Artificial neural networks and Wiener-Hopf factorization*

*supported by Russian Science Foundation grant № 23-21-00474

Elena V. Alymova [2, 3]
Oleg E. Kudryavtsev [1, 3]

[1] Southern federal university
[2] Russian Customs Academy, Rostov-on-Don
[3] InWise Systems, LLC

Modern Methods, Problems and Applications of Operator Theory and Harmonic Analysis
Rostov-on-Don, 20 - 25 August, 2023

Historical background and main goal

Pricing path-dependent options in exponential L´evy models still remains a mathematical and computational challenge.

Methods for pricing barrier options: drawbacks

- Monte Carlo methods: *slow*
- Finite difference schemes: *application entails a detailed analysis of the underlying L´evy process*
- Wiener-Hopf factorization methods: *non trivial approximate formulas are needed in general case*

The main goal

Suggest a hybrid numerical method to price barrier options under L´evy processes. The main advantage of the approach is approximation of Wiener-Hopf factors with ANNs.

L´evy processes: a short reminder

General definitions

A Lévy process is a stochastically continuous process with stationary independent increments (for general definitions, see e.g. Sato (1999)). A Lévy process can be completely specified by its characteristic exponent, ψ, definable from the equality $E[e^{i\xi X(t)}] = e^{-t\psi(\xi)}$.

The characteristic exponent of L´evy process

The characteristic exponent is given by the Lévy-Khintchine formula:

$$\psi(\xi) = \frac{\sigma^2}{2}\xi^2 - i\mu\xi + \int_{-\infty}^{+\infty} (1 - e^{i\xi y} + i\xi y 1_{|y|\leq 1}) F(dy),$$

where σ^2 is the variance of the Gaussian component, and the Lévy measure $F(dy)$ satisfies $\int_{\mathrm{R}\setminus\{0\}} \min\{1, y^2\} F(dy) < +\infty$.

The Wiener-Hopf factorization

X_t is a Lévy process with characteristic exponent $\psi(\xi)$,
$\bar{X}_t = \sup_{0\leqslant s\leqslant t} X_s$, $\underline{X}_t = \inf_{0\leqslant s\leqslant t} X_s$ are supremum and infimum processes

Let $q > 0$, $T_q \sim \mathrm{Exp}\, q$. Introduce the following characteristic functions:

$$\phi_q^-(\xi) = E[e^{i\xi \underline{X}_{T_q}}], \phi_q^+(\xi) = E^x[e^{i\xi \bar{X}_{T_q}}].$$

The Wiener-Hopf factorization formula

$$q(q+\psi(\xi))^{-1} = \phi_q^+(\xi)\phi_q^-(\xi).$$

Main ideas

Carr's randomization or the Laplace transform reduces the pricing problem to the calculation of the appropriate sequence of expectations of the following type

$$V_q(x) = E\left[G(x + X_{T_q})1_{[h;+\infty)}(x + \underline{X}_{T_q})\right],$$

Each expectation can be calculated using the Wiener-Hopf factorization method and the Fast Fourier Transform algorithm when the factors are known.

$$V_q = \mathcal{F}^{-1}_{\xi\to x}\phi^-_q(\xi)\mathcal{F}_{x\to\xi}1_{[h;+\infty)}\mathcal{F}^{-1}_{\xi\to x}\phi^+_q(\xi)\mathcal{F}_{x\to\xi}G,$$

where $\mathcal{F}$ is the Fourier transform

An efficient approximation of the Wiener-Hopf factors in the exact formula for the solution is obtained by using artificial neural networks.

