Neural Networks for Derivatives Modelling

Candidate No:1062797

University of Oxford

A thesis submitted in partial fulfillment of the MSc in

Mathematical and Computational Finance

June 28, 2022

Abstract

[INCOMPLETE]

Neural Networks are beneficial in terms of pricing approximators as: inference in price is fast, first order and second order differentials can be obtained relatively fast. Furthermore, Neural Networks may be able to overcome the curse of dimensionality and be applied to complex pricing problems, where other numerical methods may fail.

Contents

1 Motivation

Consider a typical workflow for a end-user in an investment bank or market maker: A end-user uses a given pricing function to obtain the price(s) for the a given derivatives product(s) and parameter(s), and then use the outputs of the pricing function to inform some decision or action. Under-the-hood, the pricing function is invoked, which: 1) takes in the collection of input parameters: the current values of the given asset(s), the payoff structure of the product(s), and parameters for the *market generator* model (volatility dynamics, or modelling assumptions of the underlying assets), 2) invokes some underlying numerical method, and finally 3) produces the outputs, which are typically the price of the derivative and the *sensitivities* of the price to the input parameters. For example, a trader may need prices in order to decide whether to trade against a market quote, or may potentially use the sensitivities for a hedging strategy. On the other hand, a risk manager may need to obtain prices under various numerous scenarios to model the potential losses of a portfolio (VaR), or to calculate XVA adjustments.

	Electronic / Flow	**Exotics**	**Risk Modelling**	**Hedging**	**Calibration**
Pricing Accuracy	High	Medium	Medium	-	High
Gradient Accuracy	High	High	High	High	-
No-Arb	High	Medium	Medium	-	High
Speed	Milliseconds to Daily	Minutely to Daily	Daily	Milliseconds to Daily	Milliseconds to Daily

Table 1: Some applications for pricing functions and potential requirements

Arguably, a key business objective for derivatives quants in these financial institutions, is to improve upon the precision and latency of the pricing functions. To accomplish this, derivatives quants generally leverage and develop the three families of numerical methods for derivatives pricing: analytic expressions, Monte Carlo (MC) Methods, and Partial Differential Equation (PDE) solvers.

Analytic expressions allow for fast 'true' prices, but are only available for few market models and payoffs (e.g. the Black-Scholes formula and the Heston characteristic transform). For many payoffs and market models, this leaves MC and PDE methods, which are themselves approximation methods. In the case of MC and PDE methods, approximation error to the 'true' model price can be reduced with greater computational effort at a known convergence rate - this allows for a controlled tradeoff between a requisite level of error and time / computational effort.

However, several aspects may lead to an increase in the total computational effort required in a pricing function. Firstly, in a bank, not only the prices are needed but also the first-order sensitivities, and in some cases the second-order sensitivities [46]. This is given that the sensitivities represent exposures to the input parameters, and also in some cases, the potential hedging strategy (delta). In some cases, higher order sensitivities need to be hedged or accounted for as well (e.g. option gamma). As the *risk* sensitivities also need to be computed (through some other numerical procedure), the computational effort may grow even larger. Secondly, in the Monte Carlo and PDE settings, a solution can be obtained over the state and time discretisation, but if the volatility model or payoff parameters change, the numerical method must be reinvoked (parametric pricing, or pricing *families* of derivatives). This is particularly relevant for *calibration*, in which the undetermined volatility model parameters must be obtained through non-linear optimisation necessitating many invocations of the pricing function. Finally, the computational effort may also grow larger when the complexity of the (*exotic*) derivative product increases, such as products with a high dimensionality in the number of underlying assets (e.g. Basket options), or payoffs with more complex features such as callability (e.g. Bermudan swaptions).

One-way to improve speed may be to consider approximation methods, for example using *machine learning* - generate accurate prices using the potentially expensive methods of MC / PDE, and then learn some functional mapping from the input parameters to the prices with a fast inference time. However, unlike standard approximation or prediction problems in machine learning, in derivatives modelling an approximating function may need to satisfy certain properties. In addition to being sufficiently smooth and differentiable to obtain sensitivities, another key requisite for any pricing function is adherence to no-arbitrage constraints. If violated, a consequence is that the pricing approximation could produce *static* arbitrage opportunities in its prices, which could present a material possibility of a loss if the arbitrage can be exploited, (e.g. the produced prices are quoted, and the market is very liquid).

Towards some practical applications, there may be acceptable tradeoffs between the criterion of interest: pricing error, gradient errors, no-arbitrage adherence, speed, and performance on the downstream application. Different end-users and applications may have different needs in terms of latency and the various error metrics, as depicted in 1. In the context of liquid, flow, or electronically traded products, a high degree of pricing accuracy and adherence to no-arbitrage is needed, as well as accuracy in the gradients as well, if the model hedge is used to hedge (we note that the electronic setting may amount to very fast and repeated calibration, pricing with some bid-ask spread, and then hedging). For exotic derivatives, pricing accuracy may be important, but some small error in pricing may be acceptable as mispricings may not be able to be well exploited for illiquid products. However, accurate gradients may be needed in this case for use as the hedging strategy, and to accurately inform a trader of key risks. In the case of risk modelling, some pricing error may be acceptable, but an increase in speed may be desired.

Neural networks may be one approximation method that allows for a tradeoff between these errors, with additional desirable properties. Neural networks have theoretically, universal approximation, and empirically, an ability to 'learn' non-parametric data-driven relationships. Sensitivities can be obtained quickly with automatic differentiation, and an overall very fast inference time can be potentially obtained. Finally, neural networks can potentially overcome the *curse-of-dimensionality*, and can serve as a model agnostic (in the sense of volatility or market generator models) and method agnostic (MC, PDE) extension to existing numeric pricing methods. A vast literature on the application of neural networks toward derivatives modelling has emerged. In terms of industrial applications, [19] of Risk-Fuel / Scotiabank, and [31] of Danske Bank, respectively, used neural network approximations of pricing functions to speed up XVA calculations, and [6] of JP Morgan proposed using neural networks to learn hedging strategies, and applied this towards automated hedging of vanilla equity options.

Dissertation Aim and Structure: One key concern, is that an arbitrary neural network construction method may not necessarily adhere to no-arbitrage and smoothness conditions. Hence the aim of this dissertation is to explore the various contexts, and evaluate neural networks for derivatives modelling:

- Under different construction and training methods
- Under various volatility models and payoffs,
- Under several criterion: pricing error, error in gradients, no-arbitrage adherence, latency, and performance on a given downstream task.

This dissertation is structured as follows: Chapter 2 consists of the relevant mathematical formulations for derivatives pricing: pricing as a conditional expectation, PDE, various methods to obtain sensitivities (Finite Differences, AAD), and the principles of no-arbitrage. Chapter 3 is a review of neural networks from a machine learning perspective, and why their properties may make them suitable as pricing function approximations. Chapter 4 is a review of the relevant literature on applying neural networks for pricing and sensitivities approximation, and how different constructions can incorporate the no-arbitrage pricing principles from Chapter 2. In Chapter 5, we evaluate the various neural network construction methodsfrom Chapter 4, and examine their behaviour in various numerous settings. Finally, in Chapter 6 we review the overall discussion and consider some potential next steps for exploration.

2 Preliminaries

In this section, we review the formulation of derivatives pricing and risk calculations.

[INCOMPLETE, tidy this section to make much more coherent]

Problem Setup

Consider the problem of computing derivative prices and sensitivities with respect to some input parameters $\mathbf{X} \in \mathcal{X}$. First, consider a filtered probability space $(\Omega, \mathcal{F}, \mathcal{F}_t, \mathbb{P})$. Suppose there is a $\mathcal{F}_t$-adapted d-dimensional stochastic process $\mathbf{S}_t \in \mathbb{R}^d, t \in [0, T]$, representing the value of the underlying assets, or cash flows, through. For simplicity, assume that $\mathbf{S}_t$ is a Ito diffusion, although we could also consider more genera ally Levy or other processes. We write $\mathbf{S}_t$ in place of $\mathbf{S}_t(\mathbf{X})$, although the evolution of the stochastic process $\mathbf{S}_t$ may depends on some of the parameters in $\mathbf{X}$

$$\begin{aligned}
\alpha_0, \nu, F_0, S_0 > 0, \rho \in [-1,1], \beta \in [0,1] \\
\mathbf{S_t}(\mathbf{X}) = (F_t, \alpha_t), \mathbf{X} = (\alpha_0, F_0, \beta, \rho, \nu) \\
dF_t(\beta, F_0, \rho) = \alpha_t F_t^{\beta}[\rho dW_t + \sqrt{1-\rho^2} W_t^{\top}] \\
d\alpha_t(\nu, \alpha_0) = \nu \alpha_t dW_t, \quad d[W, W^{\top}]_t = 0
\end{aligned} \tag{SABR as a Parametric SDE}$$

The choice of dynamics for the stochastic process $\mathbf{S}_t$, which we will also refer to as the choice of *volatility* or *market* model, represents the quant's assumptions for the dynamics of the underlying assets. Now suppose that there exists some equivalent local martingale measure $\mathbb{Q} \sim \mathbb{P}$, such that $\mathbf{S}_t$ is a $\mathbb{Q}$-local martingale (alternatively, let $\mathbf{S}_t$ be a $\mathbb{Q}$-local martingale under some change of numeraire). Then in the most general case, a European payoff is characterised by a measurable payoff function $h : \mathcal{F} \to \mathbb{R}$ on the underlying value of the stochastic process $\mathbf{S}_T$, evaluated at time $t + \tau = T$.

Pricing as Conditional Expectation

European payoffs: A European option is given by

$$f(\tau, \mathbf{S_t}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[h(\mathbf{S_T})|\mathcal{F}_T\right] = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_T] = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_t, \tau, \mathbf{X}] \tag{1}$$

In Equation 1, we assume that $\mathbf{S}_T$ is a Markov Process, such that we can write $\mathbb{E}^{\mathbb{Q}}[\cdot|\mathcal{F}_t] = \mathbb{E}^{\mathbb{Q}}[\cdot|\mathbf{S}_t]$. In addition, the stochastic process $\mathbf{S}_t$ is conditionally independent of the fixed parameters $\tau, \mathbf{X}$, or that $\mathbf{S}_t$ is one of the parameters in $\mathbf{X}_t$, we obtain the second equality. Thus the true pricing function f is equivalent be expressed as a conditional expectation of the payoff $h(\mathbf{S}_T)$ on $\mathcal{F}_t$ under some risk-neutral $\mathbb{Q}$, supposing that the conditional expectation is bounded:

$$\mathbb{E}^{\mathbb{Q}}\left[|h(\mathbf{S}_T)|\mathbf{S}_t, \mathbf{X}|\right] < \infty$$

In 1 we decouple, the time to maturity, underlying, and parameters $\tau, \mathbf{S}_t, \mathbf{X}$. We include the time-to-maturity τ into the pricing function f, as although it does not influence, it is a parameter that determines the payoff structure. More generally, the payoff function h may also have parameters involved, for example the strike K for vanilla European Call options. Thus in general, a pricing function has three components: the parameters of the volatility model, the parameters of the payoff structure, and the values of the underlyings. As an aside, we also require τ to be an input, for the purpose the Feynman-Kac formula, and to obtain the corresponding sensitivity when using a neural network approximation.

Hence if we can evaluate the conditional expectation for a given $\mathbf{X}$, we obtain the pricing function for the same $\mathbf{X}$. This motivates the use of Monte Carlo methods, or equivalently numerical integration.

$$\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_t, \mathbf{X}] = \int h(\mathbf{S}_T)p(\mathbf{S}_T; \mathbf{S}_t, \mathbf{X})d\mathbf{S} \tag{2}$$

Example: European Calls, 1 Asset. For example, in the case of vanilla European calls in the Black-Scholes setting, the parameters $\mathbf{X}$ consist of the contract parameters, strike and time-to-maturity K, τ, volatility model parameters σ, and underlying values S_t

$$f(\mathbf{X}) = \mathbb{E}^{\mathbb{Q}}[(S_T - K)^+|\mathbf{X}], \mathbf{X} = (\tau, K, S_t, \sigma), S_t \in \mathbb{R} \tag{3}$$

Pricing as the solution to the PDE / Stochastic Control

Now further suppose that f is sufficiently smooth, in particular once differentiable C^1 with respect to time-to-maturity $\tau = T - t$ and twice differentiable C^2 with respect to each $\mathbf{S}_t$. If we consider $f(\tau, \mathbf{S}_t(\mathbf{X})) = Y_t$ as a stochastic process, and apply Ito's lemma:

$$d\mathbf{S}_t = \boldsymbol{\sigma}(t, \mathbf{S}_t)d\mathbf{W}_t, \mathbf{W}_t \in \mathbb{R}^d, \sigma(t, \mathbf{S}_t) \in \mathbb{R}^{d\times d} \tag{4}$$

$$dY_t = d(f(\tau, \mathbf{S}_t(\mathbf{X}))) = -\frac{\partial f}{\partial \tau}dt + \sum_{i=1}^{d} \frac{\partial f}{\partial S_i}dS_{i,t} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} \frac{\partial^2 f}{\partial S_i \partial S_j}d[S_i, S_j]_t \tag{5}$$

$$= -\frac{\partial f}{\partial \tau}dt + \nabla_{\mathbf{S}} \cdot \sigma(t, \mathbf{S}_t)dW_t + \frac{1}{2}Tr(\sigma^\top H_f \sigma)dt, \quad Y_T = h(\mathbf{S}_T) \tag{6}$$

In order for Y_t to be an arbitrage-free price, Y_t must be a $\mathbb{Q}$-local martingale. Hence the drift of 6 must be zero. Then by the Feynman-Kac Theorem, the corresponding (parametric) pricing PDE is given by:

$$f_\tau = \mathcal{L}f, f(0, s, x) = h(s) \quad \forall \tau \in [0, T], \mathbf{X} \in \mathcal{X}, s \in \mathcal{S} \tag{7}$$

We could consider a payoff determined by a continuous discounted payoff function h_1 and a terminal discounted payoff function h_2:

$$f(\tau, \mathbf{S_t}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[\int_t^T h_1(u, S_u)dt + h_2(S_T)|\mathbf{S}_t, \mathbf{X}\right]$$

Subject to some initial condition (in terms of time-to-maturity τ) $f(\tau = 0, \mathbf{X}) = h(\mathbf{S_T})$ corresponding to the European Payoff, and $\mathcal{L}$ is the generator of $\mathbf{S}$

$$\int_t^T \frac{\partial f}{\partial S_i}\sigma(t, \mathbf{S}_t)dt$$

Thus this also connects pricing with *stochastic control* and Forward-Backward Stochastic Differential Equations.

