Digital Finance (2022) 4:1–15
https://doi.org/10.1007/s42521-021-00041-7

ORIGINAL ARTICLE

Delta force: option pricing with differential machine learning

Magnus Grønnegaard Frandsen[1] · Tobias Cramer Pedersen[1] · Rolf Poulsen[1]

Received: 7 May 2021 / Accepted: 4 September 2021 / Published online: 4 October 2021
© The Author(s), under exclusive licence to Springer Nature Switzerland AG 2021

Abstract
We show how and why to use a financially meaningful differential regularization method when pricing options by Monte Carlo simulation, be that in polynomial regression or neural network context.

Keywords Differential machine learning · option pricing

JEL classification C63 · G13

1 A short introduction

We do two things in this paper. First, we give a pedagogical presentation of the differential machine learning method for option pricing introduced by Huge and Savine (2020). To do this we keep the modelling and pricing framework as simple as possible (Bachelier model with all-zero rates; call option) and gradually work our way up from polynomial regression (that is suitable for illustrating the “differential” part of the method) to deep dual neural networks with automatic derivative propagation. Second, we present experimental evidence for the benefits of pricing options with differential machine learning. We use the performance of discrete Delta-hedge portfolios as a general framework for out-of-sample testing, which allows us to separate the effects of the “differential” and the “machine learning” parts, and to show that both are important.

Thanks to MathFinance Conference 2021 participants.

✉ Rolf Poulsen
rolf@math.ku.dk

[1] Department of Mathematical Sciences, University of Copenhagen, DK-2100 Copenhagen Ø, Denmark

2 Polynomial regression and Delta-regularization

Throughout the paper we use the Bachelier model in a setup where the interest and dividend rates are zero. We do this primarily to keep the exposition simple.[1] Extending to non-zero rates is easy (albeit not trivial because an exponentially discounted Bachelier stock price has time-dependent volatility), as is implementing for the Black/Scholes model (see the detour at the end of this section). That is, we have a stock with risk-neutral dynamics

$$\mathrm{d}S(t) = \sigma \mathrm{d}W(t),$$

where W is a Brownian motion under the martingale measure Q. The arbitrage-free time t price of a strike K, expiry T call option is (Musiela & Rutkowski, 1997, Section 3.3)

$$\text{Call}\,(t) = \mathrm{E}_t^Q((S(T) - K)^+) = F(S(t), t), \tag{1}$$

where the function F is given by

$$F(x, t) = (x - K)\Phi\left(\frac{x - K}{\sigma\sqrt{T - t}}\right) + \sigma\sqrt{T - t}\phi\left(\frac{x - K}{\sigma\sqrt{T - t}}\right),$$

with Φ and ϕ being, respectively, the standard normal distribution and density functions.

Suppose now that we don't know the F-function in the last part of (1), but have to work with the middle part, the conditional expectation. Then we can price call options by Monte Carlo simulation: Simulate a number of outcomes of $S(T)$ (started from whatever the current stock price is), find the associated call option payoffs, and average these to get an estimate of the call option price. This, very traditional, approach (Boyle et al., 1997) ignores an important aspect: The functional relationship between the current stock price and the call option price. Taken literally: If the stock price starts at 1.01 rather than at 1.00, we have to run all new simulations. To remedy this (putting $t = 0$ without loss of generality), we can start our simulations at a range of different $S(0)$-values and then try to exploit information across outcomes.

In pseudo-code the simulations (n of them centred around `c` and with width `d`—think $\mathtt{c} = K, \mathtt{d} = 1$) could go like this (with = being used for assignment):

```
$S_j(0) = \mathtt{c} + \mathtt{d}\,\sigma\sqrt{T}\,\mathtt{N(0,1)}$ for $j = 1, \ldots, n$
$S_j(T) = S_j(0) + \sigma\sqrt{T}\,\mathtt{N(0,1)}$
$\text{Call}_j(T) = \max(S_j(T) - K, 0)$
```

[1] Over the last decade the Bachelier model has had a revival in quant circles. There are several reasons for this: It is born with a negative skew in implied volatilities, things we thought were positive have gone negative (interest rates—even oil prices), and working with constant (rather than proportional) coefficients is mathematically easier—as for instance in Jesper Andreasen and Brian Huge's work (Andreasen & Huge, 2012) on option pricing via expansions.

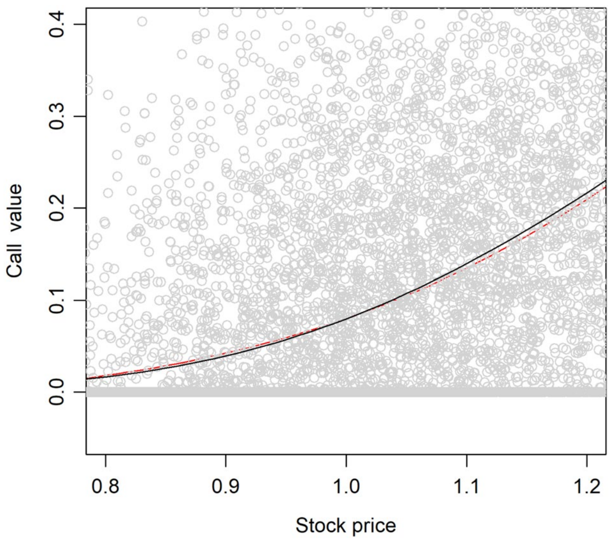

Fig. 1 Pricing an expiry $T = 1$, strike $K = 1$ call option in the Bachelier model with $\sigma = 0.2$. The scattered grey circles are (some of) 10,000 simulated (initial stock price, call option payoff)-pairs. The black curve is the true pricing function, the red curve is the estimated pricing function obtained from using a seventh degree polynomial in the regression. (Thus, for a grey dot the second coordinate is a realized time T-payoff, while for a point on the black or red curve it is an expected value of a time T-payoff conditional on the x-coordinate)