Explicit formulas for approximations of $\phi^{\pm}$

For small positive d and large $M(=2^n)$, set

$$\begin{aligned} p_k &= \frac{d}{2\pi}\int_{-\pi/d}^{\pi/d}\left(q(q+\psi(\xi))^{-1}\right)e^{-i\xi kd}d\xi, \\ \phi_q^+(\xi) &\approx \sum_{k=0}^{M/2} p_k^+ e^{i\xi kd}, \quad \sum_{k=0}^{M/2} p_k^+ = 1, p_k^+ \geqslant 0; \\ \phi_q^-(\xi) &\approx \sum_{k=0}^{M/2-1} p_k^- e^{-i\xi kd}, \quad \sum_{k=0}^{M/2-1} p_k^- = 1, p_k^- \geqslant 0; \end{aligned}$$

Problem: $\sum_{k=-M/2+1}^{M/2} p_k e^{i\xi kd} = \sum_{k=0}^{M/2} p_k^+ e^{i\xi kd} \cdot \sum_{k=0}^{M/2-1} p_k^- e^{-i\xi kd}$

Input: p_k, $k = -M/2+1, ..., M/2$

Output: p_k^+, $k = 0, ..., M/2$; $\quad p_k^-, k = 0, ..., M/2-1$.

The first task

We have:

$p_1, p_2, \dots, p_{M-1} \mid p_i \geq 0,\ \sum_{i=1}^{M-1} p_i = 1, M = 2^N, N \in \mathbb{N}$

We need:

$q_1, q_2, \dots, q_{M/2} \mid q_i \geq 0,\ 0 \leq q_i < q_{i+1} < 1, \sum_{i=1}^{M/2} q_i = 1$

$r_1, r_2, \dots, r_{M/2} \mid r_i \geq 0,\ 1 \geq r_{i+1} > r_i > 0, \sum_{i=1}^{M/2} r_i = 1$

$$q(x) = q_1 + q_2 x + \cdots + q_{M/2} x^{\frac{M}{2}-1}$$

$$r(x) = r_1 + r_2 x + \cdots + r_{M/2} x^{\frac{M}{2}-1}$$

$$p(x) = p_1 + p_2 x + \cdots + p_{M-1} x^{M-2}$$

$$p(x) = r(x)q(x)$$

The first task

Let M = 4

$$\frac{c}{x} + b + ax^2 = \left(\frac{\alpha_1}{x} + \alpha_2\right)(\beta_1 + \beta_2 x),$$

$$\alpha_i, \beta_i \ (i = 1, 2) - increase$$

$$\alpha_2 = 1 - \alpha_1$$

$$\beta_1 = 1 - \beta_2$$

Multiply and divide by X:

$$\frac{c + bx + ax^2}{x} = \frac{(\alpha_1 + \alpha_2 x)(\beta_1 + \beta_2 x)}{x}$$

$$c + bx + ax^2 = (\alpha_1 + \alpha_2 x)(\beta_1 + \beta_2 x)$$

Train / valid data generation

```python
def gen_pair():
    import random as rnd
    alpha1 = rnd.random()
    while alpha1 == 0:
        alpha1 = rnd.random()
    alpha2 = 1 - alpha1
    pair = [alpha1, alpha2]
    pair.sort()
    return pair
```

```python
def gen_factor_coeffs():
    alpha = gen_pair()
    beta = gen_pair()
    beta.sort(reverse=True)
    return alpha, beta
```

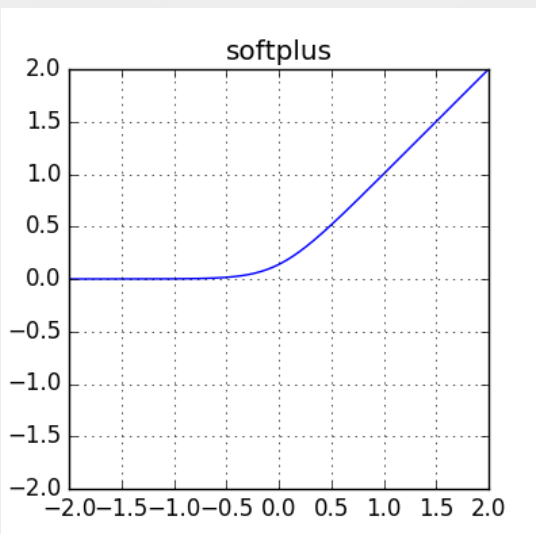
```python
def calc_abc(alpha, beta):
    a = alpha[1] * beta[1]
    b = alpha[1] * beta[0] + alpha[0] * beta[1]
    c = alpha[0] * beta[0]
    return [a, b, c]
```

