Application de l'apprentissage supervisé à la calibration des modèles de la finance quantitative

Breaking the ice between theory and practice

Olivier Pironneau[1]

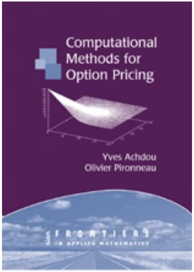
[1]Sorbonne Université (Jussieu - Paris VI - UPMC), Laboratoire J.-L. Lions (LJLL)
http://ljll.math.sorbonne-universite.fr/pironneau

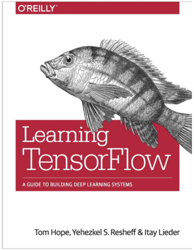
Mini cours SUMMIT 05/02/21

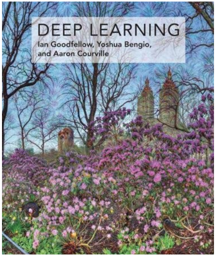
Content

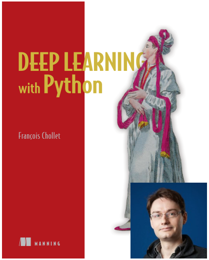
1. Option Pricing
2. Calibration with standard Optimization methods
3. Introduction to neural network
4. Calibration with Neural Networks

A personal travel-log









⇒ + **numpy** and **keras** manual (François Chollet)

Heston Model

Heston's model > Black & Scholes model for a derivative asset like a put $P_T = e^{-rT}\mathbb{E}[(K - X_T)^+]$ on an underlying financial asset X_t. It uses a stochastic volatility , $\sigma = \sqrt{V_t}$,

$$dX_t = rX_t dt + X_t\sqrt{(}V_t)dW_t$$

modeled by a mean reverting process correlated to X_t,

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

W and $\bar{W}$ are correlated Weiner processes. K, r, strike , r interest rate,

- 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.

calibration attempts to fit these parameters to reproduce market data.

Stochastic Optimization

Let $\{\Pi_{T_k}^{K_k}\}_1^M$ be the data. We may solve

$$[\kappa, \theta, \lambda] = \text{argmin}_{[\kappa,\theta,\lambda,V_0]\in U_{ad}} \sum_{k=1}^{M} \|P_{T_k}^{K_k}(\kappa, \theta, \lambda, V_0) - \Pi_{T_k}^{K_k}\|^2,$$

with $P_{T_k}^{K_k}$ given by Heston model with parameters κ, θ, λ. Mote-Carlo solution :

$$\begin{aligned}
&X^0, V^0 \text{ given do M times :}\\
&X_m^{n+1} = X_m^n + r\delta t + \sigma\sqrt{V_m^n} B_{0,1}^{m,n}\sqrt{\delta t},\\
&V_m^{n+1} = \max\{\epsilon, V_m^n + \kappa(\theta - V_m^n)\delta t + \lambda\sqrt{V_m^n}\bar{B}_{0,1}^{m,n}\sqrt{\delta t}\}\\
&P_{T_k}^{K_k}(\kappa, \theta, \lambda, V_0) = \frac{1}{M}\sum_{m=1}^{M}(K - X^N)^+
\end{aligned}$$

Works also if X is a vector, provided $K - X^N$ is changed to $K - Y \cdot X^N$.
It can be solved by optimization packages like CMA-ES.

Numerical Results

CMA-ES (www.lri.fr/∼hansen/cmaesintro.html)
Initial values are $[\kappa, \theta, \lambda] = [6, 0.05, 1]$. Target solution is $[3, 0.1, 0.2]$.
1000 cost function evaluations $\Rightarrow$ target solution was found... CPU-time $>> 1$.

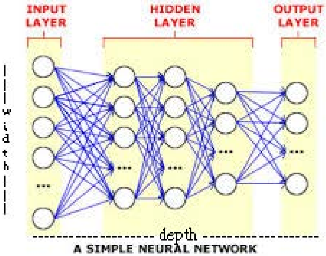
TABLE – **Performance of the Levenberg-Marquardt implemented** in `http://gouthamanbalaraman.com/blog /volatility-smile-heston-model-calibration-quantlib-python.html`

Strikes	Market Value	Model Value	Relative Error (%)
527.50	44.67893	44.46556	1.9
560.46	55.05277	55.23288	1.3
593.43	67.37152	67.66592	1.7
626.40	80.93411	81.82830	4.4

Neural Networks

A Feed-Forward Deep Neural Network is made of

- an input layers,
- L hidden layers in between. Each has an ir
- an output layer



- A hidden Layer $l \in \{1, .., L\}$ does

$$\boxed{x^l_{out} = Ax^l_{in} + b, \qquad x^{l+1}_{in} = \sigma(x^l_{out})}$$

- with x, b vectors, A matrix,
- where the activation function σ is

$$\mathrm{ReLU}(x) = \max\{x, 0\}$$

- The number of neurons in a layer is its width ; L is the depth of the DNN.