To exploit cross-sectional information we can run a least squares regression of call option payoffs on powers of the starting values. It goes like this (the last equality coming from the standard OLS formula):

$f_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_p x^p$ `[right-hand side terms: basis functions]`

crit $(\theta) = \sum_j \left(\text{Call}_j(T) - f_\theta(S_j(0))\right)^2$ `[a least squares criterion]`

$C_j = \text{Call}_j(T)$, $X_{i,j} = (S_j(T))^i$ `for` $j = 1, \dots, n, i = 0, \dots, p$

$\widehat{\theta} := \arg\min_\theta$ `crit` $(\theta) = (X^\top X)^{-1} X^\top C$

Remember that when running a regression, what we are actually doing is estimating the expected value of the left-hand side variable conditional on (i.e. as a function of) the right-hand side variables, so this is in perfect harmony with our objective.

Figure 1 shows a typical example of how this looks when using a seventh degree polynomial in the regression. The regression seems to quite accurately pick up the functional form from what looks like a "very randomly scattered" cloud of (initial stock price, call option payoff)-points.

But looks can deceive. In the left-hand panel of Fig. 2 we zoom out by plotting the polynomial regression approximation over the whole simulated range. The asymptotic behaviour of the true function is poorly captured. In the right-hand panel we have plotted (in black) the derivative of the call price wrt. the stock price (Δ), which is a quantity of particular financial relevance as we will get to in the next section, and (in red) the Δ we get from the estimated call price. The asymptotics are

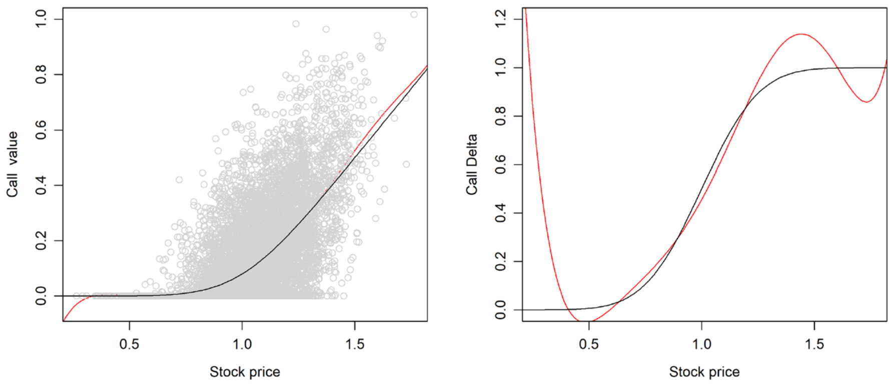

Fig. 2 Zooming out and zooming in. The left-hand panel shows the results from Fig. 1 plotted over the whole simulated range. In the right-hand panel the black curve is the true derivative of the call price wrt. the stock price (Δ), whereas the red curve is the Δ we get from the estimated call price function from Fig. 1

bad, but even in the middle of the range, differences are clearly visible. In summary: We are now less impressed by the polynomial regression method's performance.

This is where the differential part of the paper's title comes in. To explain how this works let us differentiate the left part of Eq. (1) (for $t = 0$) wrt. $S(0)$:

$$\Delta := \frac{\mathrm{d}\,\mathrm{Call}\,(0)}{\mathrm{d}S(0)} = \mathrm{E}^{Q}\left(\frac{\mathrm{d}}{\mathrm{d}S(0)}(S(T)-K)^{+}\right) = \mathrm{E}^{Q}\left(\frac{\mathrm{d}S(T)}{\mathrm{d}S(0)}\frac{\mathrm{d}}{\mathrm{d}S(T)}(S(T)-K)^{+}\right),$$

where the second equality comes from interchanging expectation and differentiation and the third from applying the chain rule.[2] The (generalized) derivative of the $(\ )^{+}$ -function is an indicator (or Heavyside) function, so the last factor inside the last expectation above is $\mathbf{1}_{S(T)\geq K}$. In the Bachelier model we can explicitly write

$$S(T) = S(0) + \sigma W(T), \tag{2}$$

making the first factor inside the expectation simply 1, so that

$$\Delta = \mathrm{E}^{Q}\left(\mathbf{1}_{S(T)\geq K}\right). \tag{3}$$

This means that by extending our simulation code by this one line

if $(S_j(T) \geq K)$ { $D_j = 1$ } else { $D_j = 0$ }

we get a bunch of simulated variables (D_j's) whose mean is the (otherwise unknown) Δ. We can include these in our least squares regression criterion,

[2] Both of the operations are what mathematicians would call formal, i.e. not rigorously justified, but let's not worry about that here; it can be fixed [(Broadie & Glasserman, 1996, Appendix A), L'Ecuyer, 1995] with the theory of generalized functions—or verified by elementary means for the Bachelier (and the Black–Scholes) model.