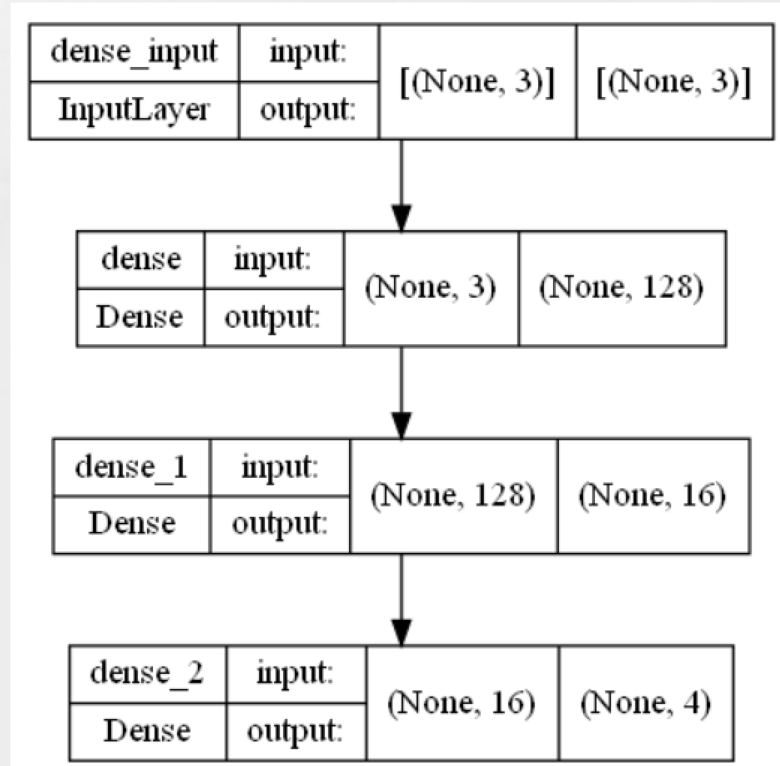
ANN Model

Activation functions:

- Sigmoid (input, hidden layers)
- Softplus (output layer)

Softplus function: $f(x) = \ln(1+e^{x})$





Loss function

- Standard MSE function
- Custom loss function

```
def custom_mse_loss_function(y_true, y_pred):
    y_complex_dist_1 = tf.signal.rfft(y_true[0:2:]) - tf.signal.rfft(y_pred[0:2:])
    y_complex_dist_2 = tf.signal.rfft(y_true[2:4:]) - tf.signal.rfft(y_pred[2:4:])
    y_real_dist_1 = tf.square(tf.math.real(y_complex_dist_1)) + tf.square(tf.math.imag(y_complex_dist_1))
    y_real_dist_2 = tf.square(tf.math.real(y_complex_dist_2)) + tf.square(tf.math.imag(y_complex_dist_2))
    solution_1_dist = tf.reduce_sum(y_real_dist_1)
    solution_2_dist = tf.reduce_sum(y_real_dist_2)
    sum_one_condition = tf.square(tf.math.abs(tf.reduce_mean(y_pred[0] + y_pred[1]) - 1)) +  \
                        tf.square(tf.math.abs(tf.reduce_mean(y_pred[2] + y_pred[3]) - 1))
    return tf.reduce_mean([solution_1_dist, solution_2_dist], axis=-1) + sum_one_condition
```

Model training results

Standard loss function (MSE)

- Quick training – 75 epochs
- Evaluation results:

```
31250/31250 [==============================] - 40s 1ms/step - loss: 2.1537e-04
```

- Average time prediction: 0.0420 seconds

Custom loss function

- Training – 170 epochs
- Evaluation results:

```
31250/31250 [==============================] - 40s 1ms/step - loss: 5.5001e-04
```