No-Arbitrage Constraints

No Dynamic Arbitrage: No dynamic arbitrage indicates the lack of a replication strategy with zero initial wealth that leads to positive wealth $\mathbb{P}$-almost surely. For Dynamic Arbitrage, a sufficient condition may be that the function g satisfies the relevant pricing PDE.

$$\mathcal{L}g = g_\tau, \quad \text{for all } \tau, K$$

If $\mathcal{L}g \neq 0$, the dynamic arbitrage strategy is given by longing the payoff if $\mathcal{L}g$ and shorting the replicating portfolio, and $\mathcal{L}g < 0$, and shorting the payoff if $\mathcal{L}g < 0$ and longing the replicating strategy.

Thus in this context , the no-arbitrage pricing function f denotes the solution to a PDE.

If g satisfies the corresponding pricing PDE, there is no *dynamic arbitrage*. For certain payoffs, there are *static arbitrage* bounds, where arbitrage can be obtained without dynamic rebalancing positions. We consider the conditions for our approximating function g to be free of static arbitrage in the case of European calls. Suppose that $g \in C^{1,2}$ is continuously and and once-differentiable with respect to $\tau = T - t$ and twice differentiable with respect to K.

The case of European calls for one asset is particularly relevaant as we can consider static no-arbitrage conditions.

Equivalently, we could consider moneyness $x = S/K > 0$, or log-moneyness $y = \log(S/K)$, which may be a more convenient representation in some cases, and assume g is twice differentiable with respect to:

$$\frac{\partial X}{\partial K} = \frac{S}{-K^2} = \frac{-X}{K} \quad , \quad \frac{\partial y}{\partial K} = \frac{-1}{K}$$
$$\frac{\partial g}{\partial K} = \frac{\partial g}{\partial x}\frac{\partial x}{\partial K} = \frac{-x}{K}\frac{\partial g}{\partial X} \quad , \quad \frac{\partial g}{\partial y}\frac{\partial y}{\partial K} = -\frac{1}{K}\frac{\partial g}{\partial y}$$
$$\frac{\partial}{\partial K}(-\frac{x}{K}\frac{\partial g}{\partial x}) = \frac{1}{K^2}(x\frac{\partial g}{\partial X} + x^2\frac{\partial^2 g}{\partial X^2})$$
$$\frac{\partial}{\partial K}(-\frac{1}{K}\frac{\partial g}{\partial y}) = \frac{1}{K^2}(\frac{\partial g}{\partial y} + \frac{\partial^2 g}{\partial y^2})$$

Then in terms of K, x, y we have:
No Static Arbitrage for European Calls: From [20] [10]

$\frac{\partial g}{\partial \tau} \geq 0$	carry, time-value
$\frac{\partial g}{\partial K} \leq 0, \frac{\partial g}{\partial x} \geq 0$	monotonically decreasing (increasing) in strike (moneyness / underlying)
$\frac{\partial^2 g}{\partial K^2} \geq 0, \frac{\partial^2 g}{\partial x} \geq 0, \frac{\partial^2 g}{\partial y} \geq 0$	convexity in strike, moneyness
$g(T-t, K) \geq (S_t - K)^+ \geq 0$	intrinsic value
$\lim_{K\to\infty} g(K, T-t) = 0$	1

We also consider no static arbitrage for European digitals. From the Breeden-Litzenberger formula we have:

Breeden-Litzenberg formula. We now have an additional approach: solve for the risk-neutral transition density $p(y, T; s, t)$ and obtain the price of any european payoff via numerical integration.

Since the payoff of a European digital $\mathbb{Q}[]$

$\frac{\partial g}{\partial \tau} \geq 0$	carry, time-value
$\frac{\partial g}{\partial K} \leq 0, \frac{\partial g}{\partial x} \geq 0$	monotonically decreasing in strike, increasing in moneyness/underlying
$\frac{\partial^2 g}{\partial K^2} \geq 0, \frac{\partial^2 g}{\partial x} \geq 0, \frac{\partial^2 g}{y} \geq 0$	convexity in strike
$g(T-t, K) \geq (S_t - K)^+ \geq 0$	intrinsic value
$\lim_{K\to\infty} g(K, T-t) = 0$	1

$$\int_{\mathbb{R}} (y-K)^+ p(y, T; s, t) dy = \int_K^\infty (y-K) p(y, T; s, t) dy$$

In the context of put-call parity, we can simply price one of the European payoffs and obtain the rest via replication, or integration from the derived transition density.

Pricing for Other Payoffs

If we are able to price a single European payoff, we can by extension, price payoffs that are linear combinations or *bundles* of European payoffs, for example interest rate caps or floors.

$$\sum_{i=1}^{n} f_i(\tau_i, \mathbf{S_t}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[\sum_{i=1}^{n} h_i(\mathbf{S}_{t_i})|\mathbf{S}_t, \mathbf{X}\right] = \sum_{i=1}^{n} \mathbb{E}^{\mathbb{Q}}\left[h_i(t_i, \mathbf{S}_i)|\mathbf{S}_t, \mathbf{X}\right]$$

However, payoffs may be a function of some subset of the trajectory $\mathbf{S}_t$, as opposed to a function of $\mathbf{S}_t$ for a single times.

Path-Dependent Payoffs: Payoffs may also be path dependent over the maturity of the option $\in [0, T]$, and depend on the entire history of $\mathbf{S_t}$:

$$f(\tau, (\mathbf{S}_u)_{u\in[0,t]}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[h_2((\mathbf{S_u})_{u\in[0,T]})|(\mathbf{S}_u)_{u\in[0,t]}\mathbf{X}\right] \tag{8}$$

The pricing function is no-longer Markovian, although in some cases we can make the pricing function Markovian through the choice of an appropriate state variable. For example, in the case of arithmetic Asian call options, we could consider the continuous-time running average:

$$f(\tau, S_t, A_t) = \mathbb{E}^{\mathbb{Q}}\left[(A_T - K)^+|S_t, A_t\right], A_t = \frac{1}{t}\int_0^t S_u dt, S_t \in \mathbb{R}, t \in [0, T] \tag{9}$$

or in the case of a knock-out barrier:

$$f(\tau, S_t, M_t) = \mathbb{E}^{\mathbb{Q}}\left[(S_T - K)^+ 1_{M_T \le b}|S_t, A_t\right], M_t = \max_{u\in[0,T]}\{S_u\} \tag{10}$$

Otherwise, we could consider approximating the true pricing function, with the conditional expectation with $\mathbb{E}[h((\mathbf{S}_u)_{u\in[t,T]})|\mathbf{S}_t]$, although this may approximation may not be arbitrage free.

Callable Payoffs: Another additional complexity, is that payoffs may also have callability features. In this case, we can determine whether to enter or exit a payoff by determing a stopping time $T^* \in \mathcal{T}$ over a set of possible exercise times $\mathcal{T}$.

$$f(\mathbf{X}) = \sup_{T^*\in\mathcal{T}} \mathbb{E}[h(\mathbf{S}_{T^*})|X] \tag{11}$$

No arbitrage bounds for non-vanilla payoffs: For non-European options, in some cases, we could replication arguments to obtain no-arbitrage lower and upper bounds. For example, for barrier options we must have:

$$V_{down} + V_{up} =$$

Or in the case of callable options with the same underlying $\mathbf{S}_t$, maturity T, and payoff function h the value is increasing in the number of exercise dates:

$$f^{\text{American}}(t, s) \ge f^{\text{Bermudan}}(\mathbf{X}) \ge f^{\text{European}}(\mathbf{X})$$

However, more generally we may not know the true no-arbitrage and boundary conditions. Different approximation methods g may not guarantee that the no-arbitrage constraints, and one question is how to incorporate these no-arbitrage constraints into a neural network approximation.

$$f\mathbb{E}^{\mathbb{Q}}\left[h_2((\mathbf{S_u})_{u\in[0,T]})|(\mathbf{S}_u)_{u\in[0,t]}\mathbf{X}\right] \tag{12}$$

Need for approximations: In general, the pricing function f in the forms (1) and 7, cannot be obtained in closed-form, with the exception of several cases. Thus in practice we consider some approximation g:

$$g(T-t,\mathbf{X}) \approx f(T-t,\mathbf{X}), \quad f(T-t,\mathbf{X}) = g(\mathbf{X}) + \epsilon \tag{13}$$

Where ϵ denotes the error, which may itself depend on $\mathbf{X}$ and the choice of numerical method. We look for a approximating function g such that ϵ is small over a relevant range of parameters $\mathbf{X}$.

Sensitivities

For market makers and traders, obtaining the sensitivities, or the differentials of the pricing function f with respect to the input parameters $\mathbf{X}$ is also of interest. Firstly, we should note that using the differentials of the pricing approximation to approximate the differentials of the true pricing function $\frac{\partial g}{\partial \mathbf{X}} \approx \frac{\partial f}{\partial \mathbf{X}}$ also introduces additional error. Thus we also require

$$\frac{\partial g}{\partial \mathbf{X}} \approx \frac{\partial f}{\partial \mathbf{X}}, \dots, \frac{\partial^d g}{\partial \mathbf{X}} \approx \frac{\partial^d f}{\partial \mathbf{X}^d}$$

Finite Differences: A natural method could be to consider approximating a partial derivative via first principles. If we have the function f, then:

$$\begin{aligned}\frac{\partial f}{\partial X_i} &= \frac{f(\cdots, X_i+\epsilon, \cdots) - f(\cdots, X_i-\epsilon, \cdots)}{2\epsilon}\\ \frac{\partial^2 f}{\partial X_i^2} &= \frac{f(\cdots, X_i+\epsilon, \cdots) - 2 * f(\cdots, X_i, \cdots) + f(\cdots, X_i-\epsilon, \cdots)}{\epsilon^2}\end{aligned}$$

For the first order this suggests that we must invoke the pricing function f, $2N_F$ times for a N_F-dimensional set of parameters $\mathbf{X} \in \mathbb{R}^{N_F}$

Adjoint Automatic Differentiation

The adjoint automatic differentiation method was first brought to [**giles2004**]

In the most general case, consider the pricing function in the form of **??**. Given regularity constraints

The sensitivities of f with respect to θ, can be computed through the application of the reverse chain rule

$$\frac{\partial f}{\partial \mathbf{X}} = (\frac{\partial \mathbf{S}_T(\mathbf{X})}{\partial \mathbf{X}})\frac{\partial}{\partial \mathbf{S}_T}\mathbb{E}^{\mathbb{Q}}[h_2(\mathbf{S}_T)] \tag{14}$$

n terms of programming implementation, it can be implemented via computional graphs and implementing *adjoint* operators. Further to this, adjoint automatic differentiation is a key tool that enables neural networks.

An issue with the adjoint method is that it fails for non-smooth payoff functions. However, we can address this somewhat with *payoff smoothing*. Consider for example, approximating a digital payoff $h(x)$ with a piecewise linear approximation $h_{ramp}(x, \epsilon)$

$$h(S_T) = 1_{S_T \geq K}, h_{ramp}(S_T, \epsilon) = 1_{S_T \geq K} + \frac{S_T - (K - \epsilon)}{\epsilon} 1_{K-\epsilon \leq S_T < K}, \frac{\partial h}{\partial S_T} = 0, \frac{\partial h_{ramp}}{\partial S_T} = \frac{1}{\epsilon} 1_{K-\epsilon \leq S_T < K}$$

Longstaff-Schwartz / Least Squares Monte Carlo

Thus far, we have described different approaches to obtain prices for fixed $\mathbf{X}$, although in the case of European Calls / Puts, we can obtain prices for a range of (τ, K), and for $(\mathbf{S}, T)$. We now consider a baseline method to approximate the pricing function f. [40] considered use the of sequential regressions as a method to approximate the value function for Americans.

For a single timestep (period), they consider approximating the true value function, or conditional expectation, f 1 using basis function regression. In [40], they consider basis functions such as Chebyshev and Legendre polynomials.

Suppose $\mathbb{E}[g(\mathbf{S}_{t_i})^2] < \infty$ the payoff is L^2 integrable.

$$g_{t_i}(\mathbf{S}_{t_i}) = \sum_{i=1}^{N_B} w_i \phi_i(\mathbf{S}_{t_i}) + b, \mathbf{S}_{t_i} \in R^d \tag{15}$$

Where in the one-step case:

$$g_{t_{N_T-1}}(\mathbf{S}_{t_{i-1}}) \approx \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_{t_{N_T-1}}] \tag{16}$$

The to price a Bermudan option they use sequential regressions:

$$g_{t_i}(t_i, \mathbf{X}) \approx \max\{\mathbb{E}^{\mathbb{Q}}[g(\mathbf{S}_{t_{i+1}})|\mathbf{S}_t]\} \tag{17}$$

$$f(X_{t_i}) = \max\{\mathbb{E}^{\mathbb{Q}}[f(X_{t_{i+1}})\}|X_{t_i}], h(X_{t_i})\}, \quad f(X_T) = h(X_T)$$

In the application of [40], they fix the contract parameters and volatility model parameters, and only $\mathbf{S_t}$ is varied (hence $\mathbf{X} = \mathbf{S}_t$.

Theorem such as Stone-Weirstrauss suggest the existence.

The use of Least Square Monte Carlo gives an approximation over the values of the stochastic process $\mathbf{S}_T$, but the volatility model parameters $\mathbf{X}$ are fixed.

In the multi-period case , we consider for $t_0 < t_i < \ldots t_n = T$:

Where h is the exercise value for the pay

[40] argues that as the number of basis functions B to infinity, g approximates the true value function, and as the number of timesteps N goes to infinity. Further

to this, in the Monte Carlo setting, we also require the number of samples M to go to infinity.

$$f(S_t) = \mathbb{E}[(S_T - K)^+ | S_t] \tag{Black-Scholes}$$

$$\mathbb{E}[(f(S_T) - \mathbb{E}[g(S_t)|S_t])^2] = 0$$

Then by the tower property of expectation

$$\mathbb{E}[(g(S_T) - \mathbb{E}[g(S_t)|S_t])^2] = 0$$

3 Neural Networks

In this section, we first review some relevant definitions of neural network concepts, and describe their features which may make them appropriate for the derivatives pricing and sensitivties problem.

[Tidy this section, possibly expand on SGD training and add Neural Tangent Kernel]

Neural Network Definition

A comprehensive outline of neural networks is found in [25], and a reference on practical implementations using `Tensorflow/Keras` can be found in [12]. An overview of neural networks and their applications towards finance can be found in [17].

Consider a dataset $\mathbf{X} \in \mathbb{R}^{N \times N_F}, \mathbf{y} \in \mathbb{R}^N$. Now, $\mathbf{X}$ is a matrix-valued inputs, or in machine learning the *features*, consisting of N samples of a N_F-dimensional vector, and the corresponding outputs are given by $\mathbf{y}$ or 'targets' to approximate. Compared to the previous section, each row vector $\mathbf{X}_i$ now represents a different parameter set. In many cases the true mapping function $\mathbf{y} = f(\mathbf{X})$ is unknown, although in the context of derivatives modelling, we know f, but not in analytic closed-form.

