Calibration of Heston Model with Keras

Olivier Pironneau

Calibration of Heston Model with Keras

Olivier Pironneau [1]

Abstract

In this work, we consider the calibration of the Heston model for European put or call options on a single or a basket of financial assets. We use the high level environment `Keras` of Google in `Python`. The calibration is done directly with prices corresponding to an array of values for the strike and the maturity or the initial values of the asset and its volatility. The calibration is for the three parameters of the Heston model or the correlation between the asset and the stochastic volatility. It turns out to be a rather easy programming exercise but a large and computer-intensive generation of the synthetic data is necessary to calibrate the Neural Network. A simple network with one hidden layer seems to be appropriate, yet the precision stalls upon a problem dependent threshold beyond which it seems difficult to go.

Keywords: Heston model, Calibration, Option pricing, Neural Networks, Partial differential equations,.

1. Introduction

Applications of Artificial Intelligence methods – neural networks of all kinds – are blossoming everywhere in at least 3 areas of quantitative finance: Market prediction [3][6], portfolio optimization [16], high dimensional futures [4] and calibration [20].
This paper's aim is the calibration of Heston's model for put and call European options making no use of implied volatility, i.e. the non-constant volatility which makes the option a solution of the Black & Scholes model.
The Black & Scholes model [5] proposed in 1973 for the pricing of financial options may be written as :

$$P_T = e^{-rT}\mathbb{E}[(K - X_T)^+], \quad C_T = e^{-rT}\mathbb{E}[(X_T - K)^+] \tag{1}$$

where P_T and C_T are respectively put and call options on the financial asset $\{X_t\}_{t\in(0,T)}$ with *maturity* T and *strike* (price at transaction time T) K; $\mathbb{E}$ denotes the conditional expectation when X_0 is known, and X_t is the solution of the stochastic differential equation,

$$\mathrm{d}X_t = rX_t\mathrm{d}t + X_t\sigma\mathrm{d}W_t, \quad X_{t=0} = X_0. \tag{2}$$

Here r is the interest rate of the money and σ is the *volatility* of the asset; W_t is a Weiner process, or – for practical purpose – $\mathrm{d}W_t$ is a zero-mean Gaussian variable with variance

[1] *olivier.pironneau@gmail.com* , LJLL, Sorbonne University, Paris, France.

$\sqrt{\mathrm{d}t}$. The correct analytical setting of the problem is well established and can be found in many books, including [1].
Heston's model [14] is a a popular extension of the Black & Scholes model. It uses a stochastic volatility , $\sigma = \sqrt{V_t}$,

$$\mathrm{d}X_t = rX_t\mathrm{d}t + X_t\sqrt{(V_t)}\mathrm{d}W_t \tag{3}$$

modelled by a mean reverting process correlated to X_t,

$$\mathrm{d}V_t = \kappa(\theta - V_t)\mathrm{d}t + \lambda\sqrt{(V_t)}\mathrm{d}\bar{W}_t, \quad V_{t=0} = V_0, \quad \rho\mathrm{d}t = \mathbb{E}[\mathrm{d}W_t \cdot \mathrm{d}\bar{W}_t]. \tag{4}$$

where W and. $\bar{W}$ are correlated Weiner processes. The meaning of the parameters of the model are

- κ is the mean reversion rate,
- θ is the long run variance,
- λ is the volatility of the volatility,
- V_0 is the square of the initial volatility
- ρ is the correlation coefficient.

The question that *calibration* attempts to answer is : can these parameters be adjusted to market data so as to predict more futures with the same model.
Calibration for Heston model has a long story; one very interesting early paper is by Mikhailov & Nogel [15] where sub-percent precision on market data (SP500 of 23^{rd} July 2002) could be reached in very fast CPU time. The problem is classic enough that `python` codes are publicly available on the internet [2], Quantlib [17] etc. A well tested method is to interpolate the data on an implied volatility surface and then use a least-square fit with the Levenberg - Marquardt quasi-Newton Method [18].
Calibration by Neural Network has been tested by many authors, such as [13],[19], [20] to cite a few. Although it works in general it has not been adopted yet by practitioners, probably because precision is an issue: a large amount of data is needed to reach sub-percent precision.
So the purpose of this study is not to propose yet a new competitive method but to investigate the feasibility of Artificial Neural Network for the calibration, with one specific application in mind: the portability of the neural network on smaller computers like phones and tablets, even for the calibration of large baskets options.
But even if the feasibility is asserted, calibration on market data will still be needed to check the precision and the stability. The results are outlined in the conclusion of this article.
Before we proceed we make one important remark:
Here, unlike [20], we do not calibrate the parameters using implied volatilities; we use directly the option prices. Then the market data are a set of put and/or call prices for a set of maturities and strikes on the same asset or set of assets in case of a basket. Due to the nature of the problem it is equivalent to assume that we have a set of prices for different initial asset prices and different maturities or times. Indeed in the model

above what counts is not X_0 and K but X_0/K, because $P_T = Ke^{-rT}\mathbb{E}[(K - X_T/K)^+]$. So we can keep K fixed and give P for a set of X_0 as data for the calibration. In the same vein, the maturity is simply the final time of the simulation. So we can compute $P_t = Ke^{-rt}\mathbb{E}[(K - X_t/K)^+]$ and use these as data, equivalent to a set of maturities.

2. Numerical Solutions

2.1. Solution with a Complex Integral

The semi-analytical formula for a Heston call [8] involves a complex integral :

$$\begin{aligned} C &= \frac{1}{2}(X_0 - Ke^{-rT}) + \frac{1}{\pi}\Re\int_0^\infty (e^{rT}\frac{\phi(u-\mathbf{i})}{\mathbf{i}uK^{\mathbf{i}u}} - K\frac{\phi(u)}{\mathbf{i}uK^{\mathbf{i}u}})\mathrm{d}u \quad \text{with} \\ \phi(u) &= e^{rT}X_0^{\mathbf{i}u}\left(\frac{1-ge^{-dT}}{1-g}\right)^{-2\frac{\theta\kappa}{\lambda^2}} \exp\big(\frac{\theta\kappa T}{\lambda^2}(\kappa - \rho\lambda\mathbf{i}u - d) + \frac{V_0}{\lambda^2}(\kappa - \rho\lambda\mathbf{i}u + d)\frac{1-e^{dT}}{1-ge^{dT}}\big) \\ d &= \sqrt{(\rho\lambda u\mathbf{i} - \kappa)^2 + \lambda^2(\mathbf{i}u + u^2)}, \quad g = \frac{\kappa - \rho\lambda\mathbf{i}u - d}{\kappa - \rho\lambda\mathbf{i}u + d}. \end{aligned} \tag{5}$$

The complex integral requires a careful discretization as explained in [15]; A `python` implementation is given in Appendix 6.1

2.2. Solution of the Feynman-Kac PDE by the Finite Element Method

Ito Calculus leads to the following equivalent problem : $P_T = u(X_0, V_0, 0)$ is solution of

$$\partial_t u + rx\partial_x u + \frac{x^2 y}{2}\partial_{xx}u + \kappa(\theta - y)\partial_y u + \frac{\lambda^2 y}{2}\partial_{yy}u + \rho\lambda xy\partial_{xy}u - ru = 0$$

with $u(x, y, T) = (K - x)^+$. A change of notation has been made: $X \mapsto x$, $V \mapsto y$, For numerical computations the domain needs to be localised to a rectangle $(X_{min}, X_{max}) \times (V_{min}, V_{max})$ with sides labeled Γ_1 (bottom, $y = V_{min}$), Γ_2 (right, $x = X_{max}$), Γ_3 (top $y = V_{max}$), Γ_4 (left, $x = X_{min}$) and boundary conditions need to be applied.
The method is easy to extend to options with barriers; then boundary conditions imposed on Γ_2 and/or Γ_4: $u(\cdot, \cdot, t) = 0$ are the barriers at $x = X_{min}$ and $x = X_{max}$.
It is known that a weak solution exists if the Feller condition [9] is satisfied: $\lambda^2 < 2\kappa\theta$; if it is not satisfied the solution may not be unique and will exist only under certain compatibility conditions on the data (Fredholm's alternative).