- Average time prediction: 0.0450 seconds

Model predicted values

Standard loss function (MSE)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	A	B	C	Alpha1	Alpha2	Beta1	Beta2	Alpha1_p	Alpha2_p	Beta1_p	Beta2_p	A_p	B_p	C_p	Sum_Alpha	Sum_Beta
2	0.033629	0.817616	0.148755	0.154919	0.845081	0.960206	0.039794	0.159645	0.842417	0.959895	0.038037	0.032043	0.814704	0.153242	1.0020613	0.997932
3	0.304617	0.654308	0.041075	0.060793	0.939207	0.675666	0.324334	0.057696	0.93492	0.686067	0.326142	0.304917	0.660235	0.039583	0.9926159	1.012209
4	0.159781	0.50246	0.337759	0.49334	0.50666	0.684639	0.315361	0.493067	0.51053	0.686645	0.314143	0.160379	0.505446	0.338562	1.003597	1.000787
5	0.142522	0.659284	0.198193	0.244257	0.755743	0.811414	0.188586	0.243866	0.756629	0.816185	0.186049	0.14077	0.662921	0.19904	1.000495	1.002235
6	0.416504	0.505367	0.078129	0.153873	0.846127	0.507752	0.492248	0.156099	0.836872	0.510762	0.491852	0.411617	0.504219	0.079729	0.9929702	1.002613

Custom loss function

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	A	B	C	Alpha1	Alpha2	Beta1	Beta2	Alpha1_p	Alpha2_p	Beta1_p	Beta2_p	A_p	B_p	C_p	Sum_Alpha	Sum_Beta
2	0.122807	0.875602	0.001591	0.001815	0.998185	0.87697	0.12303	0.00964	1.004542	0.878601	0.122159	0.122714	0.883769	0.008469	1.0141813	1.00076
3	0.262846	0.736574	0.00058	0.000787	0.999213	0.736947	0.263053	0.01033	1.002742	0.7343	0.26803	0.268765	0.739082	0.007585	1.0130719	1.002329
4	0.061194	0.937977	0.000829	0.000883	0.999117	0.938752	0.061248	0.008902	1.004077	0.938491	0.065677	0.065944	0.942902	0.008354	1.0129785	1.004168
5	0.084625	0.909768	0.005608	0.00613	0.99387	0.914854	0.085146	0.011654	0.99874	0.917566	0.08863	0.088518	0.917442	0.010694	1.0103941	1.006196
6	0.284507	0.712591	0.002902	0.004062	0.995938	0.714333	0.285667	0.01209	0.998082	0.712195	0.290034	0.289478	0.714336	0.008611	1.010172	1.002229

```
0.19314222055002805x^2+0.7322057221735289x+0.07465205727644308
Y_real: tf.Tensor([ 1.0000001 -2.2351742e-08j -0.21028669-5.6945819e-01j], shape=(2,), dtype=complex64)
Y_predict: tf.Tensor([ 1.0046489 +0.j        -0.21767092-0.5751702j], shape=(2,), dtype=complex64)
REAL alpha_1= 0.09490402450153967, alpha_2= 0.9050959754984603, beta_1= 0.7866058122248754, beta_2= 0.21339418777512464
PREDICT alpha_1= 0.094908207654953, alpha_2= 0.9062504768371582, beta_1= 0.7940860390663147, beta_2= 0.2094002217054367
Inference time 0.0462 seconds
----------
0.2872357923607543x^2+0.5237311903414599x+0.18903301729778582
Y_real: tf.Tensor([ 1.        -1.8626451e-08j -0.06914629-2.8985712e-01j], shape=(2,), dtype=complex64)
Y_predict: tf.Tensor([ 0.9937776 -1.8626451e-08j -0.06811628-2.8628656e-01j], shape=(2,), dtype=complex64)
REAL alpha_1= 0.3314141305127114, alpha_2= 0.6685858694872886, beta_1= 0.5703830944243201, beta_2= 0.4296169055756799
PREDICT alpha_1= 0.3331572711467743, alpha_2= 0.6633907556533813, beta_1= 0.566330075263977, beta_2= 0.4308898448944092
Inference time 0.0517 seconds
```

Further research

Make loss function independent on true values:

a=[alpha_1 1-alpha_1 0 0]

b=[0 beta_1 1-beta_1 0]

c=[A B C 0]

Delta = FFT(a) * FFT(b) – FFT(c)

Use M >= 3