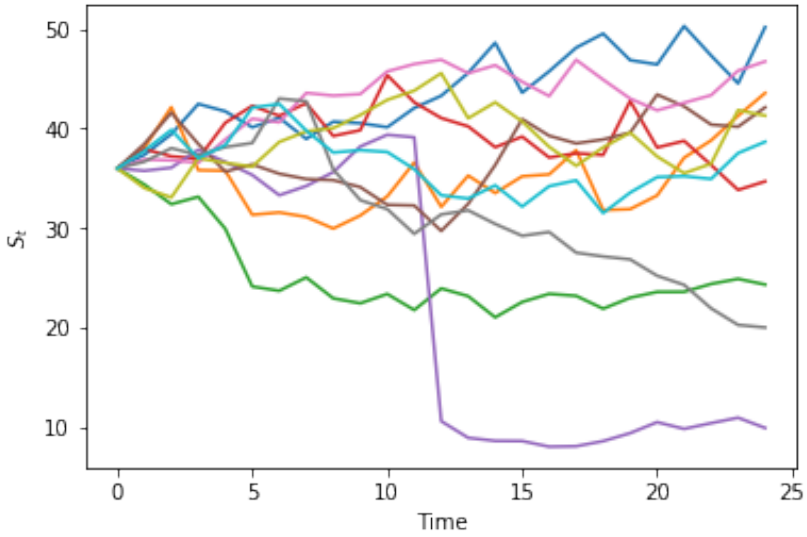


Figure 2.4: Samples of the jump diffusion process with $\sigma = 0.25$, $\lambda = 0.3$, μ = -0.75 and $\delta = 0.1$

- is a jump component at time t with distribution

$$log(1 + J_t) \approx N(log(1 + \mu_J) - \delta^2/2, \delta^2), \tag{2.10}$$

while N_t is Poisson process with intensity $\lambda > 0$ and δ plays role of standard deviation of the jump size (see more details in [6]).

Since the Merton model doesn't allow option pricing in closed form, as it happens for most of the more advanced financial models, we consider the Euler discretization for equation 2.7 for Monte-Carlo simulations:

$$S_t = S_{t-\Delta t}(e^{(r-r_J-\sigma^2/2)\Delta t+\sigma\sqrt{\Delta t}z_t^1} + (e^{\mu_J+\delta z_t^2} - 1)y_t) \tag{2.11}$$

where z_t^1, z_t^2 are normally distributed and independent, y_t is a Poisson distributed random variable with intensity $\lambda > 0$, being assumed to be independent of z_t^1, z_t^2.

Chapter 3

Calibration problem

In this chapter, the financial model calibration problem will be stated mathematically and the main algorithms to solve it numerically will be described. Moreover, the ML framework will be introduced and the connections with standard methods are going to be established.

3.1 Mathematical framework

We define an option pricing model with M. In this work we consider the: M^{BSM}, for the Black-Scholes model and M^{JD} for the Merton jump model with respect to the underlying SDEs (see equations 2.5 and 2.8). The theoretical price of the linked options will be denoted by (see more details [16], [17])

$$Q(\tau) = M^{model}(\theta; \tau, \phi), \tag{3.1}$$

where θ is the vector of model parameters. In particular, for M^{BSM} we will have a single parameter $\theta^{BSM} = \{\sigma\}$ while for M^{JD} model, $\theta^{JD} = \{\sigma, \lambda, \mu_J, \delta\}$. Variable τ is a vector, describing properties of the particular instrument, e.g. maturity date, open interest etc and ϕ stands for exogenous variables, which are not directly specified by the pricing model. In particular, they can obtained from other sources like interest rate curves or news sentiment.

Model calibration means adjusting these parameters in such a way, that theoretical $Q(\tau)$ describes market prices Q^{market} as close as possible with

respect to the given metric. Usually, Q^{market} stands for a set of quotes for some period of time $\{Q^{market}\}_i, i = 1...N$, where N stands for a number of quotes for this period. The concept of "closeness" is described by a loss function, that usually is in a form of a metric defined within a suitable space. This loss function sets the objective of our optimization problem and must be minimized to find the optimal, with respect to specific criteria, for the parameters choice. In financial context, we are interested in $Q(\tau)$ from a financial model to be as close as possible to $Q^{market}(\tau)$, otherwise we will be in risk of overpricing options with low value and underpricing potentially profitable options.

Since all the parameters lie in a subset of R^m space (with constraints related to non-negativeness of some parameters like the volatility σ), good choices for the loss function are the mean absolute error:

$$L_1(\theta^*, Q^{market}; \tau, \phi) = \sum_{n-1}^{N} |Q(\tau_i) - Q^{market}(\tau_i)|, \tag{3.2}$$

where τ_i is a properties vector of the ith out of N instruments on the market, $Q(\tau_i)$ is the value of the ith instrument based on a financial model and $Q^{market}(\tau_i)$ is the market valuation of the instrument; the mean squared error is defined the following way:

$$L_2(\theta^*, Q^{market}; \tau, \phi) = \sum_{n-1}^{N} (Q(\tau_i) - Q^{market}(\tau_i))^2, \tag{3.3}$$

while the Huber loss is defined by:

$$L_\delta(\theta^*, Q^{market}; \tau, \phi) = \begin{cases} 0.5 \times L_1(\theta^*, Q^{market}; \tau, \phi), & L_1(\theta^*, Q^{market}; \tau, \phi) \leq \delta \\ \delta \times L_2(\theta^*, Q^{market}; \tau, \phi), & otherwise, \end{cases} \tag{3.4}$$

where δ is a pre-defined theshold value (see [11] for details). The calibration problem now is the following optimization problem:

$$\theta = \arg\min_{\theta^*} L(\theta^*, Q^{market}; \tau, \phi) = \Theta(Q^{market}, \tau, \phi) \tag{3.5}$$

Herefore by a function, that takes market quotes Q^{market}, their properties τ and exogenous variables ϕ as input, and returns optimal parameters θ^*

values of the financial model with respect to pre-defined loss function, where the function

$$\Theta(Q^{market}, \tau, \phi) : R^M \to R^m. \tag{3.6}$$

describes the model calibration on task of our interest.

The function in equation 3.5 can be computed in several ways, see section 3.2). In particular we will accurately and efficiently show, that ML algorithms are able to approximate it, i.e. given the same inputs, ML model and function $\Theta(Q^{market}, \tau, \phi)$ return outputs close to real data with respect to predefined cost function.

3.2 Optimization algorithms

The optimization problem defined in 3.5 is rather challenging due to several reasons:

- non-convexity: there can and will be multiple local minimas in the defined cost function;
- ill-poseness: result of calibration can vary for each optimization run, because of Monte-Carlo simulations in pricing or stochastic nature of an optimization algorithm itself;
- non-differentiability: derivatives of different order n of $\frac{\partial^n L(\theta^*, Q^{market}; \tau, \phi)}{\partial \theta^n}$ are not always available in a closed form, because of Monte-Carlo simulations in pricing or stochastic optimization as well (see more details on the problems related to stochastic optimization of non-differentiable functionals in [18]).

In practice, to avoid the issues stated above, complicated heurictical algorithms are applied and, often, they're being re-running several times to try to achieve the optimal value among different local minimas obtained as results of different runs. This is the main motivation behind this work - to try to develop approaches that eliminate this need of complex repeating and time consuming calculations and replacing them with a single, well-posed, fast and robust approxination using ML-based models.

3.2.1 Gradient based optimization

If the optimization problem is convex and the mathematical model behind can be differentiated (e.g., to calculate the implied volatility within the Black-Scholes model, see [8]), some "standard" methods can be successfully applied. Most of them are gradient based optimization methods, like gradient descent, that sequentially update optimized parameters with the following rule:

$$\theta_{n+1} = \theta_n - \lambda \nabla L(\theta_n) \tag{3.7}$$

where θ_n is a vector of parameters to be optimized on iteration n, ∇ is the gradient of a function L that is being optimized with respect to the parameters θ and λ controls speed of optimization - with this step the coordinates of optimal parameters will change in the search space (as it is in equation 3.8).

If the second order partial derivatives of function L with respect to the parameters θ are available, second-order gradient methods, like Newton algorithm can be applied with the following parameters update rule:

$$\theta_{n+1} = \theta_n - \lambda H[L(\theta_n)]^{-1} \nabla L(\theta_n), \tag{3.8}$$

Unfortunately, these algorithms can't be applied in practice, because of potential non-convexity, ill-poseness and non-differentiability of the function L.

3.2.2 Brute force search

Since gradient based approaches are not always available, there is a need for heuristical algorithms to find the optimal value of a complicated function. The simplest way to do this is so called "brute force" or "random" search. The latter method basically picks random values of a cost function and returns optimal value with the following algorithm:

- randomly generate a first parameters set values, consider it as a minimum,
- iterate over $i = 1...I$, where I - maximum number of iterations, fixed before the optimization as a hyperparameter

sample a new set of parameters values uniformly from $\{min, max\}_j$, where j - number of parameters of a model and min and max are minimum and maximum allowed values for given parameters

if a cost function with respect to the new sample has smaller value than previous one - consider it as a new minimum,

- return minimum of a function.

Concerning theoretical applicability of such an approach, Matyas [20] showed that purely random sampling of the search-space will inevitably get a value arbitrarily close to the optimum. Of course, this method is extremely time consuming, because it guarantees convergence to the optimal parameters while $I \to \infty$. There are several heuristical improvements over random search, one of the most successful being the evolutionary algorithms, that mimic the biological process of evolution in some way to create the most effective species (e.g. candidates for optimal parameters θ in equation 3.5, see [19] for further details). In this work we have chosen the differential evolution (see [21]).

3.2.3 Differential evolution

The differential evolution is a heuristical optimization algorithm used for multidimensional real-valued functions that does not use the derivatives of the cost function which means that it does not require the optimization problem to be differentiable, as it's needed by gradient-based methods described in 3.2.1. What's imporant for our problem, differential evolution can also be used to optimize functions that are not continuous, ill-posed, noisy (for example, due to Monte Carlo simulations, that approximate a function with the use of random variables, hence, introducing noise).