Remark 1.

- *The market price of volatility risk has been taken to be zero.*
- *If X_{max} is large then the boundary condition on Γ_2 is a localization for the put. If it is not large then it corresponds to a put option with a barrier at X_{max}.*
- *If $X_{min} = 0$ then the boundary condition is irrelevant, otherwise it is a lower barrier on the option.*

After discretization in time, $u^m = u(\cdot, \cdot, t)$ solves the following variational problem

$$\int_\Omega \Big[u^m w(r + \frac{1}{dt}) + \frac{x^2 y}{2}\partial_x u^m \partial_x w + \frac{\lambda^2}{2} y\partial_y u^m \partial_y w + \rho\lambda xy\partial_y u^m \partial_x w$$

$$+\left((\kappa+\rho\lambda)y+(\frac{\lambda^2}{2}-\kappa\theta)\right)w\partial_y u^m+x(y-r)w\partial_x u^m\Big]$$
$$+\int_{\lambda_3}\frac{\lambda^2 r}{2(\kappa+\rho\lambda)}u^m w=\int_\Omega\frac{u^{m-1}w}{dt};\tag{6}$$

with $u^m=0$ on Γ_2 and Γ_4.
Finally the Finite Element Method of degree 1 or 2 on triangles can be used for spatial discretization, resulting in a time independent linear system for each m. A numerical solution can be obtained in less than a second on an Intel Core i7 and the reader is invited to test the `freefem` script of Appendix 6.2.

2.3. Monte-Carlo Solutions

For basket options the put is a function of several assets: $P(\vec{X}_0,\vec{V}_0)=e^{-rT}\mathbb{E}[\phi(\vec{X}_T)|\vec{X}_0,\vec{V}_0]$, for some pay-off ϕ, and each asset is modelled by a Heston set of stochastic differential equations:

$$\mathrm{d}X_t^i=X_t^i(r\mathrm{d}t+\sigma^i\sqrt{V_t^i}\mathrm{d}W_t^i),\quad \mathrm{d}V_t^i=\kappa^i(\theta^i-V_t^i)\mathrm{d}t+\lambda^i\sqrt{(V_t^i)}\mathrm{d}\bar{W}_t^i\tag{7}$$

with $\mathbb{E}[\mathrm{d}W_t^i\mathrm{d}W_t^j]=\rho^{ij}\mathrm{d}t$, $\mathbb{E}[\mathrm{d}W_t^i\mathrm{d}\bar{W}_t^j]=\bar{\rho}^{ij}\mathrm{d}t$ and $\mathbb{E}[\mathrm{d}\bar{W}_t^i\mathrm{d}\bar{W}_t^j]=\overline{\rho}^{ij}\mathrm{d}t$. The simplest method is to apply an Euler discretization in time to $Y_t=\log X_t$ with a round-off at $\epsilon<<1$ against negative volatilities:

$$Y^{n+1}=Y^n+r\delta t+\sigma\sqrt{V^n}B_{0,1}\sqrt{\delta t},\quad V^{n+1}=\max\{\epsilon,V^n+\kappa(\theta-V^n)\delta t+\lambda\sqrt{V^n}\bar{B}_{0,1}\sqrt{\delta t}\}\tag{8}$$

where $B_{[0,1]}$ and $\bar{B}_{[0,1]}$ are Gaussian correlated unit random variables. For vectorial system the scheme is not very different and the complexity is only in the generation of the correlated vector valued B and $\bar{B}$ at each time step ($t=n\delta t$).
Algorithm (8) must be run a large number of times (Monte-Carlo paths) and P is taken to be the mean of the expression on all paths. The method is slow but advantageous when d is larger than 2.

3. Calibration

Calibration of the parameters $\kappa,\theta,\lambda,\rho,V_0,...$ is an important long standing problem in quantitative finance. Not only a solution must be found which reproduces the data but it must be stable in the sense that adding new data should not change the parameters drastically. The data can be a mix of calls and puts for different strikes or maturity. Often it is thought that a more stable solution is found if the calibration is done against implied volatilities rather than prices, but shall not follow that path.

3.1. Solution with Heston's formula

On the basis of having as many equations as unknowns, it is theoretically not excluded to compute parameters like κ,θ,λ from the knowledge of 3 option prices only, $\{\Pi_{T_k}^{K_k}\}_{k=1}^3$, by inverting the nonlinear system,

$$P_{T_k}^{K_k}(\kappa,\theta,\lambda)=\Pi_{T_k}^{K_k}\quad k=1,2,3\tag{9}$$

with a root-finding algorithm. Here $\{T_k, K_k\}_1^3$ are given and used to compute $\{\Pi_{T_k}^{K_k}\}_{k=1}^3$ with the Heston semi-analytical formula; Heston's formula with maturity T and strike K defines a non-linear mapping $\{\kappa, \theta, \lambda\} \mapsto P_T^K(\kappa, \theta, \lambda)$.
And indeed it works when the root-finding iterative method is initialized not too far from the solution.
An example in `python` is given in Appendix 6.1 using the solver `Broyden1` with initial values $[\kappa, \theta, \lambda] = [1.4, 0.03, 0.5]$. From the knowledge of 3 synthetic Heston prices computed with $[\kappa, \theta, \lambda] = [1.5768, 0.0398, 0.575]$, with $K = 100$ and $T = 1.8$, or $T = 2.1$, or $T = 2.2$, the solver found the true values of the 3 parameters to 5 digits: $[1.5768, 0.0398, 0.5750]$. An equally good identification was found from the synthetic prices computed with $T = 2$, $K = 90$ or $K = 105$ or $K = 110$. However the computing time takes 4 forbidding minutes on a Mac pro 15" core i7, 2017. Naturally, stability and uniqueness is not guaranteed.