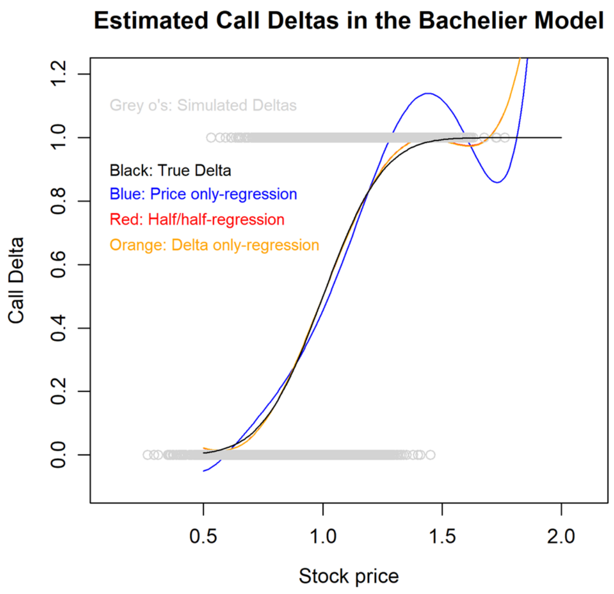

Fig. 3 Estimated Δ functions in the Fig. 1-setup when different regularizations (w's) are used

$$\text{crit}\,(\theta) = w \sum_j \left(C_j - f_\theta(S_j(0))\right)^2 + (1-w) \sum_j \left(D_j - f'_\theta(S_j(0))\right)^2, \tag{4}$$

where $f'_\theta(x) = \sum_{i=1}^p \theta_i i x^{i-1}$ and w is a weight factor that we are free to choose.[3] The introduction of the new term in Eq. (4) we will refer to as Delta-regularization. Adding regularization terms to regressions is common; for instance ridge or lasso regression (Bishop, 2006, Section 3.1.4) . But that usually comes with a trade-off of accuracy to smoothness; we must balance higher bias against lower variance. Not so here; the extra terms we introduce in the regression have the right mean.

Note that f_θ is a polynomial in x with θ_i's as coefficients, thus the parameters still enter linearly (inside the sums of squares) in Eq. (4), and it is no more difficult to minimize explicitly than the standard least squares criterion. We get

$$\hat{\theta} = (wX^\top X + (1-w)Y^\top Y)^{-1}(wX^\top C + (1-w)Y^\top D), \tag{5}$$

where X and C are as before and $Y_{i,j} = i(S_j(0))^{i-1}$.[4]

[3] In the recurrent example in this paper one can safely use $w = 1/2$ as prices and Deltas are of the same magnitude. In general we suggest using $w = sd(D)/(sd(C) + sd(D))$, where sd denotes standard deviation. This ensures that terms in the "naked" version of Eq. (4)—i.e. when $\theta = 0$—have the same variance.

[4] To make dimensions fit snugly, Y starts with a zero-column, so strictly speaking Eq. (5) can't be used for $w = 0$.

Figure 3 shows the results (for Δ) of applying different regularization schemes (i.e. using different values of w). A polynomial price approximation gives a polynomial Δ approximation, so there is a limit to how well asymptotics can be fitted, but we see that the schemes that use Delta-regularization in some form (the exact choice of w isn't terribly important) are considerably more well-behaved.

A detour on pathwise derivatives. Equation (3) is a false friend. The formula for Δ looks model independent, but we must stress that it does hinge strongly on the explicit structure of the Bachelier model as given in Eq. (2). The idea, though, is not stillborn. In the Black–Scholes model S is a geometric Brownian motion, $S(T) = S(0)\exp\left(-\frac{1}{2}\sigma^2 T + \sigma W(T)\right)$, so that

$$\frac{\mathrm{d}S(T)}{\mathrm{d}S(0)} = e^{-\frac{1}{2}\sigma^2 T + \sigma W(T)} = \frac{S(T)}{S(0)}$$

and $\Delta^{B-S} = \mathrm{E}^Q(S(T)\mathbf{1}_{S(T)\geq K}/S(0))$. If we are in a more complicated model, one that we can only simulate via time-discretization (for instance using the Euler scheme) then the chain rule can be applied locally in time and we get simulated Δs in one sweep going forward. This idea of pathwise differentiation was—to our knowledge—first suggested by Broadie and Glasserman (1996) and it plays a strong supporting role in Mike Giles and Paul Glasserman's seminal work (Giles & Glasserman, 2006) on adjoint differentiation.

3 Discrete Delta-hedge performance: a general out-of-sample test

Figure 3 looks nice, sure. But do the differences really matter? We have seen before that looks can deceive. To investigate this, we turn to the fundamental financial concept of hedging: Having sold an option we want to construct (and dynamically adjust, if needs must—which they will) a portfolio that offsets (or hedges) the risks stemming from our short position. In a complete model (such as the Bachelier or the Black–Scholes model) any simple claim (something whose time T payoff depends only on $S(T)$; for instance a call option) can be perfectly hedged by a portfolio that holds $\Delta(S_t, t)$ units of the stock at any time t and is kept self-financing by trading in the risk-free asset, where Δ denotes the derivative of the pricing function for the particular claim (F from before) wrt. the stock price (Björk, 2020, Theorem 8.5).