Feed-Forward Neural Network: A N_H-layer feed-forward neural network g (equivalently, a neural network with $N_H - 1$ hidden layers) with a one-dimensional output is characterised by:

$$\begin{aligned}
\mathbf{Z}^1 &= g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{1}(\mathbf{b}^1)^\top) \\
\mathbf{Z}^2 &= g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{1}(\mathbf{b}^2)^\top) \\
&\vdots \\
g(\mathbf{X};\boldsymbol{\theta}) &= g_n(\mathbf{Z}^{N_H-1}\mathbf{W}^n + \mathbf{1}(\mathbf{b}^n)^\top
\end{aligned} \tag{18}$$

Weights and Biases: Let $\mathbf{W}^i \in \mathbb{R}^{H_{i-1} \times H_i}$ be a real valued matrix denoting the *weights* for the i-th hidden layer, and $\mathbf{b}^i \in \mathbb{R}^{H_i}$ be a column vector denoting the *bias* term for the i-th hidden layer, where $i = 1, \dots N_H$. The product $\mathbf{1}(\mathbf{b}^i)^\top \in \mathbb{R}^{N \times H_i}$ represents adding $\mathbf{b}^i$ is added to each column of $\mathbf{Z}^{i-1}\mathbf{W}^i$. We note that much of the neural network consists of batch matrix-vector operations $\mathbf{Z}^{i-1}\mathbf{W}^i + \mathbf{1}(\mathbf{b}^i)^\top$, which are likely highly optimised in the underlying programming framework. This motivates the potential use of neural networks to obtain relatively fast inference over N samples at once (in terms of pricing predictions). Further speedups can be potentially obtained via dedicated hardware such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). In this direction, [37] mentions frameworks such as the NVIDIA TensorRT, which can further be used to speedup inference of a trained neural network and [27] discusses some considerations to further speed up neural network inference from the framework level.

Activation Functions: Let $g_i, i = 1, \dots, N_H$, denote the activation function for the i-th hidden layer. Activation functions are applied element-wise to the inputs, such that $g_i(\mathbf{Z}_{i-1})_{j,k} = g_i(\mathbf{Z}_{i-1,j,k})$. Table 2 displays some common activation functions and their first and second order gradients for a single input $x \in \mathbb{R}$. When only one hidden layer is used ($N_H - 1 = 1$), the activation functions are similar basis functions which act upon affine transformations of the input $\mathbf{X}$, however, with multiple hidden layers the neural network may be able to 'learn' more expressive non-linear transformations. Typically in a regression setting, where the output takes any real values in $\mathbb{R}$, the final layer is the identity activation $g_n(\mathbf{Z}_{i,j}^{N_H-1}) = \mathbf{Z}_{i,j}^{N_H-1}$. However, if knowledge is available about the range of the outputs, for example if we know the output takes values $y_i \in [0, 1]$, we could consider applying a final non-linear transformation g_n to constrain output to $[0, 1]$.

Activation	$g_i(x)$	$g_i'(x)$	$g_i''(x)$
ReLU	$\max\{x, 0\}$	$1_{x>0}$	0
LeakyReLU	$\max\{0, x\} + \alpha \min\{0, x\}$	$1_{x>0} + \alpha 1_{x<0}$	0
ELU	$\alpha(e^x - 1)1_{x<0} + x1_{x>0}$	$1_{x>0} + \alpha e^x 1_{x<0}$	$\alpha e^x 1_{x<0}$
Sigmoid	$\frac{1}{1+e^{-x}}$	$\frac{e^{-x}}{(1+e^{-x})^2}$	$\frac{e^{-x}(e^{-x}-1)}{(1+e^{-x})^3}$
SoftPlus	$\log(1 + e^x)$	$\frac{1}{1+e^{-x}}$	$\frac{e^{-x}}{(1+e^{-x})^2}$
Swish	$\frac{x}{1+e^{-x}}$	$\frac{1+((x+1))e^{-x}}{(1+e^{-x})^2}$	$\frac{((2-x)+(x-2)e^{-x})e^{-x}}{(1+e^{-x})^3}$
GeLU	$x\Phi(x)$	$x\phi(x) + \Phi(x)$	$(2 - x^2)\phi(x)$
tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$\frac{4}{(e^x+e^{-x})^2}$	$\frac{-8(e^x-e^{-x})}{(e^x+e^{-x})^3}$
RBF	$\exp(-\frac{x^2}{2})$	$-x\exp(-\frac{x^2}{2})$	$(x^2 - 1)\exp(-\frac{x^2}{2})$

Where $\alpha > 0$ denotes a hyperparameter, and Φ, ϕ denotes the cumulative density and probability density functions for the Gaussian distribution respectively.

Table 2: Activation Functions and their Derivatives

Automatic Differentiation: As described in the previous section, automatic differentiation enables fast computation of the gradients given an adjoint implementation of a function. In the context of neural networks, to obtain derivatives of the g with respect to the input $\mathbf{X}$, we could compute derivatives of the activation function analytically and apply the reverse chain rule through each layer. However, adjoint implementations of the activation functions g_i in the underlying deep learning framework, and noting that the gradients of the affine transformations $\mathbf{ZW} + \mathbf{1}(\mathbf{b}^\top)$ are simply $\mathbf{W}$, lead to efficient evaluations of the gradients of the neural network ∇g. The gradients ∇g enable efficient computation for the gradients with respect to the parameters, but also $\boldsymbol{\theta}$ which is needed for training the neural network, but can also be applied to obtain efficient gradients for the gradients with respect to the input $\frac{\partial g}{\partial \mathbf{X}}$. This motivates the potential for using neural networks to obtain fast first-order, and potentially higher-order differentials for derivatives modelling. [46] discusses some potential speedups in inference of the first and second order differentials using a neural network, in particular computing the sensitivities / Jacobian / Hessian with respect to $\mathbf{X}$ explicitly as opposed to using AAD. On the enginenering side, further investigtation between different deep learning implementations, such as Tensorflow, PyTorch, Jax, in Python, and Flux in Julia could also be considered.

Neural Networks as a composition of neural networks and basis functions: We note that the neural network g is a composition of $g_n(g_{n-1}(\cdots))$. Thus one way of interpreting neural networks, is to consider them as being simply a composition of sub-neural networks or non-linear affine transformations.

Anther way of viewing a neural network may be to consider it as non-parametric 'learning' a collection of non-linear basis functions (also referred to as 'latent representation' or 'features') $\mathbf{Z}^{N_H-1}$ from inputs $\mathbf{X}$ that is useful for the given task. Consider that if we have no final activation g_n, then the output is simply $\mathbf{Z}^{N_H-1}\mathbf{W}^n + \mathbf{1}(\mathbf{b}^n)^\top$, a ordinary least squares regression of $\mathbf{y}$ on $\mathbf{Z}^{N_H-1}$. This draws some comparison with the Least-Square Monte Carlo method (Equation 15) and when $N_H = 1$ (i.e. no hidden layers) and g_1 is the identity function, the neural network is *exactly* linear regression. In the case of $N_H = 2$ (one hidden layer), and fixed $\mathbf{W}^1, \mathbf{b}^1$, then $\mathbf{Z}_1(\mathbf{X})$ would represent some fixed basis functions. However, in the more general case of $N_H > 1$ and non-fixed weights and biases $\boldsymbol{\theta}$, compared with standard basis regression, the basis functions $\mathbf{Z}^{N_H-1}(\mathbf{X})$ explicitly, but are determined from the dataset and objective. In addition, we could also consider neural networks as a non-linear form of dimensionality reduction. If we have we select the final hidden layer size to be less than the input dimension $H_{N-1} < N_F$, the basis function $\mathbf{Z}^{N_H-1}(\mathbf{X})$ could represent a lower dimensional projection of the input-space. This motivates the potential application of neural networks as a basis function regression method that can be extended to high-dimensional settings.

Another implication is that we could consider a multi-output neural network to be a composition of neural networks one-dimensional output. In 18, we describe a one-dimensional neural network $H_{N_H} = 1$, although if we let $H_{N_H} \in \mathbb{Z}^+$, $\mathbf{Z}^{N_H-1}\mathbf{W}^{N_H}$ maps $\mathbf{Z}^{N_H-1}$ to a H_{N_H}-dimensional output. Alternatively, we could also note that:

$$g_n(\mathbf{Z}^{N_H-1}\mathbf{W}^{N_H}) = \Big(g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,1} + \mathbf{b}) \quad \cdots \quad g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,H_{N_H}} + \mathbf{b})\Big) \tag{19}$$

Thus each column vector of $\mathbf{W}^{N_H}$ determines a mapping to a specific output, from the shared basis functions $\mathbf{Z}^{N_H}$. The neural network may be able to 'learn' a single set of basis functions to predict multiple outputs (multi-task learning). In this context, [42] considers using a neural network with a $H_{N_H} = 10$-dimensional output of SABR-implied volatilities at fixed strikes, and [30] also considers predicting the outptus for a fixed collection of prices for strike and maturity. Another application could be to jointly model European calls and puts. This may be more efficient than standard basis functions regression, where the same collection of basis functions may not be optimal for each of the H_{N_H} outputs.

Transfer Learning: Another potential application is the notion *transfer learning*, a machine learning technique that has been applied to other fields, such as image-related modelling. Transfer learning involves retraining a neural network (usually the last few layers) which has already been trained on some similar task. For the images domain, an example could be to retrain a neural network to classify dogs instead of cats. For the derivatives modelling context. In the derivatives pricing context, we could assume for example, that the learnt basis functions in the final layer in a neural network for European Calls may also be useful for European Puts. However, as opposed to the multi-output case, we extract the basis functions $\mathbf{Z}^{H_N-1}$ (also referred to as 'feature extraction') for one given function, and then retrain and determine the optimal weights for the basis functions $\mathbf{W}^{\mathbf{N}_H}$ for this new pricing function. [3, 22] explore this application.

Ensembling: As we will later describe, neural networks have some non-determinism, and we may also be unsure how to choose a neural network from multiple architectures. As opposed to considering a single neural network, we could consider a weighed average of multiple neural networks.

$$\sum_{i=1}^{N_E} a_i g(\mathbf{X};\boldsymbol{\theta}^1) = \frac{1}{N_E}\left(\mathbf{Z}^{N_{H-1},1} \quad \mathbf{Z}^{N_{H-1},2} \quad \cdots \quad \mathbf{Z}^{N_{H-1},N_E}\right)\begin{pmatrix} a_1\mathbf{W}^{N_H,1} \\ a_2\mathbf{W}^{N_H,2} \\ \vdots \\ a_{N_E}\mathbf{W}^{N_H,N_E} \end{pmatrix} \tag{20}$$

Where the relative weights α_i could be fixed or further determined. We could also think of this as a new neural network, with a penultimate dimension $N^*_{H-1} = N_E \times N_{H-1}$.

Although in general, a point estimate for the price is needed, this method can also used to quantify uncertainty bounds for the neural network price approximation. For example, [23] considers initialising multiple random seeds, to obtain a confidence interval on prices. A somewhat related concept is Bayesian Neural Networks.

$$\left[\min\{g^1(\mathbf{X};\theta_1),\ldots g^n(\cdot;\theta_2)\} = g(\mathbf{X};\theta^-), g(\mathbf{X};\theta^+) = \max\{g^1(\mathbf{X};\theta_1),\ldots g^n(\mathbf{X};\theta_2)\}\right]$$

A potential drawback of this approach is that this may potentially lead to a more complex implementation, and reduce inference speed somewhat, given the need to evaluate n neural networks instead of 1. Another implication, is that we could aggregate neural networks, for exaample by taking a weighted average of N_E neural networks with the same input and output dimensions.

Neural Network Theory and Training

In the previous section, we characterised some of the properties of neural networks as a class of functions, and drew the comparison between neural networks and basis function regression.

Theorem 1 (Hornik (1990)) *Let $\mathcal{N}_{H_0,H_1,g}$ be the set of neural networks mapping from $\mathbb{R}^{H_0} \to \mathbb{R}^{H_1}$, with activation function $g : \mathbb{R} \to \mathbb{R}$. Suppose $f \in C^k$ is continuously k-times differentiable. Then if $g \in C^k(\mathbb{R})$ is continuously k-times differentiable, then $\mathcal{N}_{H_0,H_1,g}$ arbitrarily approximates f and its derivatives up to order n.*

The universal approximation of theorem of [29] provides a theoretical justification for using neural networks to approximate a pricing function and its derivatives; there exists some neural network that arbitrarily approximates our pricing function and it derivatives. The theorem suggests that for our derivatives modelling application, this suggests to obtain k-th order sensitivities, we must use a continuous k-times differentiable C^k activation function g. This rules out the commonly used Rectified Linear Unit (ReLU) activation function, and the LeakyReLU function.

However, the universal approximation theorem only proves existence of such a neural network, and not how to construct it. For example, we have the questions of the architectural choices: the number of hidden layers N_H, hidden units in each layer H_i, and which smooth activation function to use, and whether to consider special block architectures (residual, gated, or standard feed-forward blocks).

Suppose we first fix the number of hidden layers N_H, and the dimensions of each hidden layer $H_i, i = 1, \dots N_H$, and the choice of activation functions g_i. Then what remains is to determine the optimal $\boldsymbol{\theta}$, which denotes all *learnable* parameters of the neural network. In a feed-forward network, this amounts to the collection of all weights and biases $(\mathbf{W}^1, \dots \mathbf{W}^n, \mathbf{b}^1, \dots, \mathbf{b}^n)$. A neural network is non-parametric in the sense that we do not necessarily have to make any assumptions about or specify the functional or distributional relationships between inputs and targets $\mathbf{X}, \mathbf{y}$, but the values of $\boldsymbol{\theta}$ need to be determined or 'learnt' over some space of possible parameters Θ. For example, we could simply take the product of all real-valued matrices and vectors with the corresponding dimension as the weights and biases $\Theta = \prod_{i=1}^n \{\mathbb{R}^{H_{i-1} \times H_i}\} \prod_{i=1}^n \mathbb{R}^{H_i}$.