3.2. Solution by optimization

In quantitative finance nothing ensures that the data follows Heston's model, so only an approximate calibration can be envisaged. For practical purpose, knowing M values of an option for a different strikes K and maturities T what are the parameters for Heston's model that reproduce best these market values?
In the world of inverse problems [21] involving a PDE such problems are solved by

$$[\kappa, \theta, \lambda] = \text{argmin}_{[\kappa,\theta,\lambda]\in U_{ad}} \sum_{k=1}^{M} \|P_T^k(\kappa, \theta, \lambda) - \Pi_T^k\|^2 \mid \text{ subject to } P^k \text{ given by (6).} \quad (10)$$

For stability it is wise to add a penalization term, but here there is no theoretical need for it as this optimization problem has always a solution because it is in $\mathbb{R}^3$ and the solution of the PDE depends continuously on its coefficients, provided that U_{ad} be compact and in the range for which the solution exists.
Evidently one may use methods, other than the PDE, to compute Heston prices.

3.3. Solution with CMA-ES

CMA-ES [11] is a powerful generic stochastic optimizer. We have applied it with a criteria for the minimization built from a noisy modification of the solution of the PDE at final time $\omega(x, y)u(x, y, T)$ with x, y restricted to the lower right quarter of the domain, i.e $(x, y) \in (\frac{1}{2}X_{max}, X_{max} \times (0, \frac{1}{2}V_{max})$; 20×20 values uniformly distributed in this rectangle are taken (M=400). The noise is uniform in $(0.99, 1)$.
All financial and numerical parameters are easily readable from the `freefem` program in Appendix 6.3 and illustrated by Figure 1. Initial values for the optimizer are $[\kappa, \theta, \lambda] = [6, 0.05, 1]$. The target solution for the criteria corresponds to $[\kappa, \theta, \lambda] = [3, 0.1, 0.2]$. After 1000 function evaluations the exact solution $[3, 0.1, 0.2]$ was found.
The results are very good but too many iterations were needed for this method to be CPU-time competitive.

3.4. Solution with Levenberg-Marquardt Least-squares

A fully functional `python` program is available at [2] with explanations using the `Jupyter` note-book. The method computes the prices with Heston's semi-analytical formula (implemented with QuantLib[17]); implied volatilities are computed and the optimization

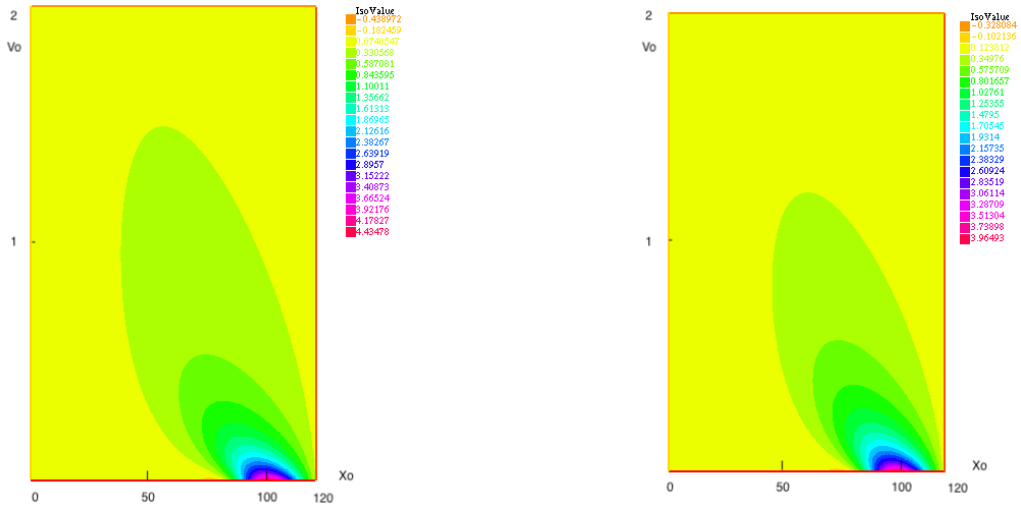

Fig. 1: Two solutions (calls with a right barrier at 120) of the Heston PDE: left $\kappa = 1.4,\ \theta = 0.03,\ \lambda = 0.5$. Right: $\kappa = 1.5768,\ \theta = 0.0398,\ \lambda = 0.575$. Other parameters are K=100, r=0.03, $\rho = -0.5711$, T=1. Note that even though the displays are not very different the optimizers can tract the differences.

problem is solved by the Levenberg-Marquardt[18] quasi Newton algorithm. We reproduce in Table 1 some computed values (for a different set of parameters) computed after calibrating $\kappa, \theta, \lambda, \rho, V_0$.

Table 1: **Performance of the Levenberg-Marquardt implemented** in [2]

Strikes	Market Value	Model Value	Relative Error (%)
527.50	44.67893	44.46556	0.4775547
560.46	55.05277	55.23288	0.3271670
593.43	67.37152	67.66592	0.4369746
626.40	80.93411	81.82830	1.1048434

3.5. Solution with the MNIST Neural Network

MNIST (see [22] or [10]) is a well known test for character recognition; it takes as input M icon size images of a handwritten character to train a convolution neural network; once trained, the network is able to recognize very quickly a new character with impressive precision.

Keras [7] is a toolbox written by François Chollet over TensorFlow (by Google) which provides an easy to use implementation of Yann Lecun's convolution network for MNIST. Observing the final state of the solution of the PDE at time T is, after all, like observing an image. To fit the MNIST standard all is needed is to downgrade the graphic image of the solution from 28×28 black and white image – MNIST stardard – to 20×20 color or B&W (see fig 1).

So the `freefem` program was run M=9900 times with uniformly random values for κ, θ and λ in $(0,3) \times (0,0.1) \times (0,0.2)$ and the results were store in a file together with the corresponding values of κ, θ, λ.
To make it harder we multiplied each pixels values by a white noise as above and we stored in the file 20×20 equally spaced values of the lower right part of the image.
In the end the Python-Keras program is very short (the full program is given in Appendix 6.4) and it is a minor modification of the MNIST example of Keras where the softmax activations are replaced by ReLU activations to change the classification problem of MNIST into an optimization problem. The core of the program is as follows:

```
model = Sequential()
input_shape = (pix,pix,1)
x_train=x_train.reshape(x_train.shape[0], pix,pix, 1)
x_test=x_test.reshape(x_test.shape[0], pix,pix, 1)
model.add(Conv2D(32,kernel_size=(pix,pix),activation='relu',
               input_shape=input_shape))
model.add(Flatten())
model.add(Dense(neurons, input_dim=pix2,use_bias=True,
               bias_initializer='random_uniform',activation='relu'))
model.add(Dense(num_classes=3, use_bias=True, bias_initializer='random_uniform'))
model.compile(loss=losses.mean_squared_error, optimizer=
            keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0))
model.fit(x_train,y_train,batch_size=32,epochs=300,verbose=1,validation_split=0.1)
```

With `pix=20, neurons=50` and 9900 samples an excellent precision was obtained in less than a minute. The biggest CPU cost is the generation of samples; the learning phase of the Network is a few minutes.
For example in one test the loss function was decreased from 0.0210 to $1.2106e-04$ in 358 iterations/epochs and the average absolute errors computed on 32 test cases is

$$\mathbb{E}\big[\|[\kappa,\theta,\lambda]_{NN} - [\kappa,\theta,\lambda]_{true}\|_{L^\infty}\big] = [0.00941, 0.00211, 0.00477]$$

The average relative error is $[0.5\%, 4\%, 2\%]$.