A simulated, discrete ($dt = T/M$) version of the hedging process along a stock path looks (once we have set the hedging in motion) like this

```
for $t =$ dt, 2dt, ..., (M-1) dt {
$S_t = S_t + \sigma\sqrt{dt}N(0,1)$
$V_t = a_t S_t + b_t$
$a_t = \Delta(S_t, t)$
$b_t = V_t - a_t S_t$ }
```

Here, a_t is our stock position, V_t is the value of our hedge portfolio, and the position b_t in the risk-free asset is chosen such that the strategy is self-financing. (We repeatedly overwrite old values.) The payoff of our hedged portfolio (the hedge error, the profit-and-loss, PnL) is $V_T - \max(S_T - K, 0)$ where

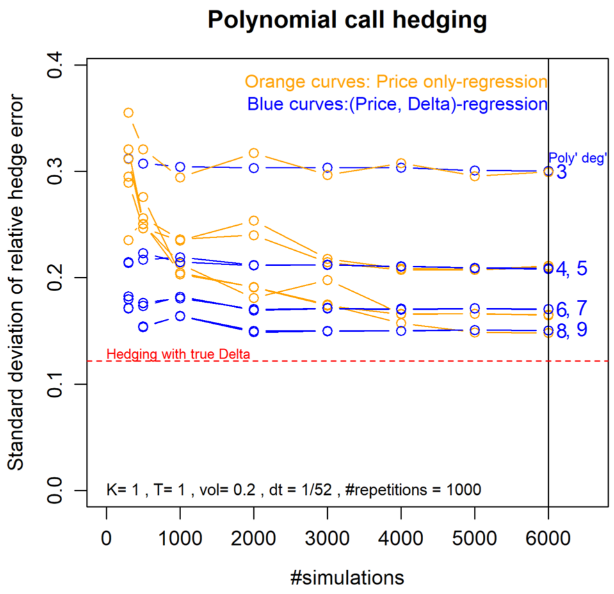

Fig. 4 Hedge error standard deviation for different polynomial regressions

$V_T = a_{T-dt}S_T + b_{T-dt}$. We can repeat the simulation over many paths and analyze the hedge error characteristics, for instance its standard error – which we (after dividing it by the initial price of the hedge portfolio) will refer to simply as “the hedge error” in the following. Note that if time were continuous, this hedge strategy would work perfectly for each and every simulated path, not just on average or in some central limit theorem sense.

This gives us a general framework for out-of-sample investigations of option price approximations (that could for instance come from the polynomial regression setup in the previous section): Run the hedge experiment where the Δ in the next-to-last line is the one the approximation under investigation provides us with.

For the polynomial regression case, we pre-compute (or pre-estimate) the regression coefficients; this has to be done for each relevant time-to-expiry. This gives us approximations that we can investigate along three dimensions: The number of simulations used in the regressions (note that this is different from the number of times we repeat the hedging along simulated paths), the degree of the polynomial used in the regression, and the extent of Delta-regularization (which we treat as “yes or no”).

The results of such an experiment are shown in Fig. 4. We see that for a fixed polynomial degree, p, the hedge errors for both regularized and unregularized pricing functions converge (as the number of simulations in the regressions is increased) to the same non-zero number (the right-hand part of the graph). This number is non-zero for two reasons. First, the discreteness of time. To control for this, the thin red line in the graph shows the hedge error we get when we use the Bachelier model’s true $\Delta = \Phi\left(\frac{S_t - K}{\sigma\sqrt{T-t}}\right)$. This is the best performance we can hope for with strategies using Δ-approximations.[5] Second, the finiteness of p means that the true function we are after (Δ—which is not a polynomial) is not spanned by our basis functions, meaning that no matter how many simulations we use, we are never going to get it quite right. But even though the limiting behaviours of the hedge errors are the same, the small sample behaviours are distinctly different depending on whether or not we regularize. With regularization (which costs little to nothing computation time-wise) the hedge errors stabilize considerably quicker (for 1000-2000 rather than 6000 simulations—a dimension in which computation time grows linearly); clearly a desirable feature. We also see that the higher p is (i.e. the more flexible our class of basis function is—in theory), the bigger the regularized/unregularized difference in time (or rather number of simulations) needed for stabilization is. Finally, note that the general hedge error level depends quite strongly on p, e.g. using a fourth degree rather than an eighth degree polynomial scales hedge errors up by a factor of about 1.4. From a theoretical point, that makes perfect sense. However, we were still a little surprised, because in the Longstaff and Schwartz method for American option pricing (Longstaff & Schwartz, 2001), which is also a polynomial regression framework, albeit one not exactly similar to ours, results typically don't depend very much on the degree of the basis function polynomials; anything beyond fourth order is overkill. The reason is that in the Longstaff and Schwartz setting, the polynomials estimate the continuation values, but these are used only to make the "exercise or hold" decisions; the proxies enter only via indicator functions and therefore only the local behaviour around the exercise boundary matters. In our setting the polynomials are used for global approximations, exacerbating differences across p's.

4 Deep neural networks with (and without) Delta-regularization

On the recurrent theme of deceiving looks, a natural question is: "Why don't you show the results for 10th, 11th, $\cdots$ degree polynomials in Fig. 4? It seems particularly relevant because you say the hedge quality depends strongly on the degree." The answer is simple: Because it does not work! When going beyond polynomials of degree nine, our code crashes; regression matrices become (numerically) singular. There are one or two hacks for dealing with this,[6] but they do not address the fundamental problem: No "natural" high-degree polynomial looks like the smooth hockey stick function we are trying to approximate (see Fig. 2); flat at 0 at one end, asymptotically affine at the other. We need a better space of basis functions. The quest for this will lead us over non-linear regression models to machine learning in the form of neural networks.