Thus we define the 'optimality' of a neural network as the optimal choice of $\boldsymbol{\theta}$, with respect to a particular loss function L, and over a feasible parameters Θ

$$\arg\min_{\theta \in \Theta} L(\mathbf{y}, g(\mathbf{X})) \tag{21}$$

For example, we could consider two common objective functions for machine learning regression. For example, we could consider the expected Mean Absolute Error (MAE) or Mean Squared Error (MSE):

$$\arg\min_{\theta \in \Theta} MAE(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \mathbb{E}[\|\mathbf{y} - g(\mathbf{X}; \theta)\|] \tag{22}$$

$$\arg\min_{\theta \in \Theta} MSE(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \mathbb{E}[\|\mathbf{y} - g(\mathbf{X}; \theta)\|^2], \tag{23}$$

In equation (25), the input parameters $\mathbf{X}$ are no longer fixed, but a $\mathbb{R}^f$ random variable over some probability space $(\mathcal{X}, \mathcal{F}^{\mathcal{X}}, \mathbb{P}^*)$. Thus in effect, we are minimising some $L^p \times \mathbb{P}^{\mathcal{X}}$ norm. However, in actuality we are only able to generate a finite samples of $\mathbf{X}$ from the true parameter space $\mathcal{X}$. Thus we aim to minimise the corresponding L^p error over the empirical measure.

$$\arg\min_{\theta \in \Theta} \mathrm{MAE}(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} |y_i - g(\mathbf{X}_i)| \tag{24}$$

$$\arg\min_{\theta \in \Theta} \mathrm{MSE}(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - g(\mathbf{X})_i)^2, \tag{25}$$

This approach is referred to as *supervised machine learning*, in which we have explicit input output pairs $\mathbf{X}, \mathbf{y}$, and the neural network learns some approximation through the loss function L. Different loss functions have different meaning, and may lead to different results for θ, although any generic loss function could be considered. We should note that although the neural network is trained on a single loss function L, we can evaluate its performance on multiple metrics, and that minimising the loss L may not necessarily, but can potentially, lead to improved performance in the other metrics.

In general, the loss function is non-convex with respect to the parameters θ. Thus the training of the neural network is a non-linear optimisation problem. To obtain an estimate for a global minima $\boldsymbol{\theta}^*$, one method is to use the *mini-batch stochastic gradient descent algorithm*. In effect, we perform iterative updating to the neural network parameters $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \nabla_{\boldsymbol{\theta_t}} L$. However, there is are no guarantees of convergence to a global minima, except under certain conditions (one such condition is if the neural network is convex with respect to θ, and λ_t is appropriately chosen).

Algorithm 1 Mini-Batch Stochastic Gradient Descent

```
Initialise parameters $\boldsymbol{\theta_0}$ randomly via PRNG, $t = 0$
while $t \leq \text{EPOCHS} \times \lceil \frac{N}{\text{BatchSize}} \rceil$ or NOT StoppingCriterion do
  for batch = $\lceil 1, \dots, \frac{N}{\text{BatchSize}} \rceil$ do
    Compute loss $L(y^{batch}, g(\mathbf{X}^{batch}))$
    Compute gradient w.r.t. loss $\nabla L$ via AAD
    $t \leftarrow t + 1$
    Set $\theta_{t+1} \leftarrow \theta_t - \eta_t \nabla_\theta L$
  end for
  if StoppingCriterion then Break
  end if
end while
```

Weight Initialisation: Firstly, we randomly initialise the parameters $\boldsymbol{\theta}_0$. Typically, the weights are drawn from some random distribution $\mathbf{W}^i_{j,k} \sim N(0, \sigma^W)$. The weight initialisation step introduces some randomness, which is why we could consider the ensembling as opposed to a single neural network.

Backpropogation Algorithm: For each iteration t, we compute the the loss L with respect to the parameters $\boldsymbol{\theta}$. Then, the gradients $\nabla_{\boldsymbol{\theta}} L \partial \boldsymbol{\theta}$ with respect to the parameters $\boldsymbol{\theta}$ can be obtained via in-built AAD in the neural network framework of choice.

This procedure is repeated over all batches in the training dataset $\mathbf{X}$, and repeated until the earlier of a given number of iterations (epochs) or some stopping criterion.

Mini-batching: In practice the gradients with respect to the neural network parameters $\nabla_\theta L$ also need to be estimated. The dataset $\mathbf{X}, \mathbf{y}$ is partitioned into chunks of size BatchSize. The consequence of this is that a smaller BatchSize enables a lower memory usage and also increases the number of updates (for the same number of passes or EPOCHS over the dataset); on the other hand a larger BatchSize may lead to more stable estimates of the true gradients $\nabla_\theta.L$ Mini-batch stochastic gradient enables training at scale. We can fix a time budget proportional to the maximum number of iterations to consider $O(\lceil \frac{N}{\text{BatchSize}} \rceil \times \text{EPOCHS})$, and the memory usage is proportional $O(BatchSize)$ as opposed to the entire dataset $O(N)$. Furthermore, the procedure can be parallelised, or further spedup using dedicated hardware such as TPUs or GPUs.

Learning Rate and Optimizers: In lieu of a fixed learning rate $\lambda_t = \lambda$, we could use an alternative learning rate schedule policy. In addition, we could also replace $\nabla_\theta L$. Adam.

Overfitting

Early Stopping: As mentioned, the empirical mean squared error is a proxy for the true error over the entire sample parameter space $\mathcal{X}$. In the case of early stopping, we partition $(\mathbf{X}, \mathbf{y})$ into $\mathbf{X}^{train}, \mathbf{y}^{train}, \mathbf{X}^{val}, \mathbf{y}^{val}$, or sample another dataset independently from the same or another parameter space $(\mathcal{X}^{val}, \mathbf{y}^{val})$. We withold $\mathbf{X}^{val}$ from training (i.e. use in gradient descent), but on each epoch, evaluate the performance of g on $\mathbf{X}^{val}, \mathbf{y}^{val}$. It may be that the neural network is able to perform well for any $\mathbf{X}_i$, but not for other points in $\mathcal{X}_{sub}$, or indeed, for a larger sample parameter space $\mathcal{X}_{sub} \subseteq \mathcal{X}_{sub2}$. Thus the number of iterations simply stops when the performance on $(\mathbf{X}^{val}, \mathbf{y}^{val})$ no longer improves. Hence in practice, unless we have a time budget, we need only tune the initial learning rate λ_0 and batch size BatchSize and set a large number of epochs, and let EarlyStopping terminate training.

Regularisation Methods: In practice, several neural network techniques have been found empirically to improve training speed and generalisation, some of which are outlined in [38]

Batch Normalisation: The outputs of each layer are normalised element-wise by batch $(\mathbf{X}_i \ominus \mu_i)/ \oslash \sigma_i$. By ensuring a constant mean and variance, this may help alleviate vanishing and exploding gradients, enabling faster training.

Dropout: [49] proposed the *Dropout* method as a form of neural network regularisation. During training, fraction p of hidden units are set to zero, and the remaining units are scaled by $1/p$ to preserve the expected mean and variance. $\mathbf{X} \otimes \mathbf{R}\frac{1}{p}$, where the entries of $R_{ij} \sim Bin(p)$ are bernoulli distributed with probability p. This has the effect of preventing ‘overdependence’ on a particular basis function.

Weight Regularisation: setting constraints on the weights, for example a penalty on the L^1 or L^2 norm on the hidden layer weights $\mathbf{W}^i$. This amounts to modifying the loss function, for example: $L + \lambda \sum_{i=1}^{n} \|\mathbf{W}^i\|_2$. It may also help to introduce sparsity, for example, we could consider pruning the basis functions in the final layer with a small weight.

Special Architectures

Another consideration is whether to use special neural network architectures in place of feed-forward neural networks 18. We describe several hidden layer, or ‘block architectures' that could be potentially used.

Gated unit: In effect, with a gated unit we multiply the outputs of two hidden layers element-wise where $\otimes$ denotes element-wise multiplication.. Here, we need $\mathbf{W}^1, \mathbf{W}^2 \in \mathbb{R}^{H_0 \times H_1}$, such that the $\mathbf{Z}_1, \mathbf{Z}_2$ have the same dimension. This can facilitate learning multiplicative interactions between the basis functions / feature. [53] uses gated units to construct a neural network that has the requisite monotonicity and convexity constraints with respect to strike and moneyness.

$$\mathbf{Z}^1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{b}^1\mathbf{1}^\top) \tag{26}$$
$$\mathbf{Z}^2 = g_2(\mathbf{X}\mathbf{W}^2 + \mathbf{b}^2\mathbf{1}^\top) \tag{27}$$
$$\mathbf{Z}^3 = \mathbf{Z}_1 \otimes \mathbf{Z}_2 \tag{gated block}$$

Residual block: Residual blocks allow for the *flow* of information from earlier layers to later layers. Here, we need $\mathbf{W}^2 \in \mathbb{R}^{H_1 \times H_1}$, such that $\mathbf{Z}^1$ and $\mathbf{Z}^2$ have the same shape.

$$\mathbf{Z}_1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{b}^1\mathbf{1}^\top) \tag{28}$$
$$\mathbf{Z}_2 = g_2(\mathbf{Z}_1\mathbf{W}^2 + \mathbf{b_2}\mathbf{1}^\top) + \mathbf{Z}_1 \tag{residual block}$$

Consider the case for a neural network with one residual block followed by a single output layer with the identity activation, with mean-squared error as the loss function. Then:

$$\begin{aligned}
\mathbf{Z}^3 &= \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top && (29)\\
L(\mathbf{y}, g(\mathbf{X})) &= \|(\mathbf{y} - \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top)\|^2 && (30)\\
&= \|\mathbf{y} - \big((g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{b_2}\mathbf{1}^\top) + \mathbf{Z}^1)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top)\big)\|^2 && (31)\\
&= \|\mathbf{y} - (g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{b_2}\mathbf{1}^\top)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top] - (\mathbf{Z}^1\mathbf{W}^3)\|^2 && (32)
\end{aligned}$$

We can think of each layer in the residual block as learning the residual of the previous output projection. This may potentially facilitate faster training of neural networks, and alleviate vanishing gradients, for no additional parameters.

Determining Optimal Architecture

In the previous sections, we described how to train a neural network to determine $\boldsymbol{\theta}$ for a fixed architecture. Indeed, in some cases, a architectural choice may indeed need to be fixed, due to resource (computational or time constraints). However, in a setting where we are able to train a neural network offline, it may be worth considering multiple families of neural network architectures. Although we may be able to train a fixed neural network architecture to a local minima, there may be some other neural network architecture that leads to a lower loss. For example, in [28] the author considered standard-feed-forward and convolutional neural networks for swaption calibration, but highlighted that after the choice of special architecture, additional complexity remains in determining the numerous architectural choices;the number of hidden units, layers, and other hyperparameters.

AutoML, Hyperparameter Tuning: From the Universal Approximation theorem, in theory, there exists some neural network with only $(N_H - 1) = 1$ which can arbitrarily approximate our target pricing function (for example in the case of a infinite width). However, empirically, in addition to the width of aa neural network, depth, the choice of special architectures also play a role. One way to determine the optimal architecture is to consider neural architecture search or automatic machine learning (AutoML). In effect, we consider some space of neural networks $\mathcal{N}$, for example in terms of the number of hidden layers or hidden units. We then evaluate some finite subset of $\mathcal{N}_{sub} \subset \mathcal{N}$ and consider:

$$\arg \min_{g_i \in \mathcal{N}_{sub}} \min_{\theta \in \Theta_{g_i}} L \tag{33}$$

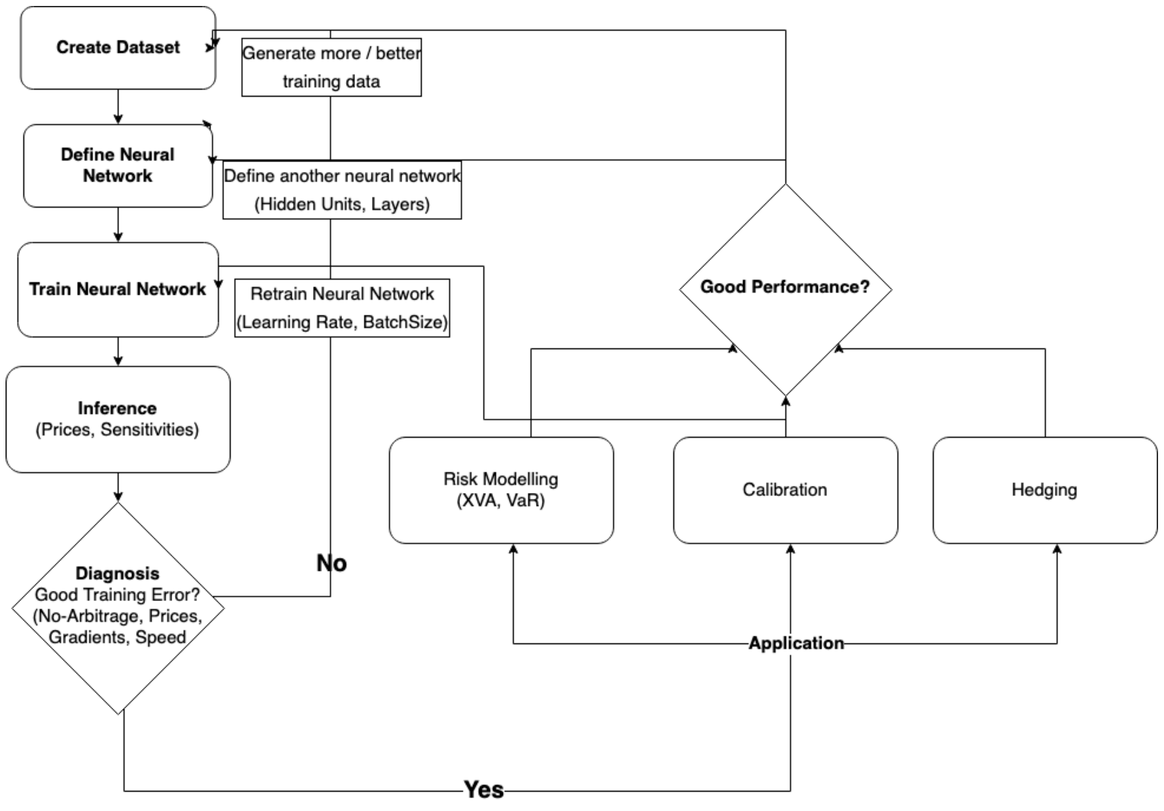
A naive method to evaluate $\mathcal{N}$ could be to define some small finite search space (grid-search) $\mathcal{N}_{sub} = \mathcal{N}$, for example through a cartesian product on a finite set of possible hidden units, layers, and activation functions.[45] described a method to conduct a grid search over a small parameter space in a way that can ”pragmatically satisfy model validation requirements”. As a simple example, in Example Grid Search we consider a grid search for a neural network with the the same number of hidden units H_i and activation function g_i for $i = 1, \dots, N_H - 1$ hidden layers.

$$H_i \times (N_H - 1) \times g_i \in \{32, 64, 128\} \times \{32, 64, 128\} \times \{\text{ELU}, \text{Swish}\} \tag{Example Grid Search}$$

However, for a larger search space (or for example, with continuous parameters), and also in the case of a fixed time constraint, a brute-force search becomes infeasible. Instead, a randomised subset must be considered through a random search, or more sophisticated optimisation methods such as Bayesian optimisation. In `Tensorflow/Keras`, this can be implemented through the `KerasTuner` API, which [24] considered in their paper for using neural networks to solve parametric pricing PDEs.