But then the convolution layer will be problematic for problems which do not have a cartesian structured input. It turns out that equally precise results are obtained without the convolution layer; i.e. without the instruction `model.add(Conv2D...` above, on the condition that the number of `neurons` is increased; `neurons=1000` in our tests.
Now that it is no longer necessary to have a structured data set for input we can reduce the number of data point as follows: we pick $M_d \times M_d$ points randomly in the set of 20×20 points x, y for which $u(x, y, T)$ is computed by the PDE solver.
When $M_d = 10$ (i.e. 100 put values) the performance of the neural network is still good: after 263 epochs, the absolute error is $[0.0142, 0.00238, 0.00443]$ (see Table 2).
But with $M_d = 7$ (i.e. 49 put values) the performance of the neural network deteriorates: after 217 epochs the precision is $[0.0224, 0.00448, 0.00705]$. (see Table 2)
A configuration with an auto-encoder type of architecture was tested: one hidden layer with 100 neurons, another with 10 and a third one with 100. The results are not better : average absolute error=[0.026521170.004858640.00668083], average relative error= $[1.4\%, 8.9\%, 2.3\%]$.

Remark 2. *So these are calibration problems using options prices which correspond to different asset prices X_0 and different initial volatilities V_0. As said in the introduction it is equivalent to a set of strikes and initial volatilities. Later we will take for data sets of strikes and maturities.*

Table 2: **Neural Network optimization using $M_d \times M_d$ values** $\{u(X_0^j, V_0^j, T)\}_{j=1}^{400}$ corresponding to a uniform 20×20 grid in the X_0, V_0 domain when $M_d = 20$. Comparison between 5 Neural Network solutions and the true 5 solutions when 9900 samples are used. After 358 iterations (epochs) the average relative error on $[\kappa, \theta, \lambda]$ is $[0.31\%, 2.11\%, 2.4\%]$.. Then the same simulation is launched with $M_d = 10$; 100 random points are used in the 20×20 grid for the Network; after 263 iterations (epochs) the average relative error on $[\kappa, \theta, \lambda]$ is $[0.6\%, 4\%, 2\%]$. Finally with $M_d = 7$, i.e. 49 random points in the grid are used to train the network; after 217 iterations (epochs) the average relative error on $[\kappa, \theta, \lambda]$ is $[0.6\%, 4\%, 1.9\%]$

M_d	κ_{NN}	θ_{NN}	λ_{NN}	κ_{true}	θ_{true}	λ_{true}
400	2.423	0.0448	0.1555	2.433	0.0446	0.1565
- -	2.709	0.0411	0.1158	2.715	0.0416	0.119
- -	2.360	0.100	0.1478	2.351	0.0989	0.1576
- -	1.900	0.0104	0.163	1.902	0.00872	0.169
- -	0.1662	0.0601	0.0242	0.1668	0.0665	0.0261
100	2.594	0.0682	0.0813	2.578	0.066	0.0905
- -	1.835	0.0620	0.1874	1.794	0.062	0.1956
- -	1.938	0.100	0.1742	1.933	0.099	0.158
- -	0.1524	0.026	0.0148	0.130	0.0020	0.0193
- -	2.851	0.017	0.0498	2.850	0.0161	0.0563
49	2.855	0.01960	0.0679	2.8505	0.0161	0.0563
- -	2.406	0.04613	0.163	2.433	0.0446	0.156
- -	2.667	0.0437	0.128	2.715	0.0416	0.119
- -	2.345	0.104	0.153	2.351	0.0989	0.157
- -	1.916	0.0125	0.178	1.902	0.0087	0.169

4. Calibration with historical data

Can Heston model be calibrated with the values at different Maturities of a call or put with all other parameters equal, same strikes, same V_0 and same ρ, etc?
To check we generated 9900 solutions of the PDE and stored for each one the values of $u(X_0, V_0, j\delta t), j = 1, 2, .., T/N_{max}$.
Then we took $N < N_{max}$ values at random in this set for each line of data and train the network with these.
For $N_{max} = 40$ and $N = 20$, the average absolute and relative error are $[0.18430, 0.0088, 0.0194]$ and $[7\%, 7\%, 8\%]$ Not too good!

4.1. Calibration of one parameter from historical data

Under the same conditions we trained the network to recover one of the 3 parameters from the knowledge of the 2 others and N=20 historical data randomly chosen from the 40 available $\{j\delta t\}_0^{40}$. The precision is worse: hardly less than 11% (see Table 4).

Table 3: **Calibration on historical data using 20 time values** $\{u(X_0, V_0, t^j)\}_{j=1}^{20}$ with t^j (uniform) random in $\{j\delta t\}_0^{40}$. Comparison between 4 Neural Network solutions and the true 4 solutions when 9900 samples are used. The average absolution error on $[\kappa, \theta, \lambda]$ is $[7\%, 7\%, 8\%]$

κ_{NN}	θ_{NN}	λ_{NN}	κ_{true}	θ_{true}	λ_{true}
1.452	0.0338	0.0473	0.952	0.0255	0.00881
0.948	0.0650	0.1221	0.679	0.0655	0.0943
1.192	0.0328	0.0644	0.518	0.0278	0.0186
1.960	0.0555	0.105	2.2991	0.0459	0.129

Table 4: **Calibration of one parameter only on historical data using 20 values** $\{u(X_0, V_0, t^j)\}_{j=1}^{20}$ with t^j (uniform) random in $0.T$. Comparison between 4 Neural Network solutions and the true 4 solutions when 9900 samples are used. The average relative error is 7% for the identification of κ, 6% for the identification of θ and 11% for the identification of λ.

κ_{NN}	κ_{true}		θ_{NN}	θ_{true}		λ_{NN}	λ_{true}
1.395	0.952		0.0477	0.0459		0.188	0.197
0.862	0.679		0.0812	0.0816		0.118	0.0974
0.964	1.26		0.0847	0.0856		0.0544	0.0329
2.372	2.299		0.0791	0.0770		0.111	0.09431

4.2. Discussion

- A simple hidden layer does just as well as two convolution layers or 3 smaller hidden layers.
- It is difficult to improve the precision beyond a certain threshold: augmenting the number of samples does not improve much.
- Tabulating from historical data only is harder but doable, precision is less that with dat with a mix of financial parameters. Identification of one parameter only knowing the 2 others isn't more precise than identifying the 3 together.