[5] For a non-infinitesimal hedge frequency, say $dt = 1/52$, we cannot say that the discrete use of the continuous Δ gives the lowest hedge error standard deviation. In fact, as shown by Wilmott (1994) we can do slightly better when rates are non-zero. However, (a) we have results that ensure dt $\rightarrow$ 0-convergence and (b) rates are zero here, so we ignore that.

[6] We could increase the range parameter d or change to an estimation method that does not explicitly invert matrices.

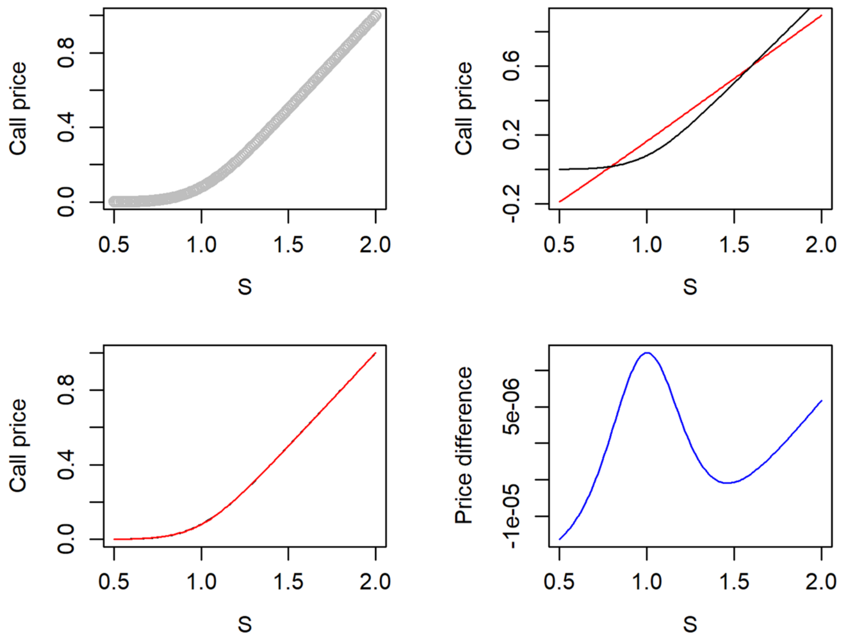

Fig. 5 SoftPlus fits. The top left panel shows call option prices (true, exact ones from the Bachelier model with our usual parameter settings). In the top right panel, the red curve is the SoftPlus fit (to the black curve) obtained with a standard gradient-free optimization method. The bottom left panel shows the similar fit when using the BFGS-method with bump-and-revalue gradient calculation. (The difference between the black and the red curves is not visible to the naked eye.) The bottom right panel (note the small values on the y-axis) shows the difference between BFGS-fitting with bump-and-revalue and analytical gradients

A function that is born with a smooth hockey stick form is SoftPlus,

$$\text{SoftPlus}\,(x) = \ln(1 + e^x).$$

For “very negative” x’s, e^x vanishes and SoftPlus returns 0, for large positive x’s, $e^x >> 1$ and SoftPlus gives us x back. A natural way to turn SoftPlus into a parameterized basis function for regression is as

$$f_\theta(x) = \theta_0 + \theta_1\ \text{SoftPlus}\,(\theta_3 x + \theta_4). \tag{6}$$

The abstract least squares problem in Eq. (4) looks the same, but there is a significant difference: The criterion function is no longer linear (or to be precise: quadratic) in the parameter vector, so we have no OLS-like closed-form expression for the optimal value (which would then be our estimate). We have to minimize numerically. “Yes, well, ok, not linear, but in no way a nasty function; how poorly can that go?” Here—in as in many other situations in life—the answer is “Quite poorly—if you are not careful”. To demonstrate this let us for a moment discard the simulation aspect. Input variables are just a deterministic range of S-values and output values are the true Bachelier call option prices. It could look like the situation in Fig. 5

where the input range is 150 equidistant points on the x-axis and output values are corresponding points in the top left panel. Suppose we throw this partially non-linear regression problem to a generic optimizer, which could look something like this:

```
estimation = optim(start = c(0,0,0,0), f=Criterion)
```

That produces the red curve in the top right panel in Fig. 5 as the approximation to the black curve. Not very accurate.

Instead we can use a gradient-based optimization method, e.g. Broyden–Fletcher–Goldfarb–Shanno. That goes something like this

```
estimation = optim(start = c(0,0,0,0),f=Criterion,
gr=Grad, method = "BFGS").
```

where we then have to implement a function (here called `Grad`) that calculates the gradient and feed it to the optimizer. That works wonders; we have done it in the bottom left panel of Fig. 5 and the difference between the red and the black curves there is invisible to the naked eye (which is why the curve looks red, as that was the one drawn last). So: using the gradient is crucial for optimization.

However, exactly how (in particular, how accurately) the gradient is calculated may not be important for the optimization to work. In fact, to produce the fit in the bottom left panel, we simply did a lazy numerical approximation based on simple finite difference bump-and-revalue of the criterion function. We also did a more careful implementation of the analytical gradient. That had absolutely no practical effect on the parameter estimate and the quality of the fit—as seen in the bottom right panel in Fig. 5.