4 Neural Networks for Derivatives Pricing

In the previous sections we considered the problem formulation of approximating derivatives prices and sensitivities, the characteristics and how to construct of neural networks. In this section, we review some of the existing literature and methods on how to neural networks in derivatives pricing and risks from the perspective of the entire workflow. [47] presents an excellent survey article on neural networks towards derivatives modelling.



Create Dataset
Generate more / better training data
Define Neural Network
Define another neural network (Hidden Units, Layers)
Train Neural Network
Retrain Neural Network (Learning Rate, BatchSize)
Good Performance?
Inference (Prices, Sensitivities)
Risk Modelling (XVA, VaR)
Calibration
Hedging
No
Diagnosis Good Training Error? (No-Arbitrage, Prices, Gradients, Speed
Application
Yes

Figure 1: Example workflow for using neural networks in derivatives modelling

As a broad generalisation, for the context of using neural networks to approximate pricing functions, there may be potentially two settings: 1) [42] [30] [19] train a neural network offline over a large dataset to approximate a given pricing function to a high degree of accuracy, and then use it online in place of the pricing function as a faster 'digital clone', 2) [31] use a neural network to solve a to solve a complex pricing problem on-the-fly (e.g. high-dimensional, callable, path-dependent), possibly with some parameters (volatility, payoff structure) fixed.

Dataset Construction

In addition to the choice of neural network, the entire workflow for training a neural network (depicted in Figure 1) is of importance. In the previous section, we described the need for inputs $\mathbf{X}, \mathbf{y}$ in the machine learning framework. However, we have not elabourated on how to generate a dataset of $\mathbf{X}, \mathbf{y}$.

For modelling the parameter space $\mathbf{X}$, a naive method could be to sample independently from each dimension, from some distribution (e.g. normala uniform). A naive example could be to simply sample uniformly in each dimension, such that $\mathbf{X} \in \mathcal{X} = \prod_{i=1}^{N_F}[a_i, b_i]$. [**babbar**] slides discusses the complexity involved with [28] mentions the possibility of sampling parameter spaces from historical observations of joint distributions parameter, to capture a meaningful space of parameter relationships (as opposed to sampling uniformly in a hypercube).

Given parameter pairs, we need to generate the corresponding target prices $\mathbf{y}$. Given an assumption of some volatility model and payoff, we can leverage known efficient numerical methods to obtain prices with MC and PDE. For example, in the case of MC we could leverage Quasi Monte Carlo, Sobol Sequences, or Control Variates, or for PDE methods we could leverage efficient implicit scheme methods or sparse grids.

However, although we may wish to approximate the true pricing function $\mathbf{y} = f(\mathbf{X})$, we can do so through defining different choices of $\mathbf{y}$. For example, in [31], they consider $\mathbf{y}$ to be sample Monte Carlo payoffs, such that y_i is an unbiased estimate of the true pricing function $\mathbb{E}[y_i] = f(\mathbf{X}_i)$.

Another approach could be to learn aa mapping for the Black-Scholes implied volatility function, such that we can obtain the pricing function through Black-Scholes $\mathbf{y} = f^{BS}(\hat{\sigma}_{imp}(\mathbf{X}) = g(\mathbf{X}), \mathbf{X})$, as in [30] and [42].

In the more general case of any volatility model, we could consider the inverse problem of calibration predict the volatility model parameters

$$g(\mathbf{y}, \mathbf{X}^{contract}) = \mathbf{X}^{vol}, f(\mathbf{X}^{vol}, \mathbf{X}^{contract}) = \mathbf{y}$$

The risk of the neural network is also present through *data risk.* Dataset quality For example, how does the NN perform on unseen parameter ranges?, An issue is that the NN may be able to , e.g. when the market regime shifts. Thus the NN must likely be trained on a very large grid of parameters.

In [31] and the standard Longstaff-Schwartz approach,

In the case of hedging, FBSE, and American / Bermudan options modelling,

The neural network approximation approach is more closely aligns with a Monte Carlo method. Thus there is error from both the neural network approximation g and the simulation of state-payoff pairs X_j, y_j

For example, consider the case of very deep in-the-money or out-of-the money strikes for a European all. A proposed solution of is to sample more from these regions, or similarly to set a greater weight w_j in a weighted loss function

Data Preprocessing: Given that we may need a numerical method to generate the price labels $\mathbf{y}$, the dataset may contain noise $\boldsymbol{\epsilon}$ $\mathbf{y} = f(\mathbf{X}) + \boldsymbol{\epsilon}$. In addition, the dataset may also contain arbitrageable prices.

In the Longstaff-Schwartz paper, they simply set out-of-the money paths to weight zero:

$$\sum_{j=1}^{N} w_j L(f(X_j), g(X_j))$$

In the Longstaff-Schwartz case, paths for which $f(X_j)$ have weight zero.

In this same style, [14]. Preprocessing, remove arbitrage out of the money paths.

Neural Network Construction for Derivatives Pricing and Risks

[INCOMPLETE, tidy this section]

We want a neural network that achieves good performance across, ideally, all objectives: inference time, no-arbitrage error, pricing error, greeks error, and performance in the downstream application. In this case, suppose we fix the neural network hyperparameters, which might correspond to assumptions or limitations. The aim is to consider whether different neural network constructions, for example different loss functions may lead to lower errors in some, if not all, of these objectives. We characterise handcrafted neural networks as described in [47]: in short, we can embed prior knowledge into the construction of the neural network, in the choice of an appropriate loss function or architecture.

A machine learning setup generally consists of several components: the modelling objective, the dataset, the model, training, and inference.

The first question is how to approach modelling for the pricing approximation task. To date, various approaches utilising neural networks for derivatives pricing and hedging have been presented, of which a comprehensive review of methods can be further found in [47]:

Supervised: Direct approximation of model prices. [32] was one of the first papers use neural networks for option pricing, although their application was on real options pricing data. Their approach to use neural networks as a direct funcntion approximation between inputs, and S&P500 options price data. However, [32] did not consider sensitivities.

Model hedging strategy / FBSDE; Approximation for a hedging or replicating strategy; In effect this can be seen as pricing via replication or stochastic control [6]. The *Deep Hedging* approach of [6] considers approximating the replicating strategy. Thus in this case, the neural network represents the delta.

$$y - \sum_{i=1}^{N} \Delta_i(X_{t_i})(S_{i+1} - S_i)$$

Learn the hedging strategy with a neural network, and price can be obtained via superhedging. However the drawback is that this hedging strategy is tuned to a single payoff and requires MC simulations for pricing. Other papers that explore this reinforcement learning based approach include

Neural Network Architecture - 'Hard Constraints'

Hard Constraints: This refers to architectural constraints mostly through fixing the activation functions and imposing constraints on the weights.

Smoothness: Firstly, to obtain smooth (or at least, continuous) approximations for the *d*-th order partial derivatives, we require $g(\cdot)$ to be C^d continuous d-time differentiable with respect to its inputs [33]. Given that the composition of smooth functions is smooth, this leads to the neural network g being smooth in $\mathbf{X}$. [11] uses this approach. Given that we may also require smooth gradient approximations, we need the differentials of the activation function to be smooth as well. This suggests that the softplus, swish, and gelu activation functions may be appropriate, although the latter two are non-monotonic. [11] *softplus* activation for the strike and sigmoid activation for the time-to-maturity, with non-negative weight constraints $\mathbf{W}^i \geq 0$. Thus this guarantees that the neural network is non-negative, monotonic in strike and time, and convex in strike. [33] propose a modified elu function with: $R(z) = \alpha(e^z - 1)1_{z\leq 0}+$

Loss Functions for Derivatives Pricing and Risks

We now consider a choice of specific loss functions for the supervised learning task.

In the most simple case, we consider direct approximation, for example with mean squared error as a loss function. For a single observation: (X_i, y_i), we could consider the Mean Squared Error of our pricing approximation

Loss Function with Price:

$$L_{price}(y_i, g(X_i)) = (y_i - g(\mathbf{X}_i))^2 \tag{34}$$

Loss Function with Implied Volatility: Alternatively, we could consider modelling some proxy for price, for example Black-Scholes or Bachelier implied volatility. In this case, we could either let y_i denote the implied volatility, or consider applying the Black-Scholes formula to the predicted Black-Scholes implied volatility $f^{BS}(g(X_i))$:

$$L_{price}(y_i, f^{BS}(g(X_i)))L_{price}(y_i, g(X_i)) \tag{35}$$

[42] considers predicting implied volatility. A potential advantage of this is that the implied volatility might be more well defined in terms of being bounded over some range $[\sigma_{min}, \sigma_{max}]$, whereas the output range for a call option might be unbounded. In addition, formulation in this way means that we also obtain fast calibration for the Black-Scholes model.

Control Variate: Another first choice could be to change the predicted output. The proposed solution of [1] uses a two step approach: they first compute some discrete set of prices (via MC or FD), fit a cubic spline interpolation with the target asymptotics, and then compute the *residuals* of the pricing error against the cubic spline fit. This control variates method can also be connected with asymptotic expansions, for example in [21], the neural network is used to correct the errors of the

SABR approximation of [26]. In this context, the neural network is used to 'correct' the error of another pricing approximation.

$$L_{price}(y_i, g(X_i)) = (y_i - \text{ControlVar}_i - g(\mathbf{X}_i))^2 \tag{36}$$

Given that the cubic spline interpolation has the correct asymptotics, their proposed architecture leverages a specific activation function, the radial basis function (RBF) to ensure that the predicted price $ControlVar + g(\mathbf{X_i})$ has the correct asymptotics.

$$g_i(x) = \exp(-\frac{1}{2}\|\mathbf{x}\mathbf{W}_i + \mathbf{b}_i\|^2) \tag{37}$$

This is given that as $x \to \pm\infty$ we have $g_i(x) \to 0$. Thus if our neural network consists only of RBF activations, we can obtain the correct target asymptotics. [1] highlights that this is important, as neural networks are generally able to interpolate within the domain of training, but unable to extrapolate beyond its training domain. Moreover, for European payoffs, asymptotic conditions also correspond to no-arbitrage bounds; thus in this case the predicted price is guaranteed to satisfy (some) no-arbitrage conditions.

Soft Penalties: Another approach is to incorporate no-arbitrage constraints into the loss function, which [33] describes as a *soft penalty* approach. This is similar to the penalty method in convex optimisation. For example, we could consider any of the no-arbitrage constraints, and include a loss term if the . One such example could be a penalty $((S-K)^+ - g(S))^+$, i.e. a penalty for being beneath the intrinsic call bound. In the most general case, let $L_{cons_1}, \ldots L_{cons_P}$ denote P soft penalties. Then in this case, the loss function is given by:

$$L_{\text{soft penalty}}\lambda_0 L_{\text{price}} + \sum_{i=1}^{P} \lambda_i L_{\text{cons}_i} \tag{38}$$

The soft penalty approach is able to penalise linear constraints in the pricing approximation, and , but not necessarily account for the asymptotic boundary cases as the control variate approach. In addition, the soft penalty approach is not *guaranteed* to ensure that the no-arbitrage constraints are satisfied, unlike a hard constrained neural network. The additional challenges are now that the problem becomes multi-objective optimisation problem, and we must decide the relative weightings λ_i for the error in pricing and each no-arbitrage constraint. In addition, if we consider higher-order differentials, this necessarily increases the total training time, as during each training iteration we must compute the gradients of the parameters with respect to the differentials $\frac{\partial g}{\partial \mathbf{X}} \frac{\partial \mathbf{X}}{\partial \theta}$.

Differential Machine Learning: The proposed method from [31] is to consider a joint loss function, an approach they describe as *differential machine learning.*

$$\lambda L_{price}(g(X_i), f(X_i)) + \lambda L_{greeks}\left(\frac{\partial g(X_i)}{\partial X}, \frac{\partial g(X_i)}{\partial X}\right), \lambda \geq 0 \tag{39}$$

The partial derivative $\frac{\partial f(X_i)}{\partial X}$ can be obtained at some (small) extra computational cost given AAD. [31] argues that the λ_1 is not significant and thhey set it to $\lambda_1 = 1$, but in practice this could be determined by considering prior knowledge of the relative scales, or through trial-and-error.

Neural PDEs: A further extension could be to consider the corresponding PDE associated with the , which can be described as a *Physics-Inspired Neural Network* or a *Neural PDE* approach.

Use of neural networks to solve (high-dimensional) PDEs as in [**sirignano**]. Given that in some cases, pricing problems can be formulated as. In addition, in our application we would need to solve the PDE over a range of model parameters.

$$L_{price}(g(X_i), f(X_i)) + \lambda_1 L_{greeks}\left(\frac{\partial g(X_i)}{\partial X}, \frac{\partial g(X_i)}{\partial X}\right) + \lambda_2 L_{PDE}(\mathcal{L}g) \tag{40}$$

Where $\mathcal{L}$ denotes a PDE operator and $\lambda_1, \lambda_2 \geq 0$ are weights for each loss function. For example, in the case of black scholes, we add the differential:

$$\mathcal{L}g = \frac{\partial g}{\partial t} + rs\frac{\partial g}{\partial s} + \frac{\sigma^2 s^2}{2}\frac{\partial^2 g}{\partial s^2} - rg$$

[52] argues that the inclusion of a PDE loss term leads to self-consistency, given that if the neural network approximation satisfies the pricing PDE (in their application, the Dupire Local Volatility PDE), $\mathcal{L}g = 0$ there is no dynamic arbitrage.

[43] proposes a method to determine the relative weightings between the pricing error and the PDE error

In the most general case, we could have a complicated loss function that becomes

$$\lambda L_{price} + \lambda_1 L_{greeks} + \lambda_2 L_{PDE}(\mathcal{L}g) + \sum_{i=1}^{P} \lambda_i L_{cons_i} \tag{41}$$

Interpreting and Monitoring Neural Networks

We briefly discuss some issues with neural networks during the downstream applciations phase in 1. The first issue is the 'black-box' nature and lack of interepretability. In this case, we only use neural networks as an approximating function and overlay on top of some given market generator model and numerical method, which may alleviate some of the *black-box* issues [13]. However, the neural network may still produce unexpected outputs. We can evaluate and interpret neural networks to some extent. In the simple case, we can use graphical methods, or evaluate behaviour around boundary conditions. In addition, we can leverage machine learning interpretability methods [44], which [4] explores in the context of using deep neural networks for Heston calibration. For example, to interpret the pricing function, we could consider:

- Dependency plots against one or two dependent variables, boundary conditions.
- First order partial derivatives, Second order (Hessian), higher order derivatives.
- Output of penultimate layer (basis functions or latent representation)

- Machine-learning interpretabilty methods: Shapley Values, LIME
- Evaluating the correctness in the implementation of the Neural Network AAD versus finite differences.