Solution with a Neural Network (I)

- The Neural Network : one input layer, one hidden layer with 1000 neurons, one output layer,
- Inputs=$[\kappa, \theta, \lambda, V_0, \rho]$.
- Output : $P_{T_k}^{K_k}$.
  - $d = 3,\ T = 1,\ K = 600,\ r = 0.01$
  - $\kappa = 10.9811,\ \theta = 0.132331,\ \lambda = 4.018157,\ \rho = [-0.35156, -0.5, -0.2]$
  - $X_0 = [259.37, 100, 150],\ V_0 = 0.198778$

Synthetic observations Π : 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 $N_K = 5$

Solution with a Neural Network (II)

Basket with 3 assets The number of Monte-Carlo paths is Z=10000. With 1000 samples an absolute mean precision 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 – **Basket of 3 Options : Results as a function of the number of samples**.

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
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
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
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

Solution with a Neural Network (III)

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

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

For the identification of the parameters $[\kappa, \theta, \lambda]$ by a Neural Network, the results are shown in Table 3.

TABLE – **Basket with 6 assets.** 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 $[8\%, 60\%, 12\%]$. 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

Neural Network : Existence of Solution

Remark : The output of L Neuron layers with an activation $ReLU(x) = \max(0, x)$ is a piecewise linear function in x, L-polynomial in a_{ij}, b_j.

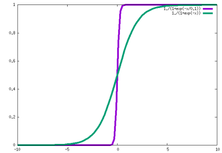
$x \mapsto f(x) = \max(0, A_1 x_2 + b_2) \mid x_2 = \max(0, A_1 x_1 + b_1) \mid x_1 = \max(0, Ax + b)$

Theorem[1] *ReLU networks are at least as expressive as free knot linear splines.*

The Representation Theorem (Hornik et al., Cybenko, 1989-91) *Feed-forward network with one hidden layer of large enough width and a "squashing" activation function can approximate any integrable function to any accuracy.*

Remark (Bruno Després) Let $f \in C^1(R) \cap W^{1,\infty}(\mathbb{R})$

$$\begin{aligned} f(x) &= \int_{-\infty}^{x} f'(y)dy = \int_R H(x-y)f'(y)dy \approx \sum_{j=-J}^{J} \phi(\frac{x}{\epsilon} - \frac{j\delta x}{\epsilon})f'(j\delta x)\delta x \\ &= \sum_{-J}^{J} \omega_j \phi(a_j x + b_j) \end{aligned}$$



where $H(x)$ is Heaviside and ϕ a sigmoid to approximate H.
Notice that a, b tend to infinity with precision.

1. Nonlinear Approximation and (Deep) ReLU Networks I. Daubechies, R. DeVore, S. Foucart, B. Hanin, and G. Petrova In. arXiv :1905.02199

A Toy Problem : One single hidden layer

Compute $x, y \in \mathbb{R}^d \mapsto x \cdot y \in \mathbb{R}$

TABLE – Influence of the # samples on Loss (DNN=100. Batch size=32, epoch=200)

Samples\ Dim	1	2	4	8	16	32
1000	.0136				.245	
10000	.0029				.089	
20000	.0029	.0076	.013	.037	.072	
30000	.0020				.069	
40000	.0011				.067	
50000	.0013	.0037	.0131	.074	.061	
100000	.0013	.0010	.0043	.012	.030	.057

Conclusion : 10% precision is easy, 1% is hard ; the minimum loss is too large (SG is noisy). In principle, enlarging the network should improve the precision but with this implementation it doesn't seem to happen.

Deep Network

Theorem [2] . *A bounded function of $x \in \mathbb{R}$ with bounded first derivative can be approximated with precision $O(1/(N \log N))$ by the network of fig 5 of fixed width (here 5) and of depth N.*

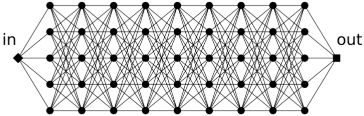


Fig 1 : multilayer perceptron (MLP)

TABLE – Influence of the depth of the network on the Loss (DNN to simulate $x, y \mapsto xy$)

Samples\ nb of Layers N	2	4	8	16	32
10000	.013	.011	.004	.011	.167
20000				.067	.185

Here too precision improves with depth – but much slower – , until so value beyond which it increases.

2. Dmitry Yarotsky, Quantified advantage of discontinuous weight selection in approximations with deep neural networks, arXiv : 1705.01365v1

Conclusion

Final Remarks

- Keras + Tensorflow does things that others (CMAES) don't do
- Convergence can be slow and insufficient
- Not so hard to learn thanks to Google
- Serious theoretical limitations for certification
- Deep learning is useful for optimization, inverse problems, CPU accelerations, games (Pareto)
- ... but not just these of course.

Thanks for the invitation