Surprisingly, the major benefit of using analytical rather than bump-and-revalue methods for gradient calculations can lie in speed, not accuracy. Let us use the case at hand to indicate why. The key part is calculating the derivatives of f_θ wrt. the p (+1 if you are being pedantic about zero-offset) coordinates of θ, i.e. the function ∇f_θ taking values in $\mathbb{R}^p$, so we will focus on that.[7] Symmetric second-order bump-and-revalue-calculation here requires $2 * 4 = 8$ function calls (of SoftPlus); calculation time is $O(p)$ we could say. The analytical gradient is (by the chain rule)

$$\nabla f_\theta(x) = (1,\ \text{SoftPlus}\,(\theta_3 x + \theta_4), \theta_2\theta_3\ \text{SoftPlus}\,'(\theta_3 x + \theta_4), \theta_2\ \text{SoftPlus}\,'(\theta_3 x + \theta_4))^\top,$$

where $\text{SoftPlus}\,'(x) = 1/(1 + \exp(-x))$. This requires only three calls to external functions—of roughly the same computational complexity (SoftPlus $'$ may even be slightly faster than SoftPlus; exp and division vs. exp and log). But we can do better: The SoftPlus $'$ function is evaluated at the same point for both the 3rd and the 4th coordinate, so we really only need two external function calls. Moreover: If inside SoftPlus in f_θ we had chosen to use a polynomial in x with θ_i's as coefficients, we would still only need two external function calls to calculate the gradient—independently of the dimension of the parameter vector; wrt. p computation time is $O(1)$. This is quite counterintuitive given the basic construction of derivatives. It may seem that we are reading a lot into being able to skip one function call in a fairly

[7] Handling the square and particularly the sum part of the criterion function efficiently does require thought – but of a different (vectorization) nature.

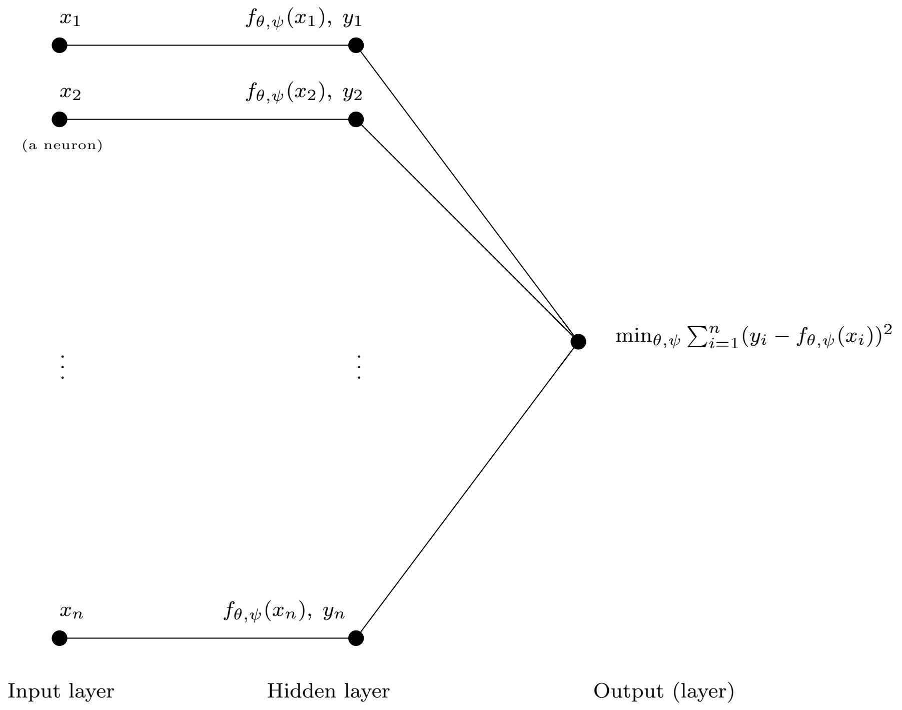
x_1
$f_{\theta,\psi}(x_1),\ y_1$
x_2
$f_{\theta,\psi}(x_2),\ y_2$
(a neuron)
$\min_{\theta,\psi}\sum_{i=1}^n(y_i - f_{\theta,\psi}(x_i))^2$
x_n
$f_{\theta,\psi}(x_n),\ y_n$
Input layer
Hidden layer
Output (layer)

Fig. 6 A partially non-linear regression problem (without Delta-regularization) formulated as a neural network with one hidden layer. S_0-values play the role of the x_i's; call payoffs play the role of the y_i's

simple setting. But there are bigger things in store. One comes from noting that the order conclusion still holds when we revert back to the situation where the output variables are simulated payoffs. And it holds when we add a differential regularization term to the least squares criterion (where we need the second derivative of SoftPlus, but that is just $\frac{e^{-x}}{(1+e^{-x})^2}$). Moreover, the ideas extend (with increasingly careful use of the chain rule) to more general classes of functions defined (recursively) via composition. This is the fundamental driving force behind the backpropagation algorithm, which is arguably the reason why machine learning works.

The parametrization in Eq. (6) works well in this example because the basis function chosen already strongly resembles the Bachelier pricing function. However, it is too simple to represent more complicated functions. A natural way to extend the SoftPlus universe is to introduce functions of the form

$$g_{\psi^j}(x) = \text{ SoftPlus } (\psi_0^j + \psi_1^j x), \quad j = 1, \dots, n,$$

and then use

$$f_{\theta,\psi}(x) = \theta_0 + \theta_1 g_{\psi^1}(x) + \dots + \theta_n g_{\psi^n}(x). \tag{7}$$

The g_{ψ^i}'s are called learned basis functions. 'Learned' because they depend on parameters that we are about to estimate – or learn in machine learning parlance. We use ψ to notationally indicate the parameters in which f is non-linear. The function $f_{\theta,\psi}$ can be interpreted as a *shallow* neural network as shown in Fig. 6. In machine learning lingo, this neural network has one hidden layer with n units and SoftPlus as its activation function. From the Universal Approximation Theorem (Leshno et al., 1993), we know that $f_{\theta,\psi}$ can represent any continuous function if n is large enough.

The shallow neural network $f_{\theta,\psi}$ takes a one-dimensional input x, expands it to a n-dimensional hidden layer through the g_{ψ^i}'s, and subsequently transforms it into a one-dimensional output. A generalisation of the shallow neural network consists of introducing *multiple* hidden layers. That is, the input x is altered through a sequence of transformations,

$$\mathbb{R}^{n_0} \to \mathbb{R}^{n_1} \to \mathbb{R}^{n_2} \to \cdots \to \mathbb{R}^{n_m}$$

where $n_0 = n_m = 1$ in our case. Each transition from $\mathbb{R}^{n_i}$ to $\mathbb{R}^{n_{i+1}}$ for $i \in \{0, \dots, m-1\}$ is achieved through a transformation

$$h_{i+1}(z) = \sigma_i(b_i + A_i z) \in \mathbb{R}^{n_{i+1}} \tag{8}$$

where $z \in \mathbb{R}^{n_i}$, $A_i \in \mathbb{R}^{n_{i+1} \times n_i}$, $b_i \in \mathbb{R}^{n_{i+1}}$ and σ_i is a so-called activation function, which is applied elementwise. A *deep* neural network is then defined as

$$f_\theta(x) = h_m \circ h_{m-1} \circ \cdots \circ h_1(x), \tag{9}$$

where $\circ$ denotes composition and the θ-notation now encompasses the learnable parameters A_i and b_i for all i. We say that the network has one input layer, $m-1$ hidden layers and one output layer. The entries of the A_i's are often called weights, the elements of the b_i's biases. The left dimension of A_i is referred to as the i'th layer's number of units. For completeness, we notice that the construction of the shallow neural network from Eq. (7) is a special case of a deep neural network as described in Eqs. (8) and (9).

A natural question is: "Why do you start composing functions in Eq. (9), why don't you just add more terms on the right-hand side of Eq. (7)?" Here, we know little of "hard math" theorem/proof results. But let us give two partial explanations (besides the simple "because it can be done" one). Intuitively, function composition gives greater flexibility than function addition (this can be made more rigorous Telgarsky, 2015). Experimentally/empirically, deep networks have proven superior to shallow (i.e. single layer) networks in their ability to act as universal approximators when the number of parameters is fixed (the representation in the Universal Approximation Theorem is only guaranteed in the limit).

To include a differential regularization term in a deep neural network, we must determine the x-derivative, $f'_\theta(x)$. By successive application of the chain rule,[8] we obtain

[8] Here we are skipping a bunch of nasty details regarding matrix multiplication, differentiation and restrictions on input and output dimensions.

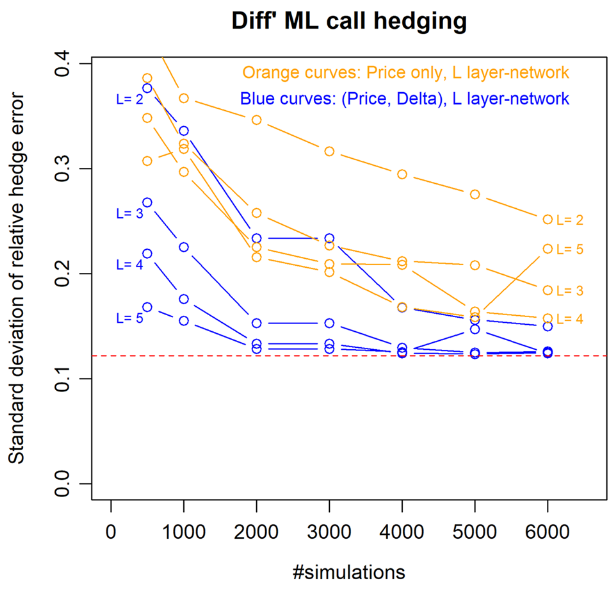

Fig. 7 Hedge errors for call price approximations based on multi-layer neural networks with SoftPlus basis functions. Each of the networks involved (52 for any point on any curve) takes about 1 s to train

$$f'_\theta(x) = h'_m \circ h_{m-1} \circ \cdots \circ h_1(x) \cdot h'_{m-1} \circ h_{m-1} \circ \cdots \circ h_1(x) \cdot \cdots \cdot h'_1(x),$$

where $h'_i(x) = \sigma'_i(b_i + A_i x) \otimes A_i$, where $\otimes$ denotes elementwise multiplication. For efficient calculation of f'_θ using the above representation, we should first evaluate $f_\theta(x)$ and store all intermediate values, which can then be utilized to calculate f'_θ. This is in its essence backpropagation.