As an evolution-inspired algorithm (see [19]), differential evolution optimizes a cost function by creating new candidates for a local optima by combining existing ones according to heuristical set of rules and keeping the candidates that have the smaller loss function. This method is very similar to the brute-force search: it is also based on random search over the space of parameters of a cost function, but with several improvements, related to the combining intermediate solution candidates instead of interative replacing, described in the following algorithm:

- define cost function L with respect to its parameters θ_i, $i = [1...m]$, where m is a dimension of the vector θ;
- define hyperparameters F - differential weight, that factors difference between solution candidates (see next steps), CR - crossover probability, which is probability of combining solution candidates, and NP - population size - initial number of solution candidates, that will be evolved with time;
- let θ be a set of NP candidates for optimal parameters, initialize them all randomly;
- for $i = 1...N$, where N - number of iterations:

  for each candidate θ_j in set of candidates:

  pick three (see [21] for the motivation of choice) candidates θ_a, θ_b, θ_c ;

  pick a random index R from dimensionality m of a cost function L;

  compute candidate's θ_i new position as follows: draw random numbers r_i for each dimension and if $r_i < CR$ or $r_i = R$: set position of θ_j as $\theta_{ai} + F * (\theta_{bi} - \theta_{ci})$;

  if the cost function L with respect to the θ_j improved, move to a new position, otherwise stay on the old one;
- at the end, pick the candidate θ with the best value of a cost function and return it as an approximate optimal solution.

In this work differential evolution is used as reference optimization algorithm for the problem 3.5, e.g. we will try to approximate results of differential evolution optimization process with ML models.

3.3 Machine learning

As we have stated in 3.2, the optimization problem we're trying to solve is particularly challenging both from an analytical and computational point of view. Even with the use of all the heuristics described in subsections 3.2.2 and 3.2.3, are needed a lot of time and computational power to calibrate a

financial model to real situation. It will be shown in the Chapter 7, that differential evolution applied to Merton jump model calibration, can take 150 seconds for a single instrument a single date, single maturity. On the other hand, in reality, an inverstor has to deal with a structured portfolio of multiple financial instruments that have to be traded on different dates and within short, if not "high frequency" based, times. What we're looking for, is for an algorithm, that is able to replicate before defined (probably noisy, ill-posed, non-convex, or non-differentiable) optimization function (see equation 3.6), represented by one of the algorithms in the Section 3.2 with an approximation, that is much faster, well-posed and differentiable.

ML algorithms allow to approximate, interpolate and extrapolate high-dimensional data. This is why they have been widely used in different areas (see application examples in [22] for physics, [23] for biology, [24] for social sciences and [36] for finance). Their concrete implementation and optimization strongly relies on the appropriate choice of relevant parameters. Given a dataset that consists of historical inputs and outputs for the model (namely, from events that happened in the past) we can train a ML model to minimize the difference between the actual output that corresponds to the given input and what this particular ML model outputs for the future observations (also called out-of-sample data). In our case, we are interested in such a model, that takes variables Q^{market}, τ, ϕ exploiting the real market data and outputs parameters $\{\theta_i\}$ of financial model that are being produced by some known optimizer (e.g. like brute force or differential evolution in our case). Hence, we can define a learning problem in terms of finding a function, namely a machine learning model, that approximates

$$\{\theta_i\} = \Theta(Q^{market}, \tau, \phi) \tag{3.9}$$

on the basis of historical values.

Therefore, the main difficulty here is not just to find such a function, that interpolates Θ on the historical data (interpolation), but such one, that will work the same for the future values, that will differ from the ones in the training dataset (extrapolation). In practice, following algorithms are used to prevent it:

- regularization: setting constraints on the learned parameters of ML model, forcing them to be smaller, and, hence prevents learning a model

that simply remembers the training data, because it has more parameters than those justified by the observations

- dropout: randomly setting some parameters of ML model to zero while training to prevent their "co-adaptation", namely an effect of learning the same pattern by two different parameters
- data augmentation: adding more, typically artifically generated data, to the training dataset
- early stopping: breaking learning process if the cost function doesn't improve on out-of-sample data for some time

In this work, all before mentioned approached will be used, see Section 5 for more details.

Coming back to the quest for the function, mentioned before, the associated definition can be mathematically seen as an auxiliary optimization problem, where we need to minimize a difference between actual and predicted financial model parameters, with respect to the ones of the ML model. For the ease of reading, we will call parameters of financial model as "parameters" θ and parameters of machine learning model as "weights" W, therefore we have:

$$W = \arg\min_{W^*} L_{ML}(W^*; Q^{market}, \tau, \phi; \theta), \tag{3.10}$$

where L_{ML} is a cost function that defines the difference between actual and predicted parameters θ based on the inputs Q^{market}, τ, ϕ (market prices of an instrument its properties and exogenous variables respectively).

In the present thesis we are going to develop a ML approach, which is built to be differentiable, so that gradient-based approaches approaches are successfully applied to optimize problem 3.10 to find optimal W^*. Following gradient decent and equation 3.7 we obtain following update rule for solving problem 3.10:

$$W_{n+1} = W_n - \lambda \nabla L_{ML}(W_n) \tag{3.11}$$

The general scenario for approximating a function $\Theta(Q^{market}, \tau, \phi)$ from equation 3.6, which will be described in details in Sections 5 and 7 for details is the following:

1. select a financial model M, for exaple, the Merton jump model with its set of parameters θ;
2. select a reference optimization algorithm, for example, differential evolution;
3. define the financial instrument related parameters vector τ and exogenous variables ϕ;
4. solve calibration problem, stated in 3.5, using reference optimization algorithm, and save obtained parameters as θ;
5. solve optimization problem 3.10 using Q^{market}, τ, ϕ and obtained value of θ in step 4, and save ML model with its weights W^*;
6. use the ML model with weights W^* for out-of sample calibrations.

Chapter 4

Artifical neural networks

Among various ML models like support vector machines, decision trees, non-parametic approaches (for more details on other ML models see [25]) we are concentrating on artificial neural networks (ANNs, see [26]) in this work. ANNs are differentiable functions, which will ease sensibility analysis of calibration model. Also, they're showing state of the art results in various applied fields (see [28], [29]) outperforming another ML approaches. In this chapter we will review main neural network types that are used in this work from mathematical point of view, how they are being trained alongside with particular architectures we have used in this work.

4.1 Multilayer perceptron

Here we will describe a general multilayer NN that consists of L layers. Each arbitrary layer, say ℓ, has N_ℓ neurons x_1^ℓ, x_2^ℓ, ..., $x_{N_\ell}^\ell$, each with a transfer function σ^ℓ. These transfer (or activation) functions may be different from layer to layer. As the Delta rule (also, see [27]) states, the activation function can be given by any differentiable function, but does not need to be linear. These neurons receive signals from the neurons in the preceding layer, $\ell - 1$. For instance, a neuron x_j^ℓ receives a signal from $x_i^{\ell-1}$ multiplied by a weight $w_{ij}^{\ell-1}$. Hence, we have an $N_{\ell-1}$ by N_ℓ weight matrix, $W^{\ell-1}$, whose elements are given by $W_{ij}^{\ell-1}$, for $i = 1, 2, \ldots, N_{\ell-1}$ and $j = 1, 2, \ldots, N_\ell$. Neuron x_j^ℓ also has a bias given by b_j^ℓ, that takes part in linear transformation from layer to layer.

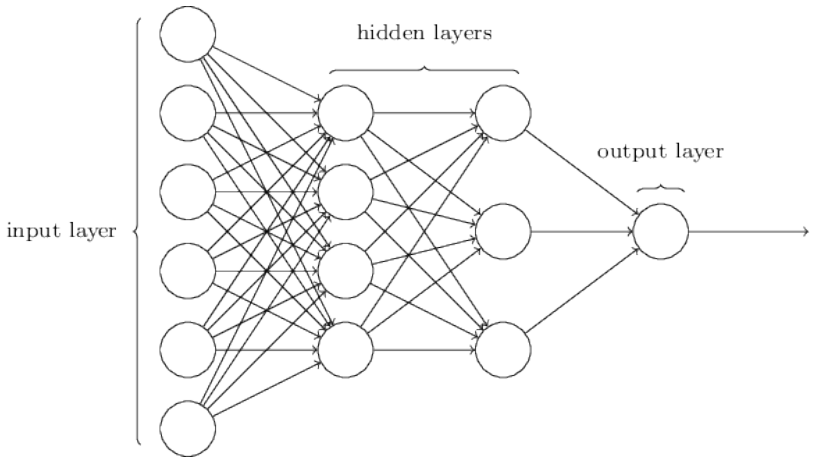


Figure 4.1: Example of a MLP architecture

For the sake of notation, we will define output of the very last layer as y_L, the total input to the unit u_i^ℓ in the layer ℓ as x_i^ℓ and the output of the unit u_i^ℓ as y_i^ℓ. In order to compute the output of a NN based on its input, the following forward propagation procedure is applied:

1. compute activations for layers with known inputs:
   $y_i^\ell = \sigma(x_i^\ell) + b_i^\ell$
2. compute inputs for the next layer from these activations:
   $x_i^\ell = \sum_j w_{ji}^{\ell-1} y_j^{\ell-1}$
3. repeat steps 1 and 2 until the last layer and return values y^L

Now we need to define the way to adjust weights W_{ij}^ℓ of this NN. Since the whole structure is differentiable, because it consists of combinations of linear transformations followed by non-linear activation functions, that are differentiable, we can optimize this function with gradient based approaches (see section 3.2.1). We need to define a loss function $E(y^L)$ and calculate partial derivatives $\frac{dE(y^L)}{dy_i^L}$, depending on the derivatives of the weights in the previous layers. This scheme leads us to the use of a backpropagation algorithm to calculate these derivatives, that is briefly described now (see more details in [27])

In particular, by the chain rule we get partial derivatives with respect to the weights in any layer:

$$\frac{\partial E}{\partial w_{ij}^{\ell}} = \frac{\partial E}{\partial x_j^{\ell+1}} \frac{\partial x_j^{\ell+1}}{\partial w_{ij}^{\ell}} = y_i^{\ell} \frac{\partial E}{\partial x_j^{\ell+1}} \tag{4.1}$$

The partial derivative with respect to the input x_j is calculated as

$$\frac{\partial E}{\partial x_j^{\ell}} = \frac{\partial E}{\partial y_j^{\ell}} \frac{\partial y_j^{\ell}}{\partial x_j^{\ell}} = \frac{\partial E}{\partial y_j^{\ell}} \frac{\partial}{\partial x_j^{\ell}} \left(\sigma(x_j^{\ell}) + b_j^{\ell}\right) = \frac{\partial E}{\partial y_j^{\ell}} \sigma'(x_j^{\ell}). \tag{4.2}$$

While, partial derivatives with respect to the outputs are calculated as

$$\frac{\partial E}{\partial y_i^{\ell}} = \sum \frac{\partial E}{\partial x_j^{\ell+1}} \frac{\partial x_j^{\ell+1}}{\partial y_i^{\ell}} = \sum \frac{\partial E}{\partial x_j^{\ell+1}} w_{ij}, \tag{4.3}$$

and concerning the last output, we also need the derivative of the error function with respect to it, so that:

$$\frac{\partial E}{\partial y_i^{L}} = \frac{d}{dy_i^{L}} E(y^L). \tag{4.4}$$

Exploiting previous results, the complete backpropagation algorithm for a multilayer perceptron can be written as follows:

1. compute errors on the output layer L:
   $\frac{\partial E}{\partial y_i^L} = \frac{d}{dy_i^L} E(y^L)$;
2. compute partial derivative of error with respect to neuron input at first layer ℓ with known error:
   $\frac{\partial E}{\partial x_j^{\ell}} = \sigma'(x_j^{\ell}) \frac{\partial E}{\partial y_j^{\ell}}$;
3. compute errors on the previous layer:
   $\frac{\partial E}{\partial y_i^{\ell}} = \sum w_{ij}^{\ell} \frac{\partial E}{\partial x_j^{\ell+1}}$;
4. repeat steps 2 and 3 for all layers $1...L-1$ until all the errors corresponding to all the weights in the NN are calculated;
5. compute the gradient of the loss function:
   $\frac{\partial E}{\partial w_{ij}^{\ell}} = y_i^{\ell} \frac{\partial E}{\partial x_j^{\ell+1}}$.

Summing up steps 1 to 5, we can train a NN with any gradient based algorithm as in the case of gradient descent and its variations (see section 3.2.1 and [32]).

4.2 Convolutional neural network

In this subsection, we are going to introduce Convolutional Neural Networks (CNNs) architectures (see figure 4.2 and [28] for details) as the first alternative to MLP architecture. The main idea behind CNNs is replacing a single weight matrix W^ℓ with a set of convolution kernels $\{\omega\}$. It allows to learn not a single map from layer to layer, but a set of different maps based on convolution, that "finds" local patterns in the activation of the previous layer. This idea, particularly when applied to computer vision (see [33]) started a so called "neural network renaissance" and has several successful applications in signal processing [35], time series analysis [36], natural language processing [34] and other areas.

Suppose that we have some $N \times N$ square neuron layer (like a table with option chain) which is followed by a convolutional layer. If we use an $m \times m$ filter ω, this convolutional layer output will be of size $(N-m+1)\times(N-m+1)$. In order to compute the pre-nonlinearity input (what corresponds to linear map in MLP, e.g. $\sum_j w_{ji}^{\ell-1} y_j^{\ell-1}$) to some unit x_{ij}^ℓ in our layer, we need to compute the following:

$$x_{ij}^\ell = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \omega_{ab} y_{(i+a)(j+b)}^{\ell-1}. \tag{4.5}$$

Then, nonlinear activation function (see Section 4.1 for details) for the result of convolution operation in equation 4.5 is applied:

$$y_{ij}^\ell = \sigma(x_{ij}^\ell). \tag{4.6}$$

Sometimes, to reduce the size of previous layer activation, so called max pooling layer is used. In particular, we can take some $k \times k$ region and output a single value that basically is the maximum value in that region. The size of the region is usually chosen depending on the typical pattern size in the input (see [33] for further information). It helps to reduce dimension of the

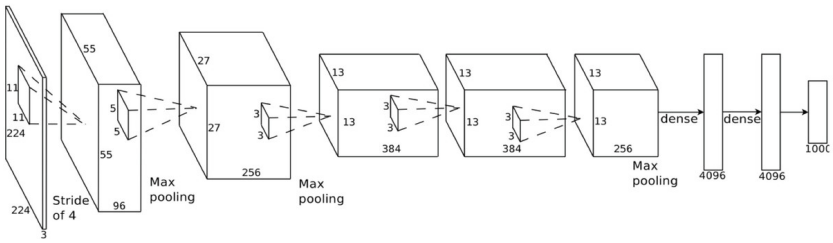


Figure 4.2: Example of a CNN architecture

problem, speed up the training and inference processes, and removes redundant elements of the input that don't contribute into learning. Since max pooling was first introduced in computer vision field, it was designed to select just pixels with high values of activation function in them as candidates for having a visual object in them.

The learning algorithm for CNNsis the same backpropagation algorithm as for MLP. Only the partial derivatives for gradient based optimization needs to be recalculated. Therefore, partial derivatives with respect to the kernels ω_{ab} are:

$$\frac{\partial E}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^{\ell}} \frac{\partial x_{ij}^{\ell}}{\partial \omega_{ab}} = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E}{\partial x_{ij}^{\ell}} y_{(i+a)(j+b)}^{\ell-1}, \tag{4.7}$$

while partial derivatives with respect to the inputs are:

$$\frac{\partial E}{\partial x_{ij}^{\ell}} = \frac{\partial E}{\partial y_{ij}^{\ell}} \frac{\partial y_{ij}^{\ell}}{\partial x_{ij}^{\ell}} = \frac{\partial E}{\partial y_{ij}^{\ell}} \frac{\partial}{\partial x_{ij}^{\ell}} \left(\sigma(x_{ij}^{\ell})\right) = \frac{\partial E}{\partial y_{ij}^{\ell}} \sigma'(x_{ij}^{\ell}), \tag{4.8}$$

and to the outputs derivatives are:

$$\frac{\partial E}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^{\ell}} \frac{\partial x_{(i-a)(j-b)}^{\ell}}{\partial y_{ij}^{\ell-1}} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} \frac{\partial E}{\partial x_{(i-a)(j-b)}^{\ell}} \omega_{ab} \tag{4.9}$$

Concerning max pooling layers, since they don't actually contribute to learning the weights, the error is just backpropagated to the place where it came from in original dimension.

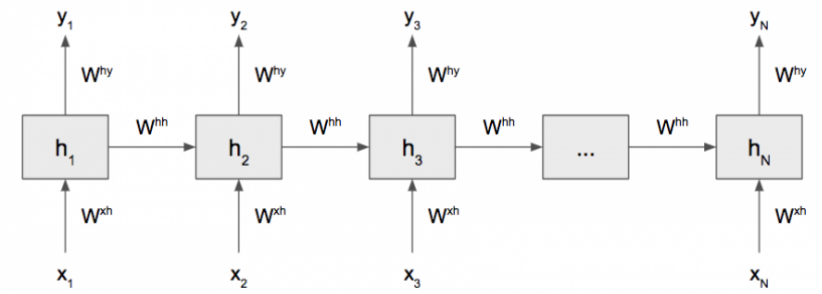


Figure 4.3: Example of a RNN architecture

4.3 Recurrent neural network

Recurrent neural networks (RNNs, see [29]) work with sequences of data. They have inputs x_i, that can be multidemensional, outputs y_i and hidden states h_i. At a single timestep, the RNN works the same a simple MLP neural network with one hidden layer (see section 4.2). But in case of RNNs there are actually two inputs: one is x_t on time t, second is a hidden state from previous timestep $t-1$. Hence, we have three separate matrices of weights: input-to-hidden weights W_{hx}, hidden-to-hidden weights W_{hh} and hidden-to-output W_{yh}. Therefore, we jave the following forward equations for RNN:

$$\begin{aligned} h_i &= \sigma(W_{hh}h_{i-1} + W_{hx}x_i + b_h) \\ \hat{y}_i &= W_{yh}h_i \end{aligned} \tag{4.10}$$

To train a recurrent neural network we have to generalize backpropagation algorithm from MLPs or CNNs. There are alternative approaches, namely "teacher forcing" and others (see [30] for further details), but they are out of the scope of this thesis. Since this backpropagation algorithm works on sequences in time, it is known as backpropagation through time (BPTT, see [29]). For every input sequence of the length k, the RNN is being unrolled into a regular MLP that has k hidden layers, but differently to MLPs, it has $k+1$ different inputs (the actual one plus k hidden states). The BPTT algorithm is the following:

1. initialize weight matrices W_{hx}, W_{hh}, W_{yh} randomly (see [29] for the choices of distributions);
2. repeat for $i = 1...N$ iterations:

unroll a RNN for k time steps (for a sequence of length k);

set the inputs in the unrolled RNN: zeros for hidden states and actual inputs x_i;

perform forward and backward propagation as in regular MLP;

average the gradients for each layer to update the matrices equally for each step.

RNNs are powerul tools for modeling sequences of data, because, oppositely to the MLPs or CNNs they model directly dependece between consequent elements of the sequence, while MLP treats a sequence as a single vector and CNN models just local patterns that depend on the convolution size. Sequences are regular data structures in finance (like asset prices time series), see, in particular, subsection 4.3.3 for a case study exploiting the RNN approach.