If a neural network is ever deployed on a downstream application, continuous model monitoring may be needed. While the neural network may perform well for the training test, or for some market conditions, performance could deteriorate if real market conditions change. In the training period, we obtain an estimate of the pricing errors for some range of parameters. We could define a region of parameters $\mathcal{X} \subseteq \mathbb{R}^d$ for which the maximum error $\boldsymbol{e} = \mathbf{y} - g(\mathbf{X}; \boldsymbol{\theta})$ of the neural network is under some ϵ. These could be stored as simply $2d$ linear constraints $d_i < X_i < u_i$ for each parameter. If the pricer is evoked outside this region, we simply revert to the original numerical method. Another method could be to consider confidence intervals, for example with an ensemble as previously described, or simply constructing a confidence interval from the standard deviation of the neural network errors $\sigma = Var(\mathbf{e})$. If however, the market parameters have been consistently away the training region of parameters, in other words, *distribution drift*, or the contract structure of a derivative changes, then this necessitates retraining of the neural network.

Other Applications of Neural Networks for Derivatives Modelling

Neural SDEs: In the Neural SDE approach [23], we represent the dynamics of some stochastic process $\mathbf{S}_t$ with a neural network, for example consider:

$$\mathbf{S}_t = g(\mathbf{S}_t; \boldsymbol{\theta}) d\mathbf{W}_t \tag{42}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} [y_i(\mathbf{X}_i) - \mathbb{E}[h_i(\mathbf{S}_T; \mathbf{X})] \| \tag{43}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} \left[y_i(\mathbf{X}_i) - \sum_{j=1}^{N_B} h_i(\mathbf{S}_T; \mathbf{X}; W_j) \right] \| \tag{44}$$

In the above, the neural network is optimised by the calibration error between the MC Prices Neural SDE and N_C underlying options. A key advantage is that the Neural SDE approach may allow for more realistic dynamics to be captured, for example in the conext of interest rates, the neural SDE could be calibrated to all swaptions as opposed to some subset. However, in terms of their use towards pricing and obtaining sensitivities, given that Neural SDEs only produce the $\mathbf{S}_t$, Monte-Carlo is needed and hence the actual inference time may be slow. In addition, although the dynamics may be more realistic and lead to lower calibration errors, dynamics cannot be explicitly controlled through volatility model parameters as in typical volatility models, although interpretability could be addressed to some extent using the *machine learning interpretability* methods described in the previous section.

Generative Adversarial Networks (GANs) Generative Adversarial Networks involve generating synthetic samples using a neural network, based on real samples, which have had applications in image, text, and video domains among others. A challenge in finance is that there is only one observed price trajectory to draw upon, and there is the infinite-dimensional nature of time series. Some literature in this domain includes [5] [9]. A question is whether risk-neutral pricing can be done under general GANs, and some literature in this direction [8] [7]. GANs also have a connection with Neural SDEs, and Neural SDEs can be considered to be a infinite-dimensional GAN *Generative Adversarial Networks* (GANS) [35]. Like with Neural SDEs, although the simulated samples may more closely resemble real-world dynamics, there is a potential lack of explicit control and interpretability which may limit its application towards pricing. Although Neural SDEs and GANs may not necessarily be used for pricing from a regulatory / model validation standpoint, both Neural SDEs and GANs may be incredibly beneficial in that they can be used to generate more realistic scenarios for risk management applications (e.g. VaR backtesting).

Alternative Methods

Polynomial Regression / Basis Regressions: [40] [51] considered basis function regression. More recently [50] discussed the use of the Karhunone-louvre basis methods. In the case of polynomial regression Consider for example the number of terms of a k-th order polynomial regression for a N_F-dimensional input becomes $\binom{N_F+k}{k}$. The Stone-Weierstrass theorem asserts that any continuous function on a compact set can be approximated by polynomial. Thus the and

Chebyschev / Tensor Methods: [2] discussed the use of Tensor Methods as a efficient alternative for neural networks. However, implementations in `Python` do not appear to be as readily available as `Tensorflow`, although this could be a direction that is further explored.

Gaussian Processes / Kernel methods: [15] [36] and numerous other papers explored the use of Gaussian Processes, which have similar properties in that once trained, inference in terms of the pricing prediction is fast, and it is also possible to obtain quick analytical gradients. A potential advantage of Gaussian Processes is that uncertain bounds can be obtained directly. However, a key drawbakc is that naive implementations of Gaussian Process Regression have $O(N^3)$ time complexity in training [17], due to the kernel matrices needing to be inverted. This suggests that they may not necessarily scale to a large number of training points required to achieve low prediction errors. However, more advanced implementations of Gaussian Processes could be potentially explored and compared against the performance of neural networks in future works.

Tree-based Methods: Tree-based methods as a machine learning method, such as Gradient Boosted Trees or Random Forests, have lead to highly competitive results on machine learning competitions such as Kaggle. Some papers, for example [18] [16] have considered their application toward pricing, given their predictive performance, and tree-based methods are also able to attain a relatively fast inference time. However, tree-based methods are in effect linear combinations of indicator functions.

Thus for applications that require pricing sensitivities, tree-based methods are not feasible given that the tree is nowhere differentiable.

Lookup Table: [39] considered directly storing pre-computed SABR prices into a lookup table. Similar to a neural network, this would lead to very fast inference (potentially even faster), however, the neural network approximation would come with smooth interpolation (given smooth activation functions) across the input space, whereas the lookup table would require further interpolation.

To summarise, Neural Networks are advantageous, in that they are scalable to large amounts of training data, able to learn non-parametric relationships. Inference time is relatively fast after training. The disadvantage is that there is no guarantee of convergence in even the first order pricing function, or that the function approximation has the desired properties (e.g. no-arbitrage), except in few theoretical cases.

5 Numerical Experiments

We conduct our numerical experiments in a similar structure as the 1 .

Geometric Brownian Motion / Black-Scholes

[INCOMPLETE, fix table, add graphs, tidy]

Context: We consider the most simple case: European Call pricing in 1D Black-Scholes. Although the neural network approximation is trivial in this case, given the availability of the Black-Scholes formula, this example allows us to examine the behaviour of the various construction methods in a setting where we can compare against the true price .

The European call case is also particularly relevant as in modelling European Calls, we may be potentially able to model all European payoffs. *Model risk-neutral implied distribution*: Let $g(T, y/K; t, X)$ be a estimate for the conditional density at time T. Thus in effect, we approximation the price of a European digital and obtain (in the 1D case).

$$g(T-t, X) \approx DF_{t,T} \int_K^\infty (\frac{y}{K} - 1)^+ g(T, y; t, X) dy$$

SDE and MC: In the case of Black-Scholes the SDE in a forward F_t, the normalised forward $M_t = F_t/K$, and log-forward are given by:

$$\begin{aligned} dF_t &= F_t \sigma dW_t, \quad F_t = F_0 \exp(-\frac{\sigma^2}{2} t + \sigma \sqrt{t} \frac{W_t}{\sqrt{t}}) \\ dM_t &= d(F_t/K) = \frac{F_t}{K} \sigma dW_t = M_t \sigma dW_t, M_0 = \frac{F_0}{K} \\ d\log(M_t) &= d\log(F_t) = \frac{-\sigma^2}{2} dt + \sigma W_t \\ h(M_T) &= (M_T - 1)^+ = \left(\frac{F_T}{K} - 1\right)^+ = (\exp(\log(M_T) - 1)^+ \end{aligned}$$

In the case of Black-Scholes, we can simulate the SDE of F_t exactly. However, we do not necessarily need to simulate the SDE to obtain MC prices for the dataset in this case, as we can leverage the closed form price.

PDE: Let $m = \log(M)$ denote the log-moneyness. We note that

$$\frac{\partial \sigma \sqrt{\tau}}{\partial \tau} = \frac{\sigma}{2\sqrt{\tau}} = \frac{\sigma^2}{2\sigma\sqrt{\tau}}, \quad \frac{\partial m}{F} = \frac{\partial m}{F} = \frac{1}{F} \tag{45}$$

$$\frac{\partial}{\partial F}(\frac{\partial g}{\partial m}\frac{\partial m}{\partial F}) = \frac{-1}{F^2}\frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}\frac{1}{F^2} \tag{46}$$

The corresponding Black-Scholes PDE is given by:

$$0 = -g_\tau - \frac{\sigma^2}{2} g_m + \frac{\sigma^2}{2} g_{mm}, g(\tau, m) = (\exp(m) - 1)^+ \tag{47}$$

$$0 = \frac{\sigma^2}{2}[-g_{\sigma\sqrt{\tau}} \frac{1}{\sigma\sqrt{\tau}} - g_m + g_{mm}], g(\tau, m) = (\exp(m) - 1)^+ \tag{48}$$

$$0 = \frac{\partial g}{\partial \sigma\sqrt{\tau}} \frac{1}{\sigma\sqrt{\tau}} - \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}, \quad g(\tau, m) \tag{49}$$

Although we do not need to solve the Black-Scholes PDE to obtain prices for the dataset, we derive the PDE in $\sigma\sqrt{\tau}, m$ by hand, so that we can use it for the Neural PDE approach. A question is whether we should multiply we should multiply $-\frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}$ by , as it could lead to different training behaviour. In addition, we have the new no-arbitrage bounds:

$$\frac{\partial g}{\partial \tau} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}} \frac{\sigma}{2\sqrt{\tau}} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}} > 0 \tag{50}$$

$$\frac{\partial g}{\partial K} < 0 \implies \frac{\partial g}{\partial m} \frac{-1}{K} < 0 \implies \frac{\partial g}{\partial m} > 0 \tag{51}$$

$$\frac{\partial^2 g}{\partial K^2} > 0 \implies \frac{1}{K^2} \frac{\partial g}{\partial K} + \frac{\partial^2 g}{\partial m^2} \frac{1}{K^2} < 0 \implies \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2} > 0 \tag{52}$$

$$(\exp(m) - 1)^+ \leq g(\sigma\sqrt{\tau}, m) \leq \exp(m) \tag{53}$$

$$\lim_{m \to -\infty} g(\sigma\sqrt{\tau}, m) = 0, \lim_{m \to \infty} = \exp(m) \tag{54}$$

$$\lim_{\sigma\sqrt{\tau} \to 0} g(\sigma\sqrt{\tau}, m) = (\exp(m) - 1)^+ \tag{55}$$

Convexity in strike $\frac{\partial^2 g}{\partial K^2} > 0$ is satisfied, for example if we let $\frac{\partial^2 g}{\partial m^2} > 0$ as well. In addition, we could also consider the no-arbitrage constraints that arise when

$$\frac{\partial g}{\partial K} = \mathbb{E}^{\mathbb{Q}}[1_{S_T < K} - 1] = \mathbb{Q}[S_T < K] - 1$$

Thus $\lim_{K \to \infty} \to \frac{\partial g}{\partial K} dK = 0, \lim_{K \to 0} \to \frac{\partial g}{\partial K} = -1$

$$\int_0^\infty \frac{\partial^2 g}{\partial K^2} = 1$$

Closed Form: In this case, we can leverage the closed form solution given by the Black-Scholes Formula. In addition, we also exploit the first-order positive homogeneity homogeneity in the underlying and strike, as in [32] [TODO: add the other references, this trick is used in many papers]. Thus we can eliminate one of F_t, K by letting $\lambda = \frac{1}{K}$ and fixing. We can further eliminate one of σ, τ by noting that in the Black-Scholes formula: The volatility σ, and time-to-maturity $\tau = T - t$ parameters are grouped together in the closed form. Thus we can exploit this time-scaling property to consider $\sigma\sqrt{\tau}$ instead of σ, τ. This presents a potential pre-processing step that we can use to

$$d_{\pm} = \frac{\log(F_t/K)}{\sigma\sqrt{\tau}} \pm \frac{(\sigma\sqrt{\tau})^2}{2}$$
$$C(\frac{F_t}{K}, 1, \sigma, \tau) = \frac{F_t}{K}\Phi(d_+) - \Phi(d_-)$$

$$\mathbb{E}^{\mathbb{Q}}[(\frac{F_t}{K} - 1)^+] = \frac{\mathbb{E}^{\mathbb{Q}}[(F_T - K)^+ | S_t, K, \sigma]}{K} \tag{56}$$
$$\lambda C(F_t, K, \sigma, \tau) = C(\lambda F_t, \lambda K, \sigma, \tau) \tag{57}$$

Dataset: We generate $N_{train} = 2^{16} = 65,536$ samples from the parameter space in table 5 to use for training the neural networks, and an independent $N_{test} = 2^{16} = 65,536$ from the same parameter space to use as a testing dataset. We sample from $\sigma\sqrt{\tau}, m$ uniformly over a range, and independently of one another, and compute the corresponding closed form call prices $\mathbf{y} = f(\sigma\sqrt{\tau}, m)$ using the Black-Scholes formula. Using the implementation of automatic differentiation using the `Jax` library in Python, we obtain efficiently obtain first-order differentiaals $\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$ for the differential method.

	Number of Samples	$m = \log(F/K)$ Log-Moneyness	$\sigma\sqrt{\tau}$ (Time-scaled Implied Volatility)
Train	65,536	Uniform $[-3, 3]$	Uniform $[0, 3]$
Test	65,536	Uniform $[-3, 3]$	Uniform $[0, 3]$

Table 3: parameter space for the Black-Scholes European calls example

Results: We obtain the following results on the testing dataset

Rough Bergomi

[INCOMPLETE, restructure and fix graphs]

In the rough volatility setting, there is a lack of closed form expressions for prices, hence prices need to be approximated by Monte Carlo. [30] used a two step approach.

MC and SDE:

The Rough Bergomi model has dynamics:

$$1 \tag{58}$$

We use the code implementation for the Turbocharged Rough Bergomi scheme [1] from [41]

Experiment: Our parameter space consists of:

[1]Accessed from: https://github.com/ryanmccrickerd/rough_bergomi

$\mathbf{S_0}$	$\log(K)$	τ	α	ρ	V_0	ξ
Underlying	Strike	Time-to-maturity	Hurst Exponent	correlation	Volatility	Vol-of-vol
$\{1\}$	$\{-0.5 + 0.1i : i = 0, \ldots, 10\}$	$\{\frac{i}{30} : i = 0, \ldots, 30\}$	$U[-0.4, 0.5]$	$U[-0.95, -0.7]$	$\{0.235\}$	$U[1.5, 2.5]$

Table 4: Rough Bergomi Parameter Space

We can only analyse error in pricing and greeks with respect to the MC estimates. However, we can analyse no-arbitrage violations for calls as before.