5. High dimensional Heston Calibration

Let us consider a basket put option

$$P(X_0, V_0, 0) = e^{-rT}\mathbb{E}[(K - \sum_1^d X^i(T))^+ | X_0, V_0] \tag{11}$$

where X_t^i is a financial asset following

$$\mathrm{d}X_t^i = X_t^i(r\mathrm{d}t + \sigma^i\sqrt{V_t}\mathrm{d}W_t^i), \text{ with } \mathrm{d}V_t = \kappa(\theta - V_t)\mathrm{d}t + \lambda\sqrt{(V_t)}\mathrm{d}W_t \tag{12}$$

with $\mathbb{E}[\mathrm{d}W_t^i \mathrm{d}W_t] = \rho^i \mathrm{d}t$.
In other words, each asset is given by the Heston model with the same unique equation for the stochastic volatility but the correlation parameter is different.
The numerical simulations are done using the Monte-Carlo algorithm with Z paths and the Euler finite difference in time scheme with 100 time steps and the following parameters

- $d = 3,\ T = 1,\ K = 600,\ r = 0.01$
- $\kappa = 10.9811,\ \theta = 0.132331,\ \lambda = 4.018157,\ \rho = [-0.35156, -0.5, -0.2]$
- $X_0 = [259.37, 100, 150],\ V_0 = 0.198778$

To solve the calibration problem with a neural network we need to feed in many cases (samples) with as much information as possible. Here each case is obtained by multiplying each values above for κ, θ, λ by uniformly random numbers between 0.5 and 1.5. For each case we compute $N_T \times N_K$ prices corresponding to maturities $\{\frac{jT}{N_T}\}_{j=1}^{N_T}$ and strikes $\{\frac{jK}{N_K}\}_{j=1}^{N_K}$. In all simulations below $N_T = 10$ and $N_K = 5$

5.1. Basket with 3 assets

The precision is a function of the number of samples given to the Neural Network and of the number of Monte-Carlo paths to compute the prices. The later is fixed at Z=10000 except for the first case (i.e. 1000 samples) where Z= 1000.
In Table 3 we display the quality of the results obtained. With 1000 samples an absolute average precision reached for on $[\kappa, \theta, \lambda]$ is $[1.177, 0.03137, 0.31215]$. With 2000 samples it is $[0.79810, 0.04900, 0.20953]$. With 5000 it is $[0.6741, 0.04087, 0.2630]$ and with 7000 samples it is $[0.7380, 0.02974, 0.1681]$.

Table 5: **Basket of 3 Options: Results as a function of the number of samples**. The data for each samples consists of 50 option prices computed with Monte-Carlo using $Z = 10000$ paths except when 1000 samples are used for which $Z = 1000$.

Samples	κ_{NN}	θ_{NN}	λ_{NN}	κ_{true}	θ_{true}	λ_{true}
1000	9.476542	0.13677481	3.0140927	8.312585	0.15803926	2.625773
- -	11.418259	0.16013892	5.845906	12.375428	0.14412636	5.858743
- -	10.904998	0.14866751	4.6934037	8.208858	0.16059723	4.3550353
- -	7.127512	0.11386083	2.9758203	7.4216833	0.0747664	3.3328052
2000	8.878693	0.06084843	2.6711965	8.079459	0.11457606	2.5260932
- -	12.800058	0.10885634	3.043743	10.208906	0.18123053	2.771562
- -	9.325264	0.08825119	3.17426	9.705886	0.11318509	3.3946927
- -	5.1053705	0.06585675	3.025517	6.481651	0.07561707	3.335925
5000	14.863845	0.12428152	3.7099943	15.191024	0.15111904	3.9258523
- -	7.4602294	0.09437443	5.7484875	7.507711	0.1756598	5.406438
- -	6.229864	0.08913025	2.8549767	6.3737087	0.14227197	2.5560155
- -	13.475615	0.06076711	3.8859534	14.284962	0.07732525	4.079917
7000	13.528603	0.17181566	2.924739	14.82564	0.11166272	3.2351701
- -	9.166567	0.13196863	3.2088246	9.5206	0.09378843	3.3152626
- -	14.621558	0.21809958	2.440264	13.822117	0.16655973	2.6039271
- -	7.1853437	0.15212183	5.805213	7.188229	0.14879756	5.7630634

5.2. Basket with 6 assets

With the same model as above we now make the problem more difficult by assuming that there are 6 assets in the basket defining the put option. We used the following data

$$X_0 = [60, 100, 150, 25, 50, 70] \text{ and } \{\rho^j\}_1^6 = [-0.3, -0.5, -0.2, -0.1, -0.7, -0.4]. \quad (13)$$

Table 6: **Basket with 3 assets.** Influence of the number of samples on the average relative (in %) and absolute precision

Samples	κ	θ	λ	κ	θ	λ
1000	3%	8%	4%	1.177	0.0313	0.312
2000	2%	15%	2%	0.798	0.0490	0.209
5000	1.5%	7%	1.8%	0.674	0.0408	0.263
7000	1.9%	6%	1%	0.7380	0.0297	0.168

For the identification of the parameters $[\kappa, \theta, \lambda]$ by a Neural Network, the results are shown in Table 7.

Table 7: **Basket with 6 assets: identification of the Heston parameters.** Neural Network results when 1000 samples are used. The data for each samples consists of 50 option prices computed with Monte-Carlo using $Z = 1000$ paths. After 661 iterations the average absolute error on $[\kappa, \theta, \lambda]$ is $[0.97531, 0.1171, 0.2626]$ and an average relative precision $[2\%, 25\%, 3\%]$. Comparison between 5 Neural Network solutions and 5 true solutions are given in the table below.

κ_{NN}	θ_{NN}	λ_{NN}	κ_{true}	θ_{true}	λ_{true}
6.058073	-0.02947991	3.9451385	5.610405	0.19017245	3.6811452
10.025507	0.04691502	2.866429	8.822984	0.11969639	2.8319445
13.495879	0.02573741	4.7397323	15.226152	0.09759247	5.4014325
10.919542	0.05577407	2.803923	9.267343	0.14904015	2.5582633
14.163019	0.0266959	4.6819143	13.727917	0.1479749	4.781782

5.3. Identificaton of ρ

Finally we turn our attention to the identification of the correlation parameters $\{\rho^j\}_1^6$. Due to the computing cost of the exercise we used only 1000 samples each computed with 1000 Monte-Carlo paths. These were generated by multiplying each component of the correlation vector$[-0.3, -0.5, -0.2, -0.1, -0.6, -0.4]$ by a uniformly random number between 0.5 and 1.5.
After 256 iterations the Neural Network reached an average absolute error of:

$$[0.071, 0.080, 0.043, 0.020, 0.14, 0.086] \text{ and relative error } [6\%, 4\%, 5\%, 6\%, 6\%, 6\%].$$

More results are shown in in Table 8.

5.4. Discussion

The following points can be made.

- The precision on the price of the put is not the same as the Neural Network error on the parameters; in general we observed a factor of two. For example, the first line of Table 5 with 7000 samples is one of the worse case: $[\kappa, \theta, \lambda] = [13.529, 0.1718, 2.925]$ leads to $P = 253$ while $[\kappa, \theta, \lambda] = [14.83, 0.112, 3.24]$ leads to $P = 238.5$. One the other hand the last line of Table 5 is a good case:

Table 8: **Basket with 6 assets: identification of ρ.** Neural Network results when 1000 samples are used. The data for each samples consists of 50 option prices for different maturity and strike, as before, computed with Monte-Carlo using $Z = 1000$ paths. After 265 iterations the average relative error is $[6\%, 4\%, 5\%, 6\%, 6\%, 6\%]$. Comparison between 5 Neural Network solutions and 5 true solutions are given below.