4.4 Neural networks architectures

Neural networks types, introduced in Sections 4.1, 4.2, 4.3 need to be defined with particular architectures, e.g. for MLP we need to adjust numbers of hidden layers, neurons in each hidden layer, activation functions etc. For CNNs, additionally, we need to define kernel sizes. Analogously, for RNNs the size of hidden state vector has to be chosen. In practice, above-mentioned degrees of freedom for building NN architectures are chosen by the experts, e.g. researchers or developers based on their experience. In this work, for all types of NNs we have chosen number of layers, neurons and other degrees of freedom, by running experiments on small subset of the data, checking performance and picking a setting with the best performance using grid search approach (see [38] for further details). In the following three subsections we will define architectures of MLP, CNN and RNN for option calibration problem, stated in equation 3.10, and motivation behind the choices.

4.4.1 Multilayer perceptron

Since multilayer perceptrons are designed to work with vector data, they will be applied to the vectors from option chains that don't have spatial structure. These vectors are obtained by "flattening", i.e. collapsing an arbitrary

matrix (or, in general, tensor) into a vector that contains all the elements of the initial object. In case of option chains (see more details about option chains structure in Chapter 5) the input dimension is relatively low (5 variables for each option and general amount of options per trading date varies from 10 to 60), so the number of neurons in each layer shouldn't be very high either. This is motivated by the idea, that in hidden layer we might want to reduce the initial hidden space to the lower dimension to eliminate redundant features (like the options, that are not being actually traded). Latest research shows (see [37] for details), that in terms of "wideness", i.e. number of neurons in each layer, and "depth", i.e. number of hidden layers of neural network architectures the latter is more crucial since more layer means higher hierarchy of the features that will be learned, we will stick to this idea as well.

The neural network we build has 5 layers, 64 neurons each with the rectified linear unit (ReLU) [39] as the activation function. The 5-layer neural network is considered to be "deep", so we need to take into account, that some problems like vanishing or exploding gradient can occur. We apply a very recent technique called "residual connections" (see [40]) to help training such deep architecture. They are used to allow gradients to flow through a network directly, without passing through non-linear activation functions. As a preliminary step, before all activations batch normalization layer (see [41] for details) is used. Dropout (see [42]) with rate 0.25 was used as main regularization method after each layer. For additional regularization each input in training phase was augmented with Gaussian noise with standard deviation of value 0.05. The grid search to find above-mentioned parameters has been performed in the following space: number of hidden layers from the set of values $\{1, 3, 5, 7, 10\}$, number of neurons in each from the set $\{16, 32, 64, 128, 256\}$, dropout rate from the set $\{0.1, 0.25, 0.5, 0.75\}$ and Gaussian noise standard deviation from the set $\{0.01, 0.05, 0.1\}$.

The output of the neural network depends on how many parameters from the financial model we want to predict. The only thing that will stay constant is the fact that there will be no activation function because the value range for different parameters can be in subset of R^m. For some exceptions, as volatility, ReLU activation function is used, that doesn't allow values less than zero. The visualization of the MLP architecture is represented on the figure 4.4 and source code for it is in appendix C, listings C.1, C.2.

4.4.2 Convolutional neural network

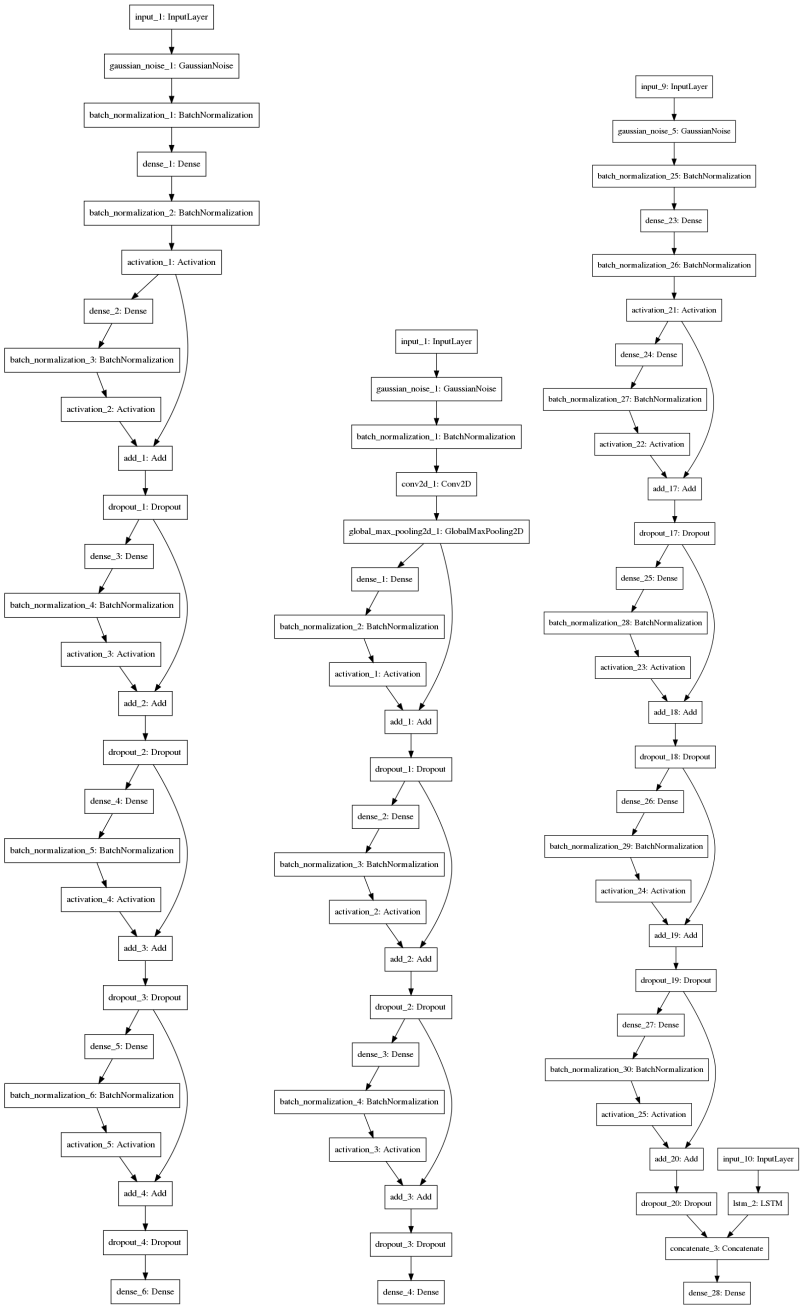
Another representation about the input data a matrix of last option prices and additional variables like bid price, ask price or others, that are the columns and different strike prices as the rows. This table has spatial structure, and, hence, spatial patterns. CNNs are designed to learn patterns in spatial data with arbitrary kernels. To train the CNN the following set of parameters needs to be defined: number of layers, number of filters in each layer, size of each filter in each layer (usually set of filters in a single layer has the same size), activation function and the method to "flatten" (see 4.4.1) obtained activation maps into a single vector for regressing the final layer on it.

In our case, just a single convolutional layer was chosen, because we don't expect rich hierarchy in table representing option chain. Over this convolutional layer, 4 fully-connected layers like in MLP are applied. This convolutional layer has to learn 64 kernels, each of size 3×3 and after obtained activation maps are flattened into a vector with global max pooling (see [31] for details). The latter takes maximum from each of the activation maps which allows us to get a 64-dimensional vector as a result of a convolutional layer. The activation function has chosen to be the same - ReLU and output layer is designed as in the MLP case. The grid search to find the parameters has been performed in the following space: number of kernels from the set of values $\{16, 32, 64, 128\}$ and kernel size from the set $\{2 \times 2, 3 \times 3\}$. The visualization of the CNN architecture is represented in figure 4.4 and corresponding source code for it is in appendix C, listing C.3.

4.4.3 Recurrent neural network

Another important assumption in Finance is that historical events somehow influence future ones (see more details in [43]). In our case we can assume, that calibrated parameters from the past N days can help to determine today's parameters values better than if we just used data from a current day. This assumption cannot be modeled directly with MLPs or with CNNs (but can be modeled as a flattened vector of past values). The idea behind using RNN to extract information from previous days alongside with another architecture that works solely on the current day is represented in the figure 4.4 and corresponding source code for it is in appendix C, listing C.4. Let's

call the NN for modeling current day as a "primary NN" and the RNN that models previous days as "auxiliary NN". To the activations from the very last hidden layer of a primary NN we append last vector $\hat{y}_i$, that is obtained from the auxiliary NN that has inputs $\{x_i\}$, where each x_i is a vector of calibrated parameters θ from 7 previous trading days. On this new vector, that is obtained after concatentation, the last layer is regressed. The size of the hidden state vector was fixed to be 64 to correspond the dimension used for MLP and CNN.



(a) MLP (b) With CNN (c) With RNN

Figure 4.4: Neural network architectures graphs

Chapter 5

Dataset description

This chapter explains how the real raw market data we have considered to test our methods has been appropriately preprocessed and then turned into training dataset.

5.1 Option chains

The main data structure used in this work is an option chain. The option chain is a matrix for an underlying asset storing all puts, calls, strike prices, and option prices for a given maturity period. For this study a historical dataset of option chains of Apple Inc. (AAPL ticker on the stock market) options was used from 2009 to 2011 because of availability of this particular dataset. Option chain contains European call and put options, their high, low and closing prices on the market, maturity dates and open interest (the total number of open options that are on the market at that time) for AAPL options. This data is usually stored in tables, in .csv format. An example of such kind of a table is on the figure 5.1.

We assume, that the main factors that can help to estimate the model parameters are:

- time to maturity (T),
- difference between underlying price and strike price (ΔP),
- last, bid and ask prices for the particular option (C, B, A),

	Underlying	Underlying Price	Expiry	Type	Strike	Last	Bid	Ask	Volume	Open Interest
2	A	15.63	20090117	C	10	5.62	5.6	5.7	0	5
3	A	15.63	20090117	P	10	0.05	0	0.01	0	65
4	A	15.63	20090117	C	12.5	2.61	3.15	3.2	0	6
5	A	15.63	20090117	P	12.5	0.11	0.02	0.04	0	371
6	A	15.63	20090117	C	15	1.01	1	1.01	0	303
7	A	15.63	20090117	P	15	0.33	0.36	0.38	0	1563
8	A	15.63	20090117	C	17.5	0.04	0.06	0.08	0	2253
9	A	15.63	20090117	P	17.5	1.85	1.9	1.95	0	1451
10	A	15.63	20090117	C	20	0.02	0	0.01	0	1239
11	A	15.63	20090117	P	20	4.2	4.3	4.4	0	319
12	A	15.63	20090117	C	20	0.59	0.01	0.1	0	211
13	A	15.63	20090117	P	20	2.6	3.1	3.35	0	556
14	A	15.63	20090117	C	22.5	0.02	0	0.01	0	579
15	A	15.63	20090117	P	22.5	6.85	6.8	6.9	0	301
16	A	15.63	20090117	C	25	0.03	0	0.01	0	243
17	A	15.63	20090117	P	25	9.5	9.3	9.4	0	750
18	A	15.63	20090117	C	25	0.2	0	0.05	0	204
19	A	15.63	20090117	P	25	6.55	8.05	8.3	0	378
20	A	15.63	20090117	C	27.5	0.02	0	0.01	0	638

Figure 5.1: Example of a table with an option chain

- open interest for the particular option (O).

For the sake of simplicity, we calibrate the financial models to price just the call options. Moreover, since geometric Brownian motion and jump diffusion are relatively primitive financial models, it's known empirically (see [44]), that they're not able to model long-term maturities very well. Thus, we are going to price only call options for the closest maturity date to avoid very high errors in calibration.

5.2 Preprocessing

Naturally, the table with variables we have selected above, is just a subtable of the initial table with option chain. We can turn it into a training example in several ways. One of the hypotheses can be that spatial position of the prices in the table plays its role and that such a representation needed that doesn't change the initial structure. Hence, each training example will be stored as a matrix $M^{m \times n}$, where m is number of variables and n is number of options traded in current day. This data representation is natural for convolutional neural networks. Another hypothesis is that this structure is neglible, and the table of features can be simple flattened into a vector - this input can be used for regular multilayer perceptrons. The performance of above-mentioned representations is represented in Chapter 7, tables 7.1 and 7.2.

	Last	Bid	Ask	PriceStrikeDiff	OpenInterest
0	57.40	55.35	56.75	-55.35	351
1	52.00	50.35	52.40	-50.35	2640
2	47.75	45.35	46.10	-45.35	2362
3	45.85	40.30	42.05	-40.35	1057
4	37.30	35.40	36.55	-35.35	2493
5	31.20	30.45	30.65	-30.35	756
6	27.45	25.50	25.70	-25.35	2388
7	23.00	20.60	20.85	-20.35	1758
8	16.35	15.90	16.10	-15.35	3754
9	11.92	11.45	11.65	-10.35	2467
10	7.75	7.60	7.70	-5.35	10643

Figure 5.2: Example of a part of the option chain variables

Of course, before-mentioned input vectors or matrices have to be normalized before being processed by ML algorithms. We decided to use min-max normalization, that "squeezes" values to be in predefined range. We have chosen a range $[0, 1]$ with following formula:

$$z_i = \frac{x_i - \min(\{x_i\})}{\max(\{x_i\}) - \min(\{x_i\})}, \tag{5.1}$$

where $\{x_i\}$ are elements of the initial input data structure and $\{z_i\}$ are corresponding normalized elements. Hence, we obtained normalized and, depending on the learning algorithm, "flattened" (see definition in Section 4.4.1) option chains of AAPL, filtered to have only call options for the closest maturity date as inputs. An example of input structure (not normalized) to a neural network can be seen in figure 5.2.

Outputs will be parameters of financial model, calibrated to the input option chains with differential evolution algorithm (see 3.2.3), i.e. optimal parameters as results of solving the problem from 3.5 for each day of a training set. The search space for parameters of Black-Scholes model and Merton jump model for differential evolution optimization is shown on the table 5.1 (minimum and maximum values).

	σ	λ	μ	δ
Black-Scholes model	(0, 5)	-	-	-
Merton jump model	(0, 5)	(0.1, 5)	(-3, 3)	(0, 3)

Table 5.1: Search space for parameters calibration

Of course, we also split the dataset into two parts: one for fitting the ML model and second - for evaluation. In particular, data from 2009 has been used to train the model, while the data from 2010-2011 was exploited to test performace of the trained ML model.

5.3 Data augmentation

The NYSE and NASDAQ, two major stock exchanges in USA, where most AAPL options are placed, trade in average about 253 days a year. It means, that for each year there is slightly more than 250 training examples available, which is not enough to correctly train the ML model. In particular, when so few data can be used, then we have a rather high probability to overfit and produce wrong calibrations on the out-of-sample data. One option is to try to gather more historical data, but this process is rather long including the necessity of calibration for each of historical date. We notice, that we already know a good approximation for the inverse of a function Θ, first introduced in formula 3.9:

$$\Theta^{-1}(\theta; \tau, \phi) \approx M^{model}(\theta; \tau, \phi) = Q^{model}. \tag{5.2}$$

It is just the normal valuation of the instruments under a given set of parameters (see [17] for further details). It means that we can generate new examples with generating random parameters θ, that, although, must be properly correlated with other parameters τ and ϕ. For example, for generating a new training sample for the Black-Scholes model calibration, we can draw a random value of volatility σ as a property of the underlying and take time-to-maturity T, strike price K (see Section 2.2 for details) directly from the market and price the option using equation 2.4. Obtained value of the option will be an approximarion of the actual market price based on the randomly drawn volatility σ of the underlying asset. Concerning particular

	Bid price B	Ask price A	Open interest O
MAE	1.82	0.78	2397.07
R^2	0.995	0.998	0.229

Table 5.2: Accuracies of variables reconstruction with multi-output linear regression

distributions for different parameters, for M^{BSM} single parameter σ has been drawn from the uniform distribution $U_\sigma \sim [0, 5]$, and for M^{JD} its parameters σ, λ, μ and δ have been drawn from $U_\sigma \sim [0, 5]$, $U_\lambda \sim [0.1, 5]$, $U_\mu \sim [-3, 3]$ and $U_\delta \sim [0, 3]$ accordingly.

However, in our case, with random generated θ new artificial examples of prices of the options can be generated, but not the ask price A, the bid price B and open interest O as another variables that take part in calibration. To reconstruct them, the multi-output linear regression (see more details in [45]) has been fit on the historical data, where bid price and open interest were regressed over the last price C, that we can easily obtain after drawing random θ, and difference between underlying price and strike price ΔP. The accuracies of reconstruction based on multi-output regression can be found in table 5.2.

Hence, the general algorithm to augment initial dataset with artificially generated samples is the following:

1. calibrate a financial model for the historical training data;
2. fit a multi-output linear regression to predict O, A, B based on C and ΔP (see Section 5.1 for details on notation);
3. for each training example:

   generate $N = 100$ random parameters θ based on the correlation matrix;

   price options of this trading day based on the randomly drawn θ;

   reconstruct other variables based on option prices;

   combine variables into an artificial training example and add to the dataset.

At the end, for each training example of 2009 trading year we generate $N = 100$ (number 100 was chosen after several experiments with numbers in range from 10 to 1000 as the minimal, that improves training process) random artificial examples, reconstruct needed variables, and, hence, obtain large enough dataset to train neural networks on.

Chapter 6

Model interpretation

This chapter briefly explains current approaches to interpret outputs of modern neural networks and shows how they can be applied to financial model calibration.

6.1 Algorithms

Most of the algorithms used to interpret behavior of neural networks aim to determine the influence of any particular weight and any particular input on the final output of the model. We are mostly interested in interpreting the input of the model, which is, in our case, the option chain and its variables (like last brice C, bid prices B and ask prices A, open interest O, see Section 5.1 for more details). Assuming that the ML model works correctly, we want to understand, what variables from the option chain influence the predicted parameters values θ at most. Interpretation algorithms can help to point on particular option and its specificities in the option chain that influenced a solution, namely, calibrated parameters, at most which can give to the analyst useful insights about the market state, e.g. find anomalies or illiquid options. We will inspect these variables visually, making a link between interpretation approaches for computer vision applications, that inspect pixels of an image, and for financal model calibration, where we inspect variables in the option chain. The picture 6.1 explains what these algorithms do in computer vision, where upper image is an original picture, and the one below is a saliency map of the original one (see Section 6.1.1). Intuitively, they highlight regions of interest that most probably contain the objects. Advanced reader can notice,

Figure 6.1: Example of saliency maps in computer vision

that algorithms in sections 6.1.1 and 6.1.2 resemble approaches for sensitivity analysis of mathematical models (see [46] for more comprehensive review).

6.1.1 Saliency maps

In computer vision, a saliency map is an image that shows each pixel quality in terms of containing objects of interest in it (see more in [47]). The goal of a saliency map is to simplify the representation of an image into something that is more meaningful and easier to analyze. Let's start with a simple linear model

$$S(I) = \omega^T I + b \tag{6.1}$$

where I is an input, w, b are parameters of a linear model and $S(I)$ is a model prediction. In this case, it is easy to see that the magnitude of elements of the vector ω defines the importance of the corresponding vector elements of I for a prediction $S(I)$. In case of highly nonlinear functions like neural networks, we can approximate $S(I)$ with a linear function using first-order Taylor expansion, hence, obtaining the result from equation 6.1, and ω can be expressend as a derivative of $S(I)$ from equation 6.1 with respect to the input image:

$$\omega = \frac{\partial S}{\partial I} \tag{6.2}$$

We indeed can always calculate this derivative, because NNs that have been used in this study are differentiable functions with respect to their

inputs and their weights (see Chapter 4 for more details). Another interpretation of this saliency map is that the magnitude of the derivative value indicates which vector elements need to be changed the least to affect the prediction at most.

The problem with gradient based approaches for NNs interpretation is in use of such activation functions like ReLU (see [39]), that have zero gradient when they're not active:

$$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases} \quad f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases} \tag{6.3}$$

which doesn't bring useful information for interpretation. Another approach is comparing each neuron's activation to its "reference" - activation that the neuron has given "reference input" denoted as δ_n. It has to satisfy several properties, described in detail in paper [48].

6.1.2 Occlusion

This method is very often used in sensitivity analysis of classical mathematical models (see [46]). It's relatively simple: we just need to systematically peturbate elements of input vector I, replacing each with pre-defined value (it can be a noise, sampled, for example, from standard Gaussian distribution, but in this work we replace replace each element, one by one, with zero, hence, imitating the absence of every particular element, see more details in [49]). After this we analyse how prediction of the ML changes given a new peturbated input. The elements of the input vector that affect the prediction at most (in terms of difference between original prediction with initial input and prediction from peturbated input), while being replaced with zero, are assumed to be important for this particular example I.

Chapter 7

Training and evaluation

This chapter unveils details about implementation of the algorithms and numerical results of their execution, in particular, compares the time needed for calibration with differential evolution (see subsection 3.2.3) and different ML models (MLP, CNN and RNN, see Chapter 4 for more details) alongside with their corresponding accuracies. Also, it shows use cases of models interpretation and shows a scenario for retraining the ML model over time.

7.1 Implementation details

Main structures that have been used in this work and have been implemented programmatically are the following:

- market environment;
- stochastic process simulation;
- european option pricing mechanism;
- numerical optimization routines;
- neural networks;

related codes have been written using programming laguage Python, while basic mathematical routines have been implemented using NumPy (see [53]) library.

Concerning the market definition, we have considered its defining as a separate (Python class) structure embedding financial variables and constants, namely current date, underlying asset name, contract expiry date, short rate and others defined in Section 2.1. Stochastic processes that suppose to simulate behavior of underlying asset (GBM, jump diffusion process, see Section 2.2 and 2.3 for details) have been implemented as classes that store their parameters and are able to generate new random paths. The code can be found in Appendix A.

Option valuation was also implemented as a separate class, that is using market environment and stochastic process to price the option. To price an option in this simulation we: generate many prices that the asset might be at maturity, calculate option payoffs for each of those generated prices, average them, and then discount the final value (Monte-Carlo approach). This and previous implementation were taken from [10], you can find a code sample in Appendix B.

The initial calibration process on the AAPL data (described in Chapter 5) that requires implementation of differential evolution optimization to create a dataset was performed with the use of SciPy (see [54]) library and its subroutines.

Finally, MLP, CNN and RNN were implemented ad trained with a library Keras (see [55]). it provides a high-level interface for easy building neural architectures and relies on another library, Tensorflow (see [56]), as backend with mathematical and numerical primitives. The interpretation module has been developed using open sourced library DeepExplain (see [57]). Specific implementations of neural networks architectures one can find in Appendix C.

7.2 Training results

As optimization algorithm for optimizing a NN was chosen Adam (see [32]) with a learning rate $\lambda = 0.0001$ (for all MLPs, CNNs and RNNs). A relatively low value was chosen to stabilize training (see [58]). Every 10 epochs, if the loss function wasn't decreased, the learning rate was decayed with a factor 0.9 to try to find a better local minimum (also, see [58] for the justifi-

cation of the method). As a loss function mean square error (MSE, defined in Section 3.1) was used.

All models were initialized to train on a train set for 500 epochs. An epoch is a full cycle of backpropagating all the examples from the training set once each. To avoid overfitting, early stopping technique (see [55]) was applied: if loss function wasn't improving for 25 epochs, the training process was stopped and as optimal weight values set was chosen the one after the last improvement.

Visually, results are represented in figures 7.1 and 7.2. For Black-Scholes model we can clearly see, that NNs very closely replicate the behavior (and the errors) made by the differential evolution optimizer. The mean average errors (MAEs) for all the models and their training and calibration time are in table 7.1.

On the figure 7.2 we can see how neural network replicates calibration of the Merton jump model. Situation is a bit different, because in most of the dates the NN has even lower error than the baseline optimizer. It can be explained with a fact, that for differential evolution solving optimization for 4 parameters is more difficult problem, it's also can be seen from the MAE on the test set, which is even higher than with the Black-Scholes model. We can conclude, that initially the dataset for calibration Merton jump model has higher error, which was compensated by artificial data generation for the neural network, that allowed to achieve better accuracies comparing to the baseline optimizer. In future work, to avoid this situation we will use error-adjusted data for training the model (see [17] for details). The mean average errors (MAEs) for the models trained to calibrate jump model and their training and calibration time are in the table 7.2.

From the obtained results, we can conclude, that MLP, CNN and RNN architectures behave very similarly and hypotheses about spatial information in the option chain table or about using previous days are not giving significant improvement comparing to single-day MLP model. Nevertheless, with the use of neural networks we could reduce calibration time from 1.2, 150 seconds to 0.05, 0.11 seconds for Black-Scholes and Merton jump models respectively. The mean average error changed from 3.31 to 3.78 (in case of MLP) for Black-Scholes model calibration and from 6.64 to 6.81 for Merton

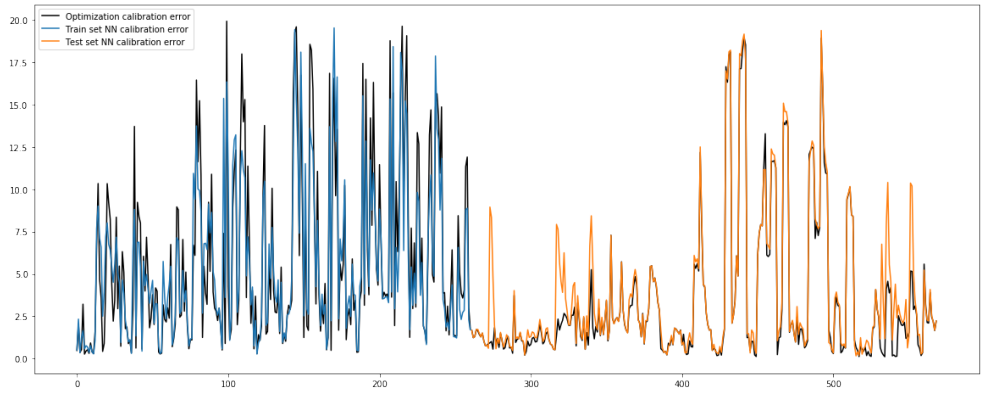


Figure 7.1: Black-Scholes model calibration for in-sample and out-of-sample data

	Differential Evolution	MLP	CNN	RNN
MAE calibration error	3.31	3.78	3.69	3.45
Training time	-	200 sec	220 sec	900 sec
Calibration time	1.2 sec	0.05 sec	0.09 sec	0.11 sec

Table 7.1: Results for Black-Scholes model calibration

jump model calibration, which can be considered as not significant.

7.3 Active learning

As it can be seen from ther figures 7.1 and 7.2 that in some particular situations that parameteres, predicted by the NN, don't allow to price the options accurate enough (at least same accurate as differential evolution would). It

	Differential Evolution	MLP	CNN	RNN
MAE calibration error	8.43	6.77	6.71	6.81
Training time	-	198 sec	202 sec	917 sec
Calibration time	150 sec	0.05 sec	0.09 sec	0.11 sec

Table 7.2: Results for jump diffusion model calibration

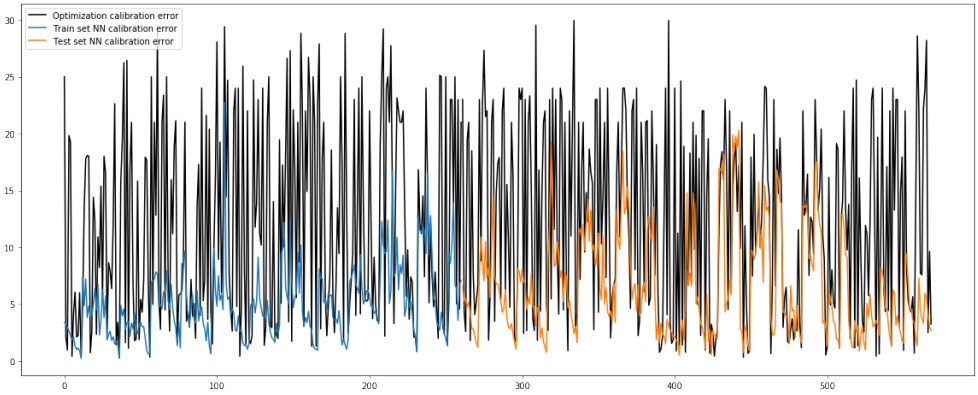


Figure 7.2: Merton jump model calibration for in-sample and out-of-sample data

can be referred to the regime switches (as it happens in case of financial crushes, regulation changes, etc) in the market or some events that didn't happen in the past, so the trained neural network, even with augmented data, can't perform correctly on this totally new sample. In general, this is normal situation, and developers of machine learning models have to collect these low-performing samples and train a model again, including them with some regularity, see [50] for more details.

In our case, we need to adapt to the market changes very quickly, and computational cost of obtaining a single training example is very high, beacause we need to calibrate it with "slow" algorithms like differential evolution, see time needed for a single calibration in tables 7.1 and 7.2. Hence, we want to minimize the amount of time and computational resources needed for retraining a model and do it as fast as possible, after a change happened.

We propose to make a calibration on out of sample data only in the situations when obtained error exceeds the maximal one from base optimizer on the validation set. It allows to minimize amount of such situations, but if one appears, we can calibrate a model for the particular day when its realized, generate 100 random parameters values (as described in Section 5.3), price options based on them, and train the neural network, already trained on the "big" training dataset, for 1 epoch. It allows slightly, for 0.02 MAE, decrease in error for future observations for the experiment with Black-Scholes model

calibration.

7.4 Results interpretation

To give human experts additional insights about how neural network can be fruitfully used to calibrate financial model and, potentially, show some new information about the market condition, we have applied occlusion method from the Section 6.1.2, to study to what particular variables from the option chain the machine learning model pays attention to. Picture 7.3 shows some examples of such interpretations. The more deep the color of a cell is - the more attention is paid to this particular variable. The columns represent last option price, bid price, ask price, difference between underlying asset price and option strike price and open interest for this particular option.

For example, in left picture, where the first 11 AAPL call options on the 06.01.2009 are represented, it can be seen, that for the upper 4 options the bid price was important, which would be totally ignored by the standard algorithms. Also, the 10th option is worth attention as the one with very high open interest comaring to the others. Interpretation methods can also give insights about when the model behaves incorrectly. For example, on the right picture on the figure 7.3, where the first 11 AAPL call options on the 19.05.2009 are shown, we can observe that almost every variable has deep green color, which means that changing any of them the output of the model will change significantly too. The latter is the case of a too sensitive calibration and can be another sign to retrain the model in an active learning mode, described in Section 7.3.

	Last	Bid	Ask	PriceStrikeDiff	OpenInterest
0	55.415713	57.28159	56.345825	-55.35	1951.315918
1	50.417407	51.932682	51.134674	-50.35	2063.460449
2	45.428999	47.024387	46.244476	-45.35	2204.676514
3	40.46087	42.906223	41.973511	-40.35	2531.736816
4	35.461483	36.265079	35.894241	-35.35	2864.928467
5	30.472064	30.855244	30.50528	-30.35	3438.651855
6	25.488356	26.196507	25.962494	-25.35	4055.310059
7	20.498782	21.381565	20.839401	-20.35	5329.794922
8	15.507426	16.322048	15.62793	-15.35	7225.553711
9	10.66186	12.10609	10.571152	-10.35	8785.532227
10	6.27362	7.977916	5.526261	-5.35	10302.166992

	Last	Bid	Ask	PriceStrikeDiff	OpenInterest
0	164.0	169.9	171.45	-170.53	69.0
1	168.8	164.2	166.4	-165.53	61.0
2	163.8	160.15	161.2	-160.53	20.0
3	158.7	155.15	156.2	-155.53	20.0
4	144.11	149.2	151.4	-150.53	30.0
5	152.0	145.15	146.6	-145.53	1.0
6	140.0	139.4	141.35	-140.53	21.0
7	126.05	135.25	136.1	-135.53	12.0
8	143.05	128.5	132.6	-130.53	17.0
9	124.55	124.5	126.25	-125.53	171.0
10	130.0	119.95	121.5	-120.53	16.0

Figure 7.3: Examples of visualization of results

Chapter 8

Summary

This works shows, how modern machine learning algorithms, in particular, deep NNs, are able to approximate solutions of optimization problems. As a practical problem, financial option pricing calibration was solved (see details in Chapter 2), as one, that is the most challenging due to the stochastic nature of the underlying data. Usually this calibration is done using complex heuristical optimization methods, described in Chapter 3, in this work as such a method differential evolution was used. Different NN types were compared, in particular, MLPs, CNNs and RNNs (more detailed information on NNs used in this work one can find in Chapter 4) and we can conclude, that even multilayer perceptron with residual connections performs reasonably well, so there is no need to use much more complex models like convolutional neural networks or recurrent neural networks. The speedup of such an approach is significant - thousands times faster exploiting neural networks with loss of accuracy in 0.3-0.5 of mean average error, as it has been shown in Chapter 7.

Moreover, the method for sensitivity analysis and model interpretation is proposed in Chapter 6, that allows to visualize, what particular options and their variables are influencing the predicted parameters at most. Last but not least, we propose a simple yet efficient scheme for scheduling retraining neural network to adapt to the market changes.

At least two paths of future research are arising. The first one is very applied: there is a room to apply currently developed framework to new financial instruments and corresponding models. Especially with respect to

interest rate and volatility models. It's also important to investigate if models trained on one set of instruments of particular class can generalize to another sets of instruments, also called transfer learning ability (see [51]). Last but not least, several improvements need to be done for the dataset creation including error-adjusted prices generation.

Another direction is more theoretical. Since models developed in this work successfully replicate behavior of complex optimization algorithms, it's interesting to investigate how this ability generalizes on another algorithms, another problems and if a model trained to approximate one optimization problem can be used to solve other related tasks. This is tightly related to meta-learning (see [52]) - a hot topic in modern optimization, machine learning and artificial intelligence research.

Chapter 9

Bibliography

[1] Maksym Artomov *Stochastic processes in biological systems : selected problems* Massachusetts Institute of Technology, http://hdl.handle.net/1721.1/57773, 2010

[2] Benth et al. *Stochastic modeling of electricity and related markets* World Sci. Publ. Co, book, 2008

[3] Black, Fischer; Myron Scholes (1973), *The Pricing of Options and Corporate Liabilities*. Journal of Political Economy. 81 (3): 637–654, 1973

[4] Oksendal, Bernt K. (2002), *Stochastic Differential Equations: An Introduction with Applications*, Springer, p. 326, ISBN 3-540-63720-6

[5] Boyle, Phelim P. *Options: A Monte Carlo approach*. Journal of Financial Economics, Volume (Year): 4 (1977), Issue (Month): 3 (May). pp. 323–338.

[6] Merton, R. C. *Option pricing when underlying stock returns are discontinuous*. Journal of Financial Economics. 3: 125–144, 1976

[7] Glyn A. Holton *"Fundamental Theorem of Asset Pricing"* October 20, 2011.

[8] Beckers, S., *"Standard deviations implied in option prices as predictors of future stock price variability"*, Journal of Banking and Finance, 5 (3): 363–381, 1981

[9] Yves J. Hilpisch. *Derivatives Analytics with Python: Data Analysis, Models, Simulation, Calibration and Hedging*. Wiley Finance, ISBN: 978-1-119-03799-6, July 2015

[10] Yves J. Hilpisch. *Python for Finance: Analyze Big Financial Data*. O'Reilly Media; 1 edition (27 Dec. 2014), ISBN: 9781491945285

[11] Hastie, Trevor; Tibshirani, Robert Friedman *The Elements of Statistical Learning*, p. 349, 2009

[12] Urij Dolgov *Calibration of Heston's stochastic volatility model to an empirical density using a genetic algorithm* . Research in IVW Köln, 3/2015

[13] Alok Gupta *A Bayesian Approach to Financial Model Calibration, Uncertainty Measures and Optimal Hedging*. Hertford College, University of Oxford, PhD thesis 2009

[14] Cybenko, G. *Approximations by superpositions of sigmoidal functions*. Mathematics of Control, Signals, and Systems, 2(4), 303–314, 1989

[15] Philipp Grohs, Dmytro Perekrestenko, Dennis Elbrächter, Helmut Bölcskei *Deep Neural Network Approximation Theory* arXiv:1901.02220, 2019

[16] Hernandez, Andres, *Model Calibration with Neural Networks*. https://ssrn.com/abstract=2812140, July 20, 2016

[17] Hernandez, Andres, *Model Calibration: Global Optimizer vs. Neural Network* . https://ssrn.com/abstract=2996930, July 3, 2017

[18] D.P. Bertseka *Stoachstic optimization with Nondifferentiable cost functionals* Journal of optimization theory and applications, Vol.12, N.2, 1973

[19] Shan Li, Liying Kang, and Xing-Ming Zhao *A Survey on Evolutionary Algorithm Based Hybrid Intelligence in Bioinformatics* Hindawi Publishing Corporation, BioMed Research International, Volume 2014

[20] Matyas, J. *"Random optimization"* Automation and Remote Control. 26 (2): 246–253., 1965

[21] Storn, R.; Price, K. *Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces.* Journal of Global Optimization, 1997

[22] Wilhelm E. Sorteberg et al. *Approximating the solution to wave propagation using deep neural networks* arXiv:1812.01609

[23] Christof Angermueller et al. *Deep learning for computational biology* Molecular Systems Biology 12(7):878, 2016

[24] G. David Garson *Neural Networks: An Introductory Guide for Social Scientists* London: Sage Publications, 1998

[25] Bishop, Christopher M. *Pattern Recognition and Machine Learning* New York: Springer, 2006.

[26] McCulloch, Warren; Walter Pitts. *A Logical Calculus of Ideas Immanent in Nervous Activity* Bulletin of Mathematical Biophysics, 1943

[27] *Rosenblatt, Frank. x. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* Spartan Books, Washington DC, 1961

[28] *Zhang, Wei. Shift-invariant pattern recognition neural network and its optical architecture.* Proceedings of Annual Conference of the Japan Society of Applied Physics, 1988

[29] Williams, Ronald J.; Hinton, Geoffrey E.; Rumelhart, David E. *Learning representations by back-propagating errors* Nature. 323 (6088): 533–536, October 1986

[30] Yoshua Bengio et al., *Professor Forcing: A New Algorithm for Training Recurrent Networks*, NIPS 2016

[31] Min Lin, Qiang Chen, Shuicheng Yan *Network In Network* ICLR 2014 conference submission

[32] Diederik P. Kingma, Jimmy Ba *Adam: A Method for Stochastic Optimization* arXiv:1412.6980, 2014

[33] A Krizhevsky, I Sutskever, GE Hinton *Imagenet classification with deep convolutional neural networks.* Advances in neural information processing systems, 1097-1105, 2012

[34] Tom Young, Devamanyu Hazarika, Soujanya Poria, Erik Cambria *Recent Trends in Deep Learning Based Natural Language Processing.* arXiv:1708.02709v8, 2018

Published: 2013

[35] Li Deng ; Geoffrey Hinton ; Brian Kingsbury *New types of deep neural network learning for speech recognition and related applications: an overview.* 2013 IEEE International Conference on Acoustics, Speech and Signal Processing

[36] L Di Persio, O Honchar *Artificial neural networks architectures for stock price prediction: Comparisons and applications.* International Journal of Circuits, Systems and Signal Processing 10, 403-413

[37] Md Zahangir Alom et al. *Deep Versus Wide Convolutional Neural Networks for Object Recognition on Neuromorphic System* arXiv:1802.02608

[38] Chin-Wei Hsu, Chih-Chung Chang and Chih-Jen Lin *A practical guide to support vector classification* Technical Report, National Taiwan University, 2010

[39] Vinod Nair and Geoffrey Hinton *Rectified Linear Units Improve Restricted Boltzmann Machines.* ICML, 2014

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun *Deep Residual Learning for Image Recognition.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016

[41] Sergey Ioffe, Christian Szegedy *Batch normalization: accelerating deep network training by reducing internal covariate shift.* Proceeding ICML'15 Proceedings of the 32nd ICML - Volume 37, Pages 448-456

[42] N Srivastava, G Hinton, A Krizhevsky, I Sutskever, R Salakhutdinov *Dropout: a simple way to prevent neural networks from overfitting.* The Journal of Machine Learning Research 15 (1), 1929-1958, 2014

[43] Sivakumar P.B., Mohandas V *Evaluating the Predictability of Financial Time Series*, Innovations and Advanced Techniques in Computer and Information Sciences and Engineering. Springer, Dordrecht

[44] Dean Teneng *Limitations of the Black-Scholes model* International Research Journal of Finance and Economics, January 2011

[45] Hanen Borchani, Gherardo Varando et al. *A survey on multi-output regression* Wiley Interdiscip. Rev. Data Min. Knowl. Discov. 2015

[46] Saltelli, A., K. Chan, and M. Scott (Eds.) *Sensitivity Analysis* Wiley Series in Probability and Statistics. New York: John Wiley and Sons

[47] Karen Simonyan, Andrea Vedaldi, Andrew Zisserman *Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps*. arXiv:1312.6034,2013

[48] Storn, R.; Price, K. *Not Just a Black Box: Learning Important Features Through Propagating Activation Differences*. Journal of Global Optimization, 1997

[49] Matthew D Zeiler, Rob Fergus *Visualizing and Understanding Convolutional Networks*. arXiv:1311.2901, 2013

[50] Bonwell, Charles, Eison, James *Active Learning: Creating Excitement in the Classroom* Information Analyses - ERIC Clearinghouse Products

[51] West, Jeremy; Ventura, Dan; Warnick, Sean *Spring Research Presentation: A Theoretical Foundation for Inductive Transfer* Brigham Young University, College of Physical and Mathematical Sciences, 2007

[52] Alex Nichol, Joshua Achiam, John Schulman *On First-Order Meta-Learning Algorithms* arXiv preprint arXiv:1803.02999, 2018

[53] Travis E, Oliphant *A guide to NumPy*. USA: Trelgol Publishing, (2006)

[54] Eric Jones and Travis Oliphant and Pearu Peterson and others *SciPy: Open source scientific tools for Python*. http://www.scipy.org/

[55] François Chollet *Keras*. https://github.com/fchollet/keras

[56] Martín Abadi, Ashish Agarwal et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. tensorflow.org

[57] *DeepExplain: attribution methods for Deep Learning*. https://github.com/marcoancona/DeepExplain

[58] Andrew Ng *Machine Learning from Stanford University* https://www.coursera.org/learn/machine-learning

Appendix A

Souce code for stochastic process simulation

```
class GBM(SimulationClass):

    def __init__(self, name, mar_env, corr=False):
        super(GBM, self).__init__(name, mar_env, corr)

    def update(self, initial_value=None, volatility=None,
    final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
            self.volatility = volatility
        if final_date is not None:
            self.final_date = final_date
        self.instrument_values = None

    def generate_paths(self, fixed_seed=False, day_count=365.):
        if self.time_grid is None:
            self.generate_time_grid()

        M = len(self.time_grid)
        I = self.paths
        paths = np.zeros((M, I))

        paths[0] = self.initial_value

        if not self.correlated:
            rand = sn_random_numbers((1, M, I),
```

```
            fixed_seed=fixed_seed)
        else:
            rand = self.random_numbers

        short_rate = self.discount_curve.short_rate

        for t in range(1, len(self.time_grid)):
            # select the right time slice from the relevant
            # random number set
            if not self.correlated:
                ran = rand[t]
            else:
                ran = np.dot(self.cholesky_matrix, rand[:, t,
    :])
                ran = ran[self.rn_set]
            dt = (self.time_grid[t] - self.time_grid[t - 1]).
    days / day_count

            # difference between two dates as year fraction
            paths[t] = paths[t - 1] * np.exp((short_rate - 0.5
                * self.volatility ** 2) * dt
                + self.volatility * np.sqrt(dt) * ran)
        # generate simulated values for the respective date
        self.instrument_values = paths
```

Listing A.1: GBM simulation class

```
class JD(SimulationClass):

    def __init__(self, name, mar_env, corr=False):
        super(JD, self).__init__(name, mar_env, corr)
        try:
            # additional parameters needed
            self.lamb = mar_env.get_constant('lambda')
            self.mu = mar_env.get_constant('mu')
            self.delt = mar_env.get_constant('delta')
        except Exception as e:
            print(e)
            print('Error parsing market environment.')


    def update(self, initial_value=None, volatility=None, lamb=
    None, mu=None, delta=None, final_date=None):
        if initial_value is not None:
            self.initial_value = initial_value
        if volatility is not None:
```

```
19              self.volatility = volatility
20          if lamb is not None:
21              self.lamb = lamb
22          if mu is not None:
23              self.mu = mu
24          if delta is not None:
25              self.delt = delta
26          if final_date is not None:
27              self.final_date = final_date
28          self.instrument_values = None
29
30
31      def generate_paths(self, fixed_seed=False, day_count=365.):
32          if self.time_grid is None:
33              self.generate_time_grid()
34
35          M = len(self.time_grid)
36          I = self.paths
37          paths = np.zeros((M, I))
38          paths[0] = self.initial_value
39
40          if self.correlated is False:
41              # if not correlated, generate random numbers
42              sn1 = sn_random_numbers((1, M, I),
43              fixed_seed=fixed_seed)
44          else:
45              # if correlated, use random number object as
    provided
46              # in market environment
47              sn1 = self.random_numbers
48
49          # standard normally distributed pseudorandom numbers
50          # for the jump component
51          sn2 = sn_random_numbers((1, M, I), fixed_seed=fixed_seed
    )
52          rj = self.lamb * (np.exp(self.mu + 0.5 * self.delt ** 2)
     - 1)
53          short_rate = self.discount_curve.short_rate
54
55          for t in range(1, len(self.time_grid)):
56              # select the right time slice from the relevant
57              # random number set
58              if self.correlated is False:
59                  ran = sn1[t]
60              else:
```

```
                # only with correlation in portfolio context
                ran = np.dot(self.cholesky_matrix, sn1[:, t, :])
                ran = ran[self.rn_set]
            dt = (self.time_grid[t] - self.time_grid[t - 1]).
    days / day_count
            # difference between two dates as year fraction
            poi = np.random.poisson(self.lamb * dt, I)
            # Poisson-distributed pseudorandom numbers for jump
    component
            paths[t] = paths[t - 1] * (np.exp((short_rate - rj
            - 0.5 * self.volatility ** 2) * dt
            + self.volatility * np.sqrt(dt) * ran)
            + (np.exp(self.mu + self.delt *
            sn2[t]) - 1) * poi)

        self.instrument_values = paths
```

Listing A.2: Jump diffusion simulation class

Appendix B

Souce code for option valuation

```
1
2 class ValuationEuropean(ValuationClass):
3
4   def generate_payoff(self, fixed_seed=False):
5
6     try:
7       # strike defined?
8       strike = self.strike
9     except:
10      pass
11
12    paths = self.underlying.get_instrument_values(fixed_seed=
      fixed_seed)
13    time_grid = self.underlying.time_grid
14    try:
15      time_index = np.where(time_grid == self.maturity)[0]
16      time_index = int(time_index)
17    except Exception as e:
18      print('Maturity date not in time grid of underlying')
19
20    maturity_value = paths[time_index]
21    # average value over whole path
22    mean_value = np.mean(paths[:time_index], axis=1)
23    # maximum value over whole path
24    max_value = np.amax(paths[:time_index], axis=1)[-1]
25    # minimum value over whole path
26    min_value = np.amin(paths[:time_index], axis=1)[-1]
27    try:
28      payoff = eval(self.payoff_func)
29      return payoff
```

```
30      except:
31        print('Error evaluating payoff function')
```

Listing B.1: European option valuation class

Appendix C

Souce code for neural networks

```
1
2  def dense_bn_block(inn, size):
3      x = Dense(size, activation='linear')(inn)
4      x = BatchNormalization()(x)
5      x = Activation('relu')(x)
6      return x
7
8  def residual_block(inn, size):
9      x = dense_bn_block(inn, size)
10     x = add([inn, x])
11     return x
```

Listing C.1: Basic residual block

```
1
2  inputs = Input(shape=(295, ))
3  x = GaussianNoise(0.05)(inputs)
4  x = BatchNormalization()(x)
5
6  for i in range(1, 5):
7      x = residual_block(x, 64)
8      x = Dropout(0.25)(x)
9
10 predictions = Dense(1, activation='linear')(x)
11
12 model = Model(inputs=inputs, outputs=predictions)
```

Listing C.2: Multilayer perceptron

```
1
```

```
inputs = Input(shape=(295, ))
x = GaussianNoise(0.05)(inputs)
x = Reshape((59, 5))(x)
x = BatchNormalization()(x)
x = Conv1D(64, 3, activaltion = 'relu')(x)
x = GlobalMaxPooling1D()(x)

for i in range(1, 5):
    x = residual_block(x, 64)
    x = Dropout(0.25)(x)

predictions = Dense(1, activation='linear')(x)

model = Model(inputs=inputs, outputs=predictions)
```

Listing C.3: Convolutional neural network

```

inputs = Input(shape=(295, ))
inputs2 = Input(shape=(7, 1, ))

x = GaussianNoise(0.05)(inputs)
x = BatchNormalization()(x)
x = dense_bn_block(x, 64)

for i in range(1, 5):
    x = residual_block(x, 64)
    x = Dropout(0.25)(x)

rnn = LSTM(64, return_sequences=False)(inputs2)
x = concatenate([x, rnn])
predictions = Dense(1, activation='linear')(x)

model = Model(inputs=[inputs, inputs2], outputs=predictions)
```

Listing C.4: Recurrent neural network