We consider two neural network architectures: a standard feed-forward **??**, a gated neural network **??**, and also compare against cubic spline regression.

- **Inputs**: $\mathbf{X} = (-\log(K), \tau, \alpha, \rho, \xi)$
- **Outputs**: Predicted call price $\approx \mathbb{E}[(\mathbf{S}_T - K)^+ | \mathbf{X}]$
- **Architecture**: *Feed-Forward*: 2 hidden layers with 100 hidden units each, hidden layers, linear output activation; *Gated*:
- **Training:** Batch Size of 32, Learning Rate: 10^{-3}. For feed-forward only, Early Stopping with a validation set of 20%, and patience of 10 epochs.
- **Regularisation** : None.

We sample $N_{\text{SPACE}} = 100$ random vectors of (α, ρ, ξ) from the distributions above. For each of these, we simulate S_T using [41], with a terminal maturity of $T = 1$ and $N_{\text{Brownian}} = 10000$ paths each, and compute the call prices for the collection of $\tau \times \log(K)$ described above. This produces a dataset of 96100 observations in 18 seconds.

Results We obtain the following results:

	ffn	gated	polynomial	
l1	0.02016	0.015743	0.011844	0.032786
l2	0.025676	0.01918	0.015266	0.046717
l_inf	0.13678	0.088745	0.058258	0.21013

Table 5: Rough Bergomi Example - Prediction Error

	ffn	gated	polynomial	
lower__bound__violation	0.46441	0.46959	0.45438	0.47755
upper__bound__violation	0	0	0	0
monotonicity__error	0.20365	0.10905	0	0.1038
time__value__error	0.50503	0.26101	0	0.3874
convex__error	0.15054	0.38567	0	0.18967

Table 6: Rough Bergomi Example - No-Arbitrage Errors

	ffn	gated	polynomial	
model__parameters	11621	11621	22550	11627
inference__time	1.9441	1.8878	168.06	1.5024

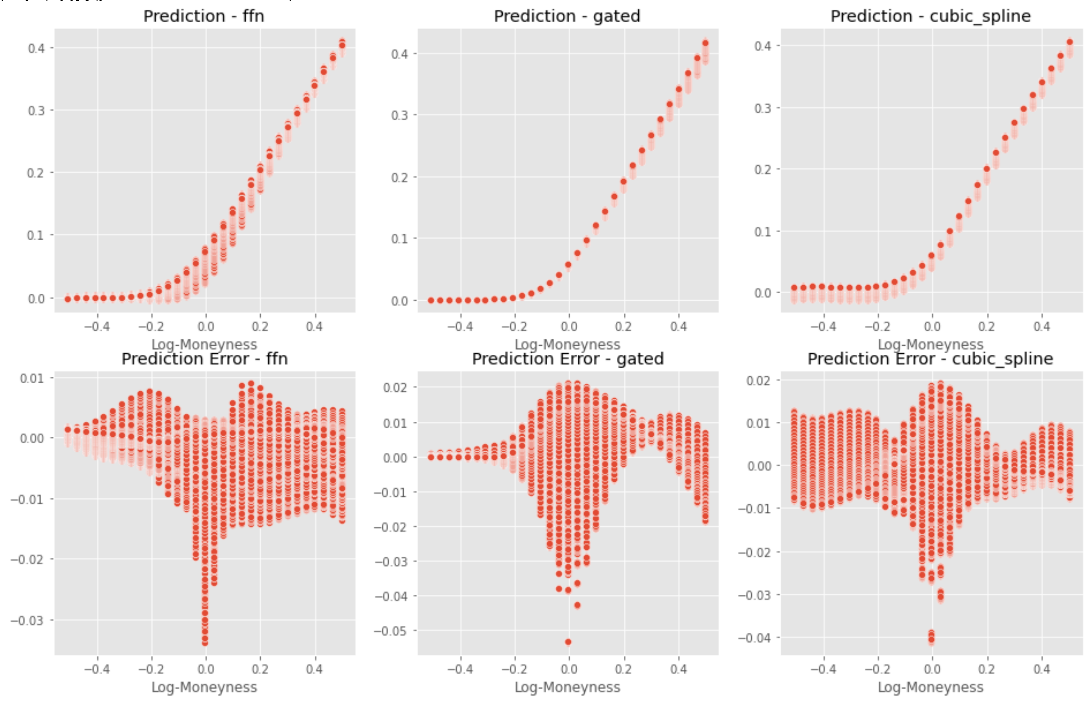


Figure 2: Errors for Rough Bergomi Approximations

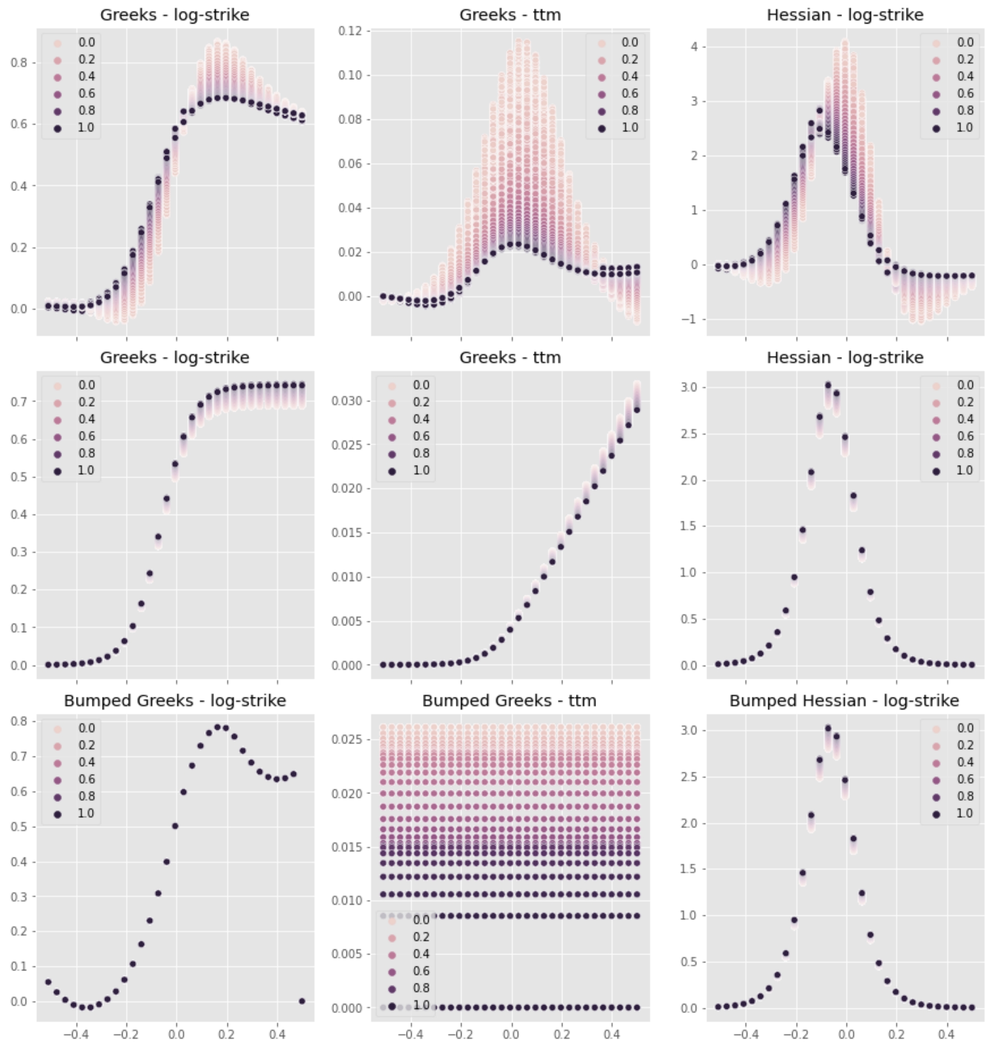


Figure 3: Greeks for Feed-foward, Gated, Cubic-spline respectively

The gated neural network is by construction able to avoid any no-arbitrage in terms of monotonicity in time to maturity in strike, and convexity in strike. The standard neural network however achieves the best result in terms of prediction error. Interestingly, standard cubic spline regression has similar performance to the unconstrained neural network, at a much shorter training time.

Remark: To verify that the neural network pricer has minimal dynamic arbitrage opportunities in this non-Markovian setting, one approach could be to evaluate whether the neural network pricer satisfies the corresponding path-dependent PDE (PPDE). Some literature towards this includes [34] and [48]

Basket Option - Arithmetic Brownian Motion

[INCOMPLETE, restructure and fix graphs]

Setup: We consider the first example from [31], a standard European Call on a basket option where the underlyings have dynamics Arithmetic Brownian Motion. In this setting, we *pretend* we do not know the true analytic formula, and instead use the neural network as an on-the-fly solver to learn from sample payoffs, similar to Longstaff-Schwartz formulation. This allows us to analyse the potential for using the neural network. We benchmark the various neural network constructions against polynomial regresion and running a Monte Carlo on each path. We consider a similar setup as the first code example [2] from [31]. time-to-maturity $\tau = T - t$, strike K, and the covariance $\mathbf{L}\mathbf{L}^\top$ are fixed, and the experiment is repeated with a different dimensionality d and number of sample paths.

SDE and MC: Let $\mathbf{S}_t$ be a d-dimensional Arithmetic Brownian motion driven by $\mathbf{W}_t$ a f-dimensional Brownian Motion over $t \in [0, T]$, with covariance $\mathbf{L}\mathbf{L}^\top dt, \mathbf{L} \in \mathbb{R}^{d\times f}$. For simplicity suppose that $\mathbf{S}_t$ is a $\mathbb{Q}$-local martingale. Suppose there is some weight vector $\mathbf{w} \in \mathbb{R}^d$ representing the index weights for a basket option, such that the value of the underlying basket is $\mathbf{w}^\top\mathbf{S}_t$. Then the payoff of the basket call option with payoff $h(\mathbf{w}^\top\mathbf{S}_t, K) = (\mathbf{w}^\top\mathbf{S}_t - K)^+$ is given by:

$$d\mathbf{S}_t = \mathbf{L}d\mathbf{W}_t, \mathbf{S}_t = \mathbf{S}_0 + \mathbf{L}\mathbf{W}_t, \mathbf{S}_t \sim N(\mathbf{S}_0, \mathbf{L}\mathbf{L}^\top t) \tag{59}$$

$$d\mathbf{X}_t = \mathbf{w}^\top d\mathbf{S}_t = \sigma dW_t^\top, \mathbf{X}_t = \mathbf{w}^\top\mathbf{S}_t, \mathbf{X}_t \sim N(\mathbf{w}^\top\mathbf{S}_0, \mathbf{w}^\top\mathbf{L}\mathbf{L}^\top\mathbf{w}) \tag{60}$$

$$h(\mathbf{w}, \mathbf{S}_t, K) = (\mathbf{w}^\top\mathbf{S}_T - K)^+ = (\mathbf{X}_T - K)^+ \tag{61}$$

$$f(\tau, \mathbf{S}_t, K, \mathbf{w}) = \mathbb{E}[(\mathbf{w}^\top\mathbf{S}_t - K)^+|\mathbf{S}_t, \mathbf{w}, \tau, K] \tag{62}$$

Remark: The Bachelier Model has a positive homogeneous relationship. In particular:

$$f(x, \sigma\sqrt{\tau}) = \mathbb{E}[(x-K)^+] = K\mathbb{E}[(\frac{x}{K} - 1)^+] = Kf(\frac{x}{K}, \sigma\sqrt{\tau}/K)$$

Thus we could possibly eliminate dependency on strike. In addition, we can eliminate the dependence on time-to-maturity τ by noting that the σ and τ terms are grouped together as $\sigma\sqrt{\tau}$.

We note that if we consider the weighted underlying w_iS_i, we are effectively as a single variable, we are effectively able to price. However, in this setup we, the neural learns a pricing function the specific strike K, maturity τ, and covariance $\mathbf{L}\mathbf{L}^\top$.

PDE: Applying Ito's lemma on $\mathbf{X}_t$ and $\mathbf{S}_t$, the corresponding PDE for the price of the basket call option is given by:

[2]Retrieved from: https://github.com/differential-machine-learning/notebooks/blob/master/DifferentialMLTF2.i

$$df(T-t, S_t; \dots) = -\frac{\partial f}{\partial t}dt + \sum_{i=1}^{d} \frac{\partial f}{\partial S_i} dS_{i,t} + \sum_{i=1}^{d}\sum_{j=1}^{d} \frac{1}{2}\frac{\partial V}{\partial S_i S_j} d[S_i, S_j] \tag{63}$$

$$0 = -\frac{\partial f}{\partial \tau} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} \frac{\partial V}{\partial S_i S_j}, f(0, \mathbf{S}) = (\mathbf{w}^\top \mathbf{S} - K)^+ \tag{64}$$

$$df(T-t, X_t; \dots) = -\frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dX_t + \frac{1}{2}d[X]_t \tag{65}$$

$$= \mathbf{w}^\top \mathbf{L} d\mathbf{W}_t + \frac{1}{2}\mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w} dt \tag{66}$$

$$= -\frac{\partial f}{\partial \tau} + \frac{\sigma^2}{2}\frac{\partial f}{\partial x^2}, f(0, \mathbf{X}) = (\mathbf{X} - K)^+ \tag{67}$$

Closed form price: The closed form price is given by the Bachelier normal formula:

For a gaussian random variable with variance a^2 and mean b, We can write:

$$aZ + b - K = a(Z + \frac{b-K}{a}) \geq 0 \implies Z \geq \frac{K-b}{a}$$
$$\mathbb{Q}[aZ + b \geq K] = \Phi(\frac{K-b}{a})$$
$$\mathbb{E}[Z \cdot 1_{aZ+b\geq K}] = \int_{(K-b)/a}^{\infty} z\frac{e^{-z^2/2}}{\sqrt{2\pi}}dz = [\frac{-1}{\sqrt{2\pi}}e^{-z^2/2}]_{(K-b)/a}^{\infty} = \frac{\exp(-((K-b)/a)^2/2)}{\sqrt{2\pi}}$$
$$\mathbb{E}[(aZ + b - K)^+] = a\phi(\frac{K-b}{a}) + (b-K)\Phi(\frac{K-b}{a})$$
$$= a\phi(\frac{b-K}{a}) + \frac{b-K}{a}\Phi((\frac{b-K}{a}))]$$