ρ^1_{NN}	ρ^2_{NN}	ρ^3_{NN}	ρ^4_{NN}	ρ^5_{NN}	ρ^6_{NN}	ρ^1_{true}	ρ^2_{true}	ρ^3_{true}	ρ^4_{true}	ρ^5_{true}	ρ^6_{true}
-0.265	-0.386	-0.164	-0.0992	-0.516	-0.353	-0.302	-0.321	-0.130	-0.132	-0.694	-0.268
-0.313	-0.617	-0.223	-0.110	-0.651	-0.488	-0.333	-0.710	-0.171	-0.0963	-0.537	-0.521
-0.329	-0.693	-0.243	-0.113	-0.696	-0.533	-0.334	-0.683	-0.292	-0.102	-0.318	-0.586
-0.321	-0.656	-0.233	-0.112	-0.674	-0.511	-0.217	-0.720	-0.287	-0.126	-0.754	-0.324
-0.273	-0.425	-0.1745	-0.101	-0.539	-0.376	-0.245	-0.360	-0.177	-0.0847	-0.378	-0.515

$[\kappa, \theta, \lambda] = [13.414, 0.157, 5.60]$ gives a price for the put $P = 279.3 \pm 1$ while $[\kappa, \theta, \lambda] = [13.65, 0.150, 5.565]$ gives $P = 279.5 \pm 1$. Here the precision is masked by the accuracy of the Monte-Carlo algorithm.

- The gain in precision obtained by augmenting the number of samples is disappointingly slow.
- The performance of the Neural Network is not really affected by the kind of solver used to compute the synthetic prices. Whether Heston's formula or the PDE or Monte-Carlo is used, the performance of the Network is the same. This could be attributed to the relative stability of neural networks with respect to noise in the data.'
- . Keras is a very friendly tool and the neural network optimization is almost the same for all examples.

6. Conclusion

In all our examples the simple Neural Network used here does the job of minimizing the loss function; it is made of an input layer, a single hidden layer with 1000 neurons and an output layer, hence it is very fast, hardly more than a minute even for multidimensional problems. The MNIST network with 2 convolution layers, which is adapted to the case of a regular array of price data, did not perform better. But there are two big limitations:

1. The cost of generating synthetic data with an Heston model solver is very CPU expensive.
2. We have not found a way to improve the precision beyond the sub-percent range, generally demanded by practitioners.

The two items are related because while 10.000 samples are hardly enough for the calibration of an option with a single asset, it is way too little for a mutliple assets basket option. And then the CPU times becomes very large. For a 3 assets basket the Monte-Carlo algorithm with 10000 paths required 7 hours to generate 10000 prices (for a set of maturities and strikes, but that doesn't count for much). Naturally multi-level and

quasi-Monte-Carlo refinements would dramatically reduce the CPU time, but still, in the face of it, we need probably several hundreds of thousands of samples, it is taxing.
Finally Keras is very well adapted to the task, in the three cases we considered: solutions of Heston models generated by Heston's semi-analytical formula or by the Feynman-Kac PDE or by Monte-Carlo, the difference between the first two and the last one is the presence of noise due to insufficient precision of Monte-Carlo for the prices.
In the end, even if we could solve the precision problem there is an imperative need to compare the prediction on market data and perhaps also use market data for the training of the Network. Work for the future!

References

[1] Achdou, A. and O. Pironneau, (2005): *Computational methods for option pricing*. Vol. 30, Frontiers in Applied Mathematics, SIAM series.

[2] Balaraman Goutham: `http://gouthamanbalaraman.com/blog/volatility-smile-heston-model-calibration-quantlib-python.html`

[3] Banushev B. (2019) : https://github.com/borisbanushev/stockpredictionai

[4] Beck C. and S. Becker, Ph. Grohs, N. Jaafari, and A. Jentzen (2018): Solving stochastic differential equations and Kolmogorov equations by means of deep learning. arXiV 1806.00421v1.

[5] Black, F. and M. Scholes (1973): The pricing of options and corporate liabilities, Journal of Political Economy 81, no. 3, 637–659.

[6] Borovykh A. and Sander Bohte and C.W. Oosterlee (2018-): Conditional time series forecasting with convolutional neural networks. arXiv:1703.04691v5.

[7] Chollet, F. (2015): Keras, Deep learning library for Theano and Tensorflow. https://keras.io/k

[8] Cox, J., J. Ingersoll and S. Ross (1985): A theory of the term structure of interest rates. Econometrica, 53: 389-408. (1985)

[9] Feller, W. (1951): Two singular diffusion problems, Annals of Math., 2nd Series, 54(1), 173–182.

[10] Goodfellow I. and Y. Bengio and A. Courville (2016): *Deep Learning*, MIT-Bradford.

[11] Hansen N. (2006): The CMA Evolution Strategy : A Comparing Review, in www.lri.fr/∼hansen/cmaesintro.html.

[12] Hecht F. (2012): New development in FreeFem++, J. Numer. Math., 20, pp. 251-265. (see also www.freefem.org.)

[13] Hernandez, Andres, Model Calibration with Neural Networks (July 20, 2016). Available at `http://dx.doi.org/10.2139/ssrn.2812140`

[14] Heston S. (1993): A closed-form solutions for options with stochastic volatility, Review of Financial Studies, 6, 327–343.

[15] Mikhailov, S. and Ulrich Nogel, (2003): Heston's Stochastic Volatility Model Implementation, Calibration and Some Extensions, Wilmott Magazine, vol 1 (July), p74-79.

[16] Palaniappan V. (2019) `https://github.com/VivekPa/NeuralNetworkStocks`

[17] Quantlib: `https://pypi.org/project/QuantLib-Python/`

[18] Marquardt, D: An Algorithm for Least-Squares Estimation of Nonlinear Parameters. SIAM J. Appl. Math. 11, 431-441, 1963.

[19] De Spiegeler J, Madan DB, Reyners S, Schoutens W. Machine learning for quantitative finance: fast derivative pricing, hedging and fitting. Quantitative Finance. 2018;p. 1–9.

[20] Shuaiqiang Liu and A. Borovykh and A. Lech and A. Grzelak and C.W. Oosterlee (2019): A neural network-based framework for financial model calibration

[21] Tarantola A. (1987): Inverse Problem Theory. Elsevier Science.

[22] Zhang X. and J Zhao and Y LeCun (2015): Character-level convolutional networks for text classification. Advances in neural information processing systems, 649-657.

Appendix: Programs

6.1. Calibration with the Heston solver

```
""" @author: pironneau, August 2019 """
import numpy as np
from scipy.optimize import broyden1
X0 = 95
V0 = 0.1
r = 0.03
kappa = 1.5768
theta=0.0398
lambd=0.575
rho=-0.5711

def heston(kappa,theta,lambd,T,K):
    I=complex(0,1)
    P, umax, N = 0, 1000, 10000
    du=umax/N
    aa= theta*kappa*T/lambd**2
    bb= -2*theta*kappa/lambd**2
    for i in range (1,N) :
         u2=i*du
         u1=complex(u2,-1)
         a1=rho*lambd*u1*I
         a2=rho*lambd*u2*I
         d1=np.sqrt((a1-kappa)**2+lambd**2*(u1*I+u1**2))
         d2=np.sqrt((a2-kappa)**2+lambd**2*(u2*I+u2**2))
         g1=(kappa-a1-d1)/(kappa-a1+d1)
         g2=(kappa-a2-d2)/(kappa-a2+d2)
         b1=np.exp(u1*I*(np.log(X0/K)+r*T))*( (1-g1*np.exp(-d1*T))/(1-g1) )**bb
         b2=np.exp(u2*I*(np.log(X0/K)+r*T))*( (1-g2*np.exp(-d2*T))/(1-g2) )**bb
         phi1=b1*np.exp(aa*(kappa-a1-d1)\
            +V0*(kappa-a1-d1)*(1-np.exp(-d1*T))/(1-g1*np.exp(-d1*T))/lambd**2)
         phi2=b2*np.exp(aa*(kappa-a2-d2)\
            +V0*(kappa-a2-d2)*(1-np.exp(-d2*T))/(1-g2*np.exp(-d2*T))/lambd**2)
         P+= ((phi1-phi2)/(u2*I))*du
    return K*np.real((X0/K-np.exp(-r*T))/2+P/np.pi)
# Example of usage of heston()
T,K=2,100
call = heston(kappa,theta,lambd,T,K)
print("call = ",call, " put = ", call-X0+K*np.exp(-r*T))
# example of calibration
price1=heston(kappa,theta,lambd,T,90)
price2=heston(kappa,theta,lambd,T,105)
price3=heston(kappa,theta,lambd,T,110)

def F(x):
    return    [(price1-heston(x[0],x[1],x[2],T,90)), \
               (price2-heston(x[0],x[1],x[2],T,105)), \
               (price3-heston(x[0],x[1],x[2],T,110))]
x = broyden1(F, [1.4,0.03,0.5], f_tol=1e-14)
print("[kappa,theta,lambda] =",x)
```

The program gives the following results:

```
call =  12.356330803154561  put =  11.532784161579428
[kappa,theta,lambda] = [1.5768 0.0398 0.575 ]
```

6.2. *freefem script to solve the Heston PDE*

The freefem PDE solver for Mac, PC, Linux, is available for free download at www.freefem.org.

```
//verbosity=0;
int n=15, Nmax=5*n; // controls mesh and time step
real   T= 1, // maturity
       K = 600, // strike
       S0 = 659.37, // spot
       kappa = 10.9811 , // stoch vol params kappa, theta, lambda
       theta = 0.132331, // dS_t=S_t(r dt + sqrt(v_t)d W_1t
       lambda = 4.018157, // dv_t = kappa(theta-v_t) + lambda sqrt(v_t)dW_2t
       xlam=0., // extra parameter only in the PDE
       r = 0.01, // interest
       q = 0.,  // dividend
       rho=-0.351560, // correlation W_1.W_2
       sig0=0.198778,  // initial vol
       Smin = 0,  // Left barrier
       Smax=1000, // Right barrier ( orlocalization in S (-> x) if no right
           barrier)
       vmax=2,  // localization in v (-> y)
       LL = 100 // scale for y (results should not depend on choice of LL)
;
mesh th = square(5*n,5*n,[Smin*(1-x)+x*Smax,y*LL*vmax]); // uniform mesh

fespace Vh(th,P1);
func f =max(K-x,0.); //put
Vh uref=f,u,uold,w;
int ji=0;
real   mu=r-q,
       dt =T/Nmax,
       t = 0,
       a11 = 1./LL/2.,
       a22 = lambda^2*LL/2,
       a21 =  rho * lambda,
       b1  =  mu + 1/dt,
       c1  = (kappa+xlam+rho*lambda),
       c2  = (lambda^2/2  -kappa*theta)*LL,
       c3 = lambda^2*mu/(2*c1);
/////////////// the solver //////////
problem HESTON(u,w,init=ji) = int2d(th)( u*w*b1
            + dx(u) * dx(w) * x * x * y * a11
            + dy(u) * dy(w) * y * a22
            + dy(u) * dx(w) * y * x * a21
            + dy(u) * w * (c1 * y + c2)
            + dx(u) * w * (y/LL-mu)*x
          ) + int1d(th,3)(u*w*c3)
            + on(2, u=0)  + on(4,u=K*exp(-r*t))
            - int2d(th)( uold*w/dt);
u=f;
for (int  n=0; n<Nmax ; n++){
      uold=u;
      HESTON; t= t+dt; ji=1;
      cout<<u(S0,sig0)<<endl;
}
cout<<"Put at time zero ="<<u(S0,sig0)<<endl;
////////////////////////////////  end solver /////////////////
```

6.3. The optimization test with CMA-ES and the generation of samples for the Neural Network.

```
real noise=0.01,box=0.5;  // box in (0,1), 1 is full domain

int NV=3;  // number of unknown or random parameters
real[int] cc(NV);  // random coefficients

int iimax=20, jjmax=20, // iimax x jjmax points output
//int iimax=25, jjmax=25, // iimax x jjmax points output
    Ntrain=10000;  // number of PDE results for training and testing


srandomdev();
randinit(random());
// For Neural Network
// train images
for(int k=0;k<Ntrain;k++){
    cc[0] = kappa0*randreal1();
    cc[2] = lambda0*randreal1();
    cc[1] = theta0*randreal1();
//  cout<<"kappa= "<<cc[0]<<"  theta= "<<cc[1]<<"  lambda= "<<cc[2]<<endl;
    fff<<cc[0]<<"  "<<cc[1]<<"  "<<cc[2]<<"  ";
    cost(cc); // solves PDE
    real umax=u[].max, umin=u[].min ;
    if(umax-umin<0.01) umax=umin+0.01;
//  plot(u, fill=1);
    if(! timesignal)
        for(int jj=0;jj<jjmax;jj++)
            for(int ii=0;ii<iimax;ii++)
                fff << noise*randreal1()

+(u(Smax*(1-box+ii*box/(iimax-1)),LL*vmax*jj*box/(jjmax-1))
                                -umin)/(umax-umin)/(1+noise)
                << " ";
    fff<<endl;
    cout<<k<<" of "<<Ntrain<<endl;
}
// below is by genetic optimization
/*
cc[0]=kappa0; cc[1]=theta0; cc[2]=lambda0;
cout<<"exact solution "<<cc[0]<<" "<<cc[1]<< " "<<cc[2]<<endl;
cost(cc); uref=u;
plot(th,u,dim=3,fill=1,wait=1,ps="hestonE.eps");
cc[0]=2*kappa0;   cc[1]=0.5*theta0; cc[2]=5*lambda0;
cout<<"initial start "<<cc[0]<<" "<<cc[1]<< " "<<cc[2]<<endl;
cost(cc);
//plot(th,u,dim=3,fill=1,wait=1,ps="heston0.eps");
real minimum =
cmaes(cost,cc,stopTolFun=0.1e-5,stopMaxFunEval=1000,stopMaxIter=100000);
cout<<"minimum= "<<minimum<<" Solution: kappa= "<<cc[0]<<" theta= "<<cc[1]<<"
lambda= "<<cc[2]<<endl;
*/
```

```
import keras
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, Flatten
from keras import losses
from keras.callbacks import EarlyStopping

Z = np.loadtxt('data.txt')
n_test, n_train = 32,9900
pix=20
pix2 = pix*pix
epochs = 1000
batch_size=32
num_classes = 3
npoint=10 # to take the full image set npoint=pix
id=np.random.uniform(0,pix2,size=npoint*npoint)
pix,pix2 = npoint, npoint*npoint
x_test  = np.zeros([n_test,pix2],'float32')
x_train = np.zeros([n_train,pix2],'float32')
y_test= np.zeros([n_test, num_classes],'float32')
y_train= np.zeros([n_train, num_classes],'float32')
#############################################################
for i in range(n_train):
    for j in range(num_classes):
        y_train[i,j] = Z[i,j]
    for j in range(pix2):
        x_train[i,j] = Z[i,int(id[j])+num_classes]
#        x_train[i,j] = Z[i,j+num_classes]  # use this if npoint=pix
for i in range(n_test):
    k=n_train+i
    for j in range(num_classes):
        y_test[i,j] = Z[k,j]
    for j in range(pix2):
        x_test[i,j] = Z[k,int(id[j])+num_classes]
#        x_test[i,j] = Z[k,j+num_classes]

model = Sequential()
input_shape = (pix,pix,1)
x_train=x_train.reshape(x_train.shape[0], pix,pix, 1)
x_test=x_test.reshape(x_test.shape[0], pix,pix, 1)
model.add(Flatten())
model.add(Dense(1000, input_dim=pix2,use_bias=True,
               bias_initializer='random_uniform',activation='relu'))
model.add(Dense(num_classes, use_bias=True, bias_initializer='random_uniform'))

model.compile(loss=losses.mean_squared_error, optimizer=
    keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0))
es = EarlyStopping(monitor='val_loss',mode='min',verbose=1,min_delta=1.e-7,
                   patience=40,restore_best_weights=True)
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
          verbose=1, validation_split=0.1,callbacks=[es])
u_test = model.predict(x_test)
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', (score)**0.5)
som=0
for i in range(n_test):
    som+=np.abs(u_test[i]-y_test[i])
    print(u_test[i]," ", y_test[i])
print("Average abs error= ",som/n_test)
```

1

```
"""
Created on Sat Aug 10 12:20:09 2019
@author: pironneau
"""
import numpy as np
import time

def mc_heston(option_type,S0,K,T,initial_var,long_term_var,rate_reversion
              ,vol_of_vol,corr,r,num_reps,steps, ntime,nK):
    """
    option_type:    'p' put option 'c' call option
    S0[d]:           the spot price of underlying stock; d is size of S0
    T:              the maturity of options
    initial_var:    the initial value of variance
    long_term_var:  the long term average of price variance
    rate_reversion: the mean reversion rate for the variance
    vol_of_vol:     the vol of vol(the variance of the variance of stock price)
    corr[d]:        the correlation between W1[.] and W2
    r:              the risk free rate
    reps:           the number of repeat for monte carlo simulation
    steps:          the number of steps in each simulation
    """
    eps=0.0001
    delta_t = T/float(steps)
    Vt=np.zeros([int(steps)])
    W1=np.zeros([int(steps)])
    d=np.size(S0)
    st=np.array(d)
    corr2=np.sqrt(1-corr**2)
    aux1 = 0.5 * vol_of_vol * np.sqrt(delta_t)
    aux2 = 0.25 * vol_of_vol**2 * delta_t
    payoff = np.zeros([int(steps/ntime),nK])
    for i in range(num_reps):
        vt = initial_var
        for j in range(steps):
            W1[j] = np.random.normal(0, 1)
            vt = (np.sqrt(vt) + aux1 * W1[j])**2\
                 - rate_reversion * (vt - long_term_var)*delta_t -aux2
            vt=max(eps,vt)
            Vt[j] = vt
        st = np.log(S0)
        for j in range(steps):
            aux3 = (r - 0.5*Vt[j])*delta_t
            aux4 = np.sqrt(Vt[j]*delta_t)
            for k in range(d):
                w2=(corr[k]*W1[j]+corr2[k]*np.random.normal(0, 1))
                st[k] += aux3 + aux4*w2
            jj=int(j/ntime)
            if jj*ntime==j :
                sst=np.sum(np.exp(st))
                for k in range(nK):
                    if option_type == 'c':
                        payoff[jj,k] += max(sst - (K*k)/(nK-1), 0)
                    elif option_type == 'p':
                        payoff[jj,k] += max((K*k)/(nK-1)- sst, 0)
    return (payoff/float(num_reps)) * (np.exp(-r*T))
 ######################################################################
S0=np.array([60,100, 150,25,50,70])
K=600
```

1

```
"""
Created on Sat Aug 10 12:20:09 2019
@author: pironneau
"""
import numpy as np
import time

def mc_heston(option_type,S0,K,T,initial_var,long_term_var,rate_reversion
              ,vol_of_vol,corr,r,num_reps,steps, ntime,nK):
    """
    option_type:    'p' put option 'c' call option
    S0[d]:           the spot price of underlying stock; d is size of S0
    T:              the maturity of options
    initial_var:    the initial value of variance
    long_term_var:  the long term average of price variance
    rate_reversion: the mean reversion rate for the variance
    vol_of_vol:     the vol of vol(the variance of the variance of stock price)
    corr[d]:        the correlation between W1[.] and W2
    r:              the risk free rate
    reps:           the number of repeat for monte carlo simulation
    steps:          the number of steps in each simulation
    """
    eps=0.0001
    delta_t = T/float(steps)
    Vt=np.zeros([int(steps)])
    W1=np.zeros([int(steps)])
    d=np.size(S0)
    st=np.array(d)
    corr2=np.sqrt(1-corr**2)
    aux1 = 0.5 * vol_of_vol * np.sqrt(delta_t)
    aux2 = 0.25 * vol_of_vol**2 * delta_t
    payoff = np.zeros([int(steps/ntime),nK])
    for i in range(num_reps):
        vt = initial_var
        for j in range(steps):
            W1[j] = np.random.normal(0, 1)
            vt = (np.sqrt(vt) + aux1 * W1[j])**2\
                 - rate_reversion * (vt - long_term_var)*delta_t -aux2
            vt=max(eps,vt)
            Vt[j] = vt
        st = np.log(S0)
        for j in range(steps):
            aux3 = (r - 0.5*Vt[j])*delta_t
            aux4 = np.sqrt(Vt[j]*delta_t)
            for k in range(d):
                w2=(corr[k]*W1[j]+corr2[k]*np.random.normal(0, 1))
                st[k] += aux3 + aux4*w2
            jj=int(j/ntime)
            if jj*ntime==j :
                sst=np.sum(np.exp(st))
                for k in range(nK):
                    if option_type == 'c':
                        payoff[jj,k] += max(sst - (K*k)/(nK-1), 0)
                    elif option_type == 'p':
                        payoff[jj,k] += max((K*k)/(nK-1)- sst, 0)
    return (payoff/float(num_reps)) * (np.exp(-r*T))
 ######################################################################
S0=np.array([60,100, 150,25,50,70])
K=600
```

1