With f_θ and f'_θ available, we return to our ultimate goal of minimising Eq. (4). The realm of analytical formulas was left some pages ago, so we must minimise numerically—using gradient descent, as per our previous insight. To do so, we must compute the gradient of Eq. (4) with respect to the parameter vector θ. Now, both the computation of f_θ and of f'_θ consist solely of compositions of simple functions, so such a differentiation is fundamentally an exercise in careful application of the chain rule, where all the non-linearity comes through the activation function, whose derivative we know analytically. This is the generalised backpropagation algorithm known as automatic adjoint differentiation, and handling it is the cornerstone of advanced computational libraries such as Tensorflow and PyTorch for Python.

To recap: Given an input x data vector and parameter vector θ, the order of operations is as follows. First, the approximation of the price $f_\theta(x)$ is computed. Second, a subsequent backpropagation with respect to x is conducted to compute $f'_\theta(x)$—our approximation of Δ. Third, the two approximations are combined into our criterion function from Eq. (4). Fourth, the entire expression (sometimes called a dual network) is backpropagated with respect to θ. Fifth, the θ-gradient is used to compute a step of a gradient descent scheme to minimise the criterion function, thus updating the θ-vector. The whole procedure is repeated until a suitable numerical convergence criterion is met.

In Fig. 7 we show hedge performance in an experiment that is completely similar to the one from Fig. 4 except the Δs come from different multi-layer SoftPlus neural networks with and without Delta-regularization. The unregularized networks take long to stabilize and increasing the depth of the network (the number of layers) does nothing to alleviate that. In fact (compare to Fig. 4), without regularization the performance of the networks is no better than that of simple polynomial regressions. For the Delta-regularized networks it is a different story. Hedge performance stabilizes quickly (as a function of the number of simulations) and improves markedly with the depth of the network. Specifically, regularized networks with four or five layers achieve the theoretically optimal performance bound that eluded us in the polynomial regression setting.

It is often the case that what is gained in numerical precision, is lost in computational speed. The important question is whether or not the trade-off is favourable. While the polynomial approximations are essentially instant to compute, the neural networks require time to train. The overarching theme is unsurprisingly that sample size and network complexity increases training time. The four layer network with Δ regularisation at 4000 training samples is arguably the first to reach the performance of the true Δ. A single of these networks requires 1.64 s to train in our Python implementation. The equivalent four layer network without Delta-regularisation requires 1.38 s of training, but at an clear cost to precision. Averaging across all sample sizes and depths, the Delta-regularised networks are approximately 36% slower to train. While perhaps daunting in relative terms, the largest difference in absolute terms of 0.92 s belongs to the 5 layer networks with 5000 training samples. That difference is certainly excusable when compared with the relative performance.

5 Conclusion

We have demonstrated the benefits of Delta-regularization in option price simulation; first for polynomial regression, second in a deep (dual) neural network setting. The more flexible your class of basis functions is, the more important regularization is.

Based on the results in this paper, using polynomial regression with, say, an eight degree polynomial and Delta-regularizaion might seem reasonable, when taking into account ease of implementation and computation speed, and weighing it against the hedge error gap in Fig. 4. However, polynomial regression scales badly with the dimension of the underlying risk factors. e.g. settings with multiple underlyings or models with stochastic volatility. Because of the cross-terms, the number of basis functions (and hence parameters) is essentially the power of the polynomial raised to the dimension af the state variable; e.g. $8^1 = 8$ vs. $8^2 = 64$. Contrarily, the number of parameters grows only linearly with the dimension of the input variable in the neural networks, we "just" need an A-matrix in Eq. (8) with columns for the new state variables. However, we must stress that the word "just" in the previous sentence belongs in inverted commas. The practical implementation of networks with multi-dimensional state-variables is a considerable leap from what we have described in this paper, however a wealth of information (including notebooks with Python code that can be run in Google Colab) is provided for all and sundry by Brian Huge and Antoine Savine at https://github.com/differential-machine-learning.

References

Andreasen, J., & Huge, Brian. (2012). ZABR—Expansions for the Masses. Working paper at http://ssrn.com.

Bishop, C. (2006). *Pattern recognition and machine learning* (2nd ed.). Springer.

Björk, T. (2020). *Arbitrage theory in continuous time* (4th ed.). Oxford.

Boyle, P., Broadie, M., & Glasserman, P. (1997). Monte Carlo method for security pricing. *Journal of Economic Dynamics and Control, 21,* 1267–1321.

Broadie, Mark, & Glasserman, Paul. (1996). Estimating security price derivatives using simulation. *Management Science, 42*(2), 269–285.

Giles, M., & Glasserman, P. (2006). Smoking adjoints: Fast Monte Carlo Greeks. *Risk*. p. 88–92

Huge, B., & Savine, A. (2020). Differential machine learning: The shape of things to come. *Risk*. p. 76–81

L'Ecuyer, P. (1995). On the interchange of derivative and expectation for likelihood ratio derivative estimators. *Management Science, 41*(4), 738–748.

Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a non-polynomial activation function can approximate any function. *Neural networks, 6*(6), 861–867.

Longstaff, F. A., & Schwartz, E. S. (2001). Valuing American options by simulation: A simple least-squares approach. *Review of Financial Studies, 14*(1), 113–147.

Musiela, M., & Rutkowski, M. (1997). *Martingale methods in financial modelling* (2nd ed.). Springer.

Telgarsky, M. (2015). Representation Benefits of deep feed forward networks. At arXiv arxiv:1509.08101

Wilmott, P. (1994). Discrete charms. *Risk*. p. 48–51

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.