In this case, we have $a = \sigma\sqrt{\tau} = \sqrt{\mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w}}\sqrt{\tau}, b = \mathbf{w}^\top \mathbf{S}_0$. Thus the closed-form price for the basket option is given by:

$$f(\mathbf{X}_0, \sigma, \tau, K) = \sigma\sqrt{\tau}[\phi(d_1) + d_1\Phi(d_1)], d_1 = \frac{\mathbf{X}_0 - K}{\sigma\sqrt{\tau}} \tag{68}$$

The analytic greeks are given by, noting that $\frac{\partial d_1}{\partial \tau} = -\frac{d_1}{2\tau}$ and $\frac{\partial d_1}{\partial x} = \frac{1}{\sigma\sqrt{\tau}}$

$$\frac{\partial f}{\partial x} = \sigma\sqrt{\tau}[-\frac{d_1}{\sigma\sqrt{\tau}}\phi(d_1) + \frac{1}{\sigma\sqrt{\tau}}\Phi(d_1) + \phi(d_1)\frac{d_1}{\sigma\sqrt{\tau}}] = \Phi(d_1) \tag{69}$$

$$\frac{\partial f}{\partial \tau} = \frac{f}{2\tau} + \sigma\sqrt{\tau}[\frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1}{2\sqrt{\tau}}\Phi(d_1)] = \frac{\sigma}{2\sqrt{\tau}}\phi(d_1) \tag{70}$$

$$\frac{\partial f}{\partial x^2} = \frac{1}{\sigma\sqrt{\tau}}\phi(d_1) \tag{71}$$

Neural Networks: We use the same neural network architecture and random seed initialisation for the weights for the feed-forward and differential neural network:

- **Inputs**: the values of the underlying $\mathbf{S}_0 \in \mathbb{R}^d$
- **Outputs**: predicted price for strike $K = 1$ $f(\mathbf{S}_0) \approx \mathbb{E}[(\mathbf{w}^\top \mathbf{S}_T - 1)^+ | \mathbf{S}_0]$
- **Architecture**: Standard feed-forward, 4 Hidden layers of 30 hidden units each, softplus activation in all hidden layers, linear output activation
- **Training:** Batch Size of 32, Learning Rate: 10^{-3}. For feed-forward only, Early Stopping with a validation set of 20%, and patience of 50 epochs.
- **Regularisation** : None.

In the neural network case, we do not have an estimate for $\frac{\partial g}{\partial \tau}$. Thus we only verify the pricing error, percentage of no-arbitrage bound violations, and error in the estimate of the sensitivity to the basket factor.

Dataset: We compare a standard feed-forward neural network trained on the sample payoffs only 36, and a neural network trained on the pathwise differentials 39, and standard Monte Carlo estimation. We also compute the analytic prices and greeks from equations 45-48. We generate samples $i = 1, \ldots, N_{\text{samples}} = 10,000$ samples of $\mathbf{S_{i,0}}, \mathbf{W}_{i,0} \in \mathbb{R}^d$, simulating $\mathbf{S_{i,T}} = \mathbf{S}_{i,t} + \mathbf{L}\mathbf{W_{i,T}}$ exactly from 61. We obtain pathwise differentials $\frac{\partial h}{\partial \mathbf{S}_{i,T}}$ via automatic differentiation, although in this case the pathwise differentials are known analytically: $\mathbf{w}^\top 1_{\mathbf{X}_T \geq 1.0}$. Given the pathwise differentials are not differentiable, we cannot obtain a Hessian estimate from Monte Carlo with AAD.

In our setup, we consider the following parameter space. We simulate a random covariance matrix by taking a Cholesky decomposition of a single posiive-definite sample of matrix of standard normal random variables. $\mathbf{L}\mathbf{L}^\top = 0.2\mathbf{Z}^\top\mathbf{Z}, Z_{i,j} \sim N(0,1)$

d (No. Assets)	τ (Time-to-Maturity)	K (Strike)	$\mathbf{w}$ (Basket Weights)	$\mathbf{L}$ (Covariance)
100	$\{1\}$	$\{1\}$	$\{\frac{1}{d}\mathbf{1}\}$	$\{\mathbf{L} : \mathbf{L}\mathbf{L}^\top = 0.2\mathbf{Z}^\top\mathbf{Z}$, $det(\mathbf{Z}^\top\mathbf{Z}) > 0\}$ $Z_{ij} \sim N(0,1)$

Table 8: Fixed Parameters for Basket Example

	No. Samples	S_t *Underlying*
Train	10,000	$1 + \sqrt{30/250}\mathbf{Z}$
Test	10,000	$1 + \sqrt{30/250}\mathbf{Z}$

Table 9: Sample Parameter Space for Basket Example

Results: We obtain the following results.

	CV	FFN	DiffNet	ResNet	PolyReg	MC
MAE	9.215×10^{-2}	9.694×10^{-2}	5.386×10^{-3}	7.228×10^{-2}	1.023×10^{-1}	7.031×10^{-4}
RMSE	1.177×10^{-1}	1.245×10^{-1}	7.128×10^{-3}	9.261×10^{-2}	1.292×10^{-1}	7.353×10^{-4}
Max	5.043×10^{-1}	5.920×10^{-1}	4.998×10^{-2}	5.097×10^{-1}	5.352×10^{-1}	1.041×10^{-3}

Table 10: Bachelier Basket Example - Prediction Errors

In Table **??**, Max denotes the maximum error in the testing dataset, MAE denotes the average absolute error in the dataset, and RMSE denotes the root-mean-squared error in the dataset. The best performing neural network approach in this case is this Differential approach, which outperforms the other neural network constructions, and unlike previously, the polynomial regression as well. Although the DiffNet outperforms the other neural networks and polynomial regression by an order of 10 in pricing errors, in this case it has an order of $\times 10$ greater pricing errors compared to a Monte Carlo pricing approach.

	CV	FFN	DiffNet	ResNet	PolyReg	MC
MAE	2.275×10^{-3}	2.820×10^{-3}	6.036×10^{-4}	2.257×10^{-3}	1.346×10^{-1}	3.017×10^{-5}
RMSE	2.883×10^{-3}	3.531×10^{-3}	7.623×10^{-4}	2.816×10^{-3}	1.724×10^{-1}	3.437×10^{-5}
Max	1.292×10^{-2}	1.481×10^{-2}	4.639×10^{-3}	1.130×10^{-2}	1.027	6.816×10^{-5}

Table 11: Bachelier Basket Example - Gradient Errors

Table 11 displays the errors in the gradient for the basket factor, which should simply be the average of all first-order gradients $\mathbf{w}^\top \frac{\partial g}{\partial \mathbf{X}}$. The DiffNet is the best performing neural network approach in terms of gradients error, although this may likely be because the differential approach is trained to minimise the error in the gradients as well. However, the DiffNet also has a $\times 10$ error in the gradient compared to Monte Carlo.

	CV	FFN	DiffNet	ResNet	PolyReg	MC
Pred LowerBound	3.109×10^{-1}	3.182×10^{-1}	6.280×10^{-3}	2.649×10^{-1}	3.216×10^{-1}	0
Grad LowerBound	5.622×10^{-2}	1.246×10^{-1}	4.000×10^{-5}	5.514×10^{-2}	4.860×10^{-1}	0

Table 12: Bachelier Basket Example - No-Arbitrage Violations

Table 12 depicts the no-arbitrage errors for this example. We are unable to verify the accuracy in no-arbitrage call relation with respect to time-to-maturity for the neural network and regression approaches, given that τ is not included as a variable. On the other hand, we cannot verify the convexity of the Monte Carlo pricing approach, given that pathwise call payoffs $(\mathbf{w}^\top S_{i,T} - K)^+$ are not twice-differentiable, although we could consider payoff smoothing. Again, the differential approach produces the lowest errors, but there is zero no-arbitrage violation using MC pricing.

	CV	FFN	DiffNet	ResNet	PolyReg	MC
InfTime	7.288×10^{-2}	$\mathbf{6.337 \times 10^{-2}}$	6.739×10^{-2}	7.450×10^{-2}	52.41	131.4
TrainTime	15.64	**15.14**	18.09	16.51	22.9	-
TotalTime	15.71	**15.20**	18.16	16.59	75.32	131.4
Params	20902	20901	20901	20901	5151	-

Table 13: Bachelier Basket Example - Time Complexity

Finally, table 13 depicts the relative complexity in terms of the number of parameters of the model, and the training and inference time required. In the on-the-fly setting, the true time required is (denoted TotalTime). In this case, the best-performing DiffNet has a roughly 7-fold speedup versus the MC pricing approach.

Remarks: As per the hypothesis of [31], the differential neural network achieves lower errors in both the price approximation, and gradients. We should note that in this case, it may be more appropriate to model the basket value $\mathbf{w}^\top \mathbf{S}_t$, which is also an Arithemtic Brownian Motion, as a univariate process, and then apply the chain rule on $\frac{\partial \mathbf{w}^\top \mathbf{S}_t}{\partial S_{i,t}}$. However, although in this case, there is a neural network construction that outperforms polynomial regression. In this case, it appears. . At least for our particular choice of sample seed and the given neural network architecture, all neural network approaches underperform a MC estimate in all error measures. However, this example may be still too simple given the problem is one-step, and the terminal distribution can be simulated exactly.

6 Conclusion

[INCOMPLETE] To summarise the resuls:

- Dataset quality and construction likely plays a role in training, convergence, and extrapolation. Cleaner (in terms of estimation error for MC and PDEs) and more data is better, but this increases training time.
- Differential training (whether through explicit greek labels, or the neural PDE term) seems to lead to better results empirically
- Hard constraints can be embedded into a neural network to prevent some no-arbitrage conditions, at the cost of some flexibility.
- Neural networks can be trained to solve complex payoffs and high-dimensional problems / market models, but need to be retrained on new market parameters or contract specifications. In some cases, it can be slower and less accurate than even Monte Carlo.

Future Directions: Further investigation into Neural PDE methods. Comparison against the tensor methods of [2], Potz, Glau, as no convenient `Python` implementation exists yet. Investigation into Quantum Deep Learning and potential applications for derivatives modelling. Investigation of time-varying real world measure of dynamics, of how calibrated parameters evolves udner real world dynamics. Deep Hedging: Learning Risk-Neutral Implied Volatility Dynamics

References

[1] Alexandre Antonov, Michael Konikov, and Vladimir Piterbarg. "Neural networks with asymptotics control". In: *Available at SSRN 3544698* (2020).

[2] Alexandre Antonov and Vladimir Piterbarg. "Alternatives to Deep Neural Networks for Function Approximations in Finance". In: *Available at SSRN 3958331* (2021).

[3] Erik Alexander Aslaksen Jonasson. *Differential Deep Learning for Pricing Exotic Financial Derivatives.* 2021.

[4] Damiano Brigo et al. "Interpretability in deep learning for finance: a case study for the Heston model". In: *Available at SSRN 3829947* (2021).

[5] Hans Buehler et al. "A data-driven market simulator for small data environments". In: *arXiv preprint arXiv:2006.14498* (2020).

[6] Hans Buehler et al. "Deep hedging". In: *Quantitative Finance* 19.8 (2019), pp. 1271–1291.

[7] Hans Buehler et al. "Deep Hedging: Learning Risk-Neutral Implied Volatility Dynamics". In: *arXiv preprint arXiv:2103.11948* (2021).

[8] Hans Buehler et al. "Deep Hedging: Learning to Remove the Drift under Trading Frictions with Minimal Equivalent Near-Martingale Measures". In: *arXiv preprint arXiv:2111.07844* (2021).

[9] Hans Buehler et al. "Generating financial markets with signatures". In: *Available at SSRN 3657366* (2020).

[10] Peter Carr and Dilip B Madan. "A note on sufficient conditions for no arbitrage". In: *Finance Research Letters* 2.3 (2005), pp. 125–130.

[11] Marc Chataigner, Stéphane Crépey, and Matthew Dixon. "Deep local volatility". In: *Risks* 8.3 (2020), p. 82.

[12] Francois Chollet. *Deep learning with Python.* Simon and Schuster, 2021.

[13] Samuel N Cohen, Derek Snow, and Lukasz Szpruch. "Black-box model risk in finance". In: *Available at SSRN 3782412* (2021).

[14] Samuel N. Cohen, Christoph Reisinger, and Sheng Wang. "Detecting and Repairing Arbitrage in Traded Option Prices". In: *Applied Mathematical Finance* 27.5 (Sept. 2020), pp. 345–373. DOI: `10.1080/1350486x.2020.1846573`. URL: `https://doi.org/10.1080%2F1350486x.2020.1846573`.

[15] Stéphane Crépey and Matthew Dixon. "Gaussian process regression for derivative portfolio modeling and application to CVA computations". In: *arXiv preprint arXiv:1901.11081* (2019).

[16] Jesse Davis et al. "Gradient boosting for quantitative finance". In: *Journal of Computational Finance* 24.4 (2020).

[17] Matthew F Dixon, Igor Halperin, and Paul Bilokon. *Machine learning in Finance.* Vol. 1170. Springer, 2020.

[18] Zineb El Filali Ech-Chafiq, Pierre Henry-Labordere, and Jérôme Lelong. "Pricing Bermudan options using regression trees/random forests". In: *arXiv preprint arXiv:2201.02587* (2021).

[19] Ryan Ferguson and Andrew Green. "Deeply Learning Derivatives". In: *arXiv preprint arXiv:1809.02233* (2018).

[20] Hans Föllmer and Alexander Schied. "Stochastic finance". In: *Stochastic Finance*. de Gruyter, 2016.

[21] Hideharu Funahashi. "Artificial neural network for option pricing with and without asymptotic correction". In: *Quantitative Finance* 21.4 (2021), pp. 575–592.

[22] Gunnlaugur Geirsson. *Deep learning exotic derivatives*. 2021.

[23] Patryk Gierjatowicz et al. "Robust pricing and hedging via neural SDEs". In: *Available at SSRN 3646241* (2020).

[24] Kathrin Glau and Linus Wunderlich. "The deep parametric PDE method: application to option pricing". In: *arXiv preprint arXiv:2012.06211* (2020).

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[26] Patrick S Hagan et al. "Managing smile risk". In: *The Best of Wilmott* 1 (2002), pp. 249–296.

[27] Horace He. "Making Deep Learning Go Brrrr From First Principles". In: (2022). URL: `https://horace.io/brrr_intro.html`.

[28] Andres Hernandez. "Model calibration with neural networks". In: *Available at SSRN 2812140* (2016).

[29] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks". In: *Neural networks* 3.5 (1990), pp. 551–560.

[30] Blanka Horvath, Aitor Muguruza, and Mehdi Tomas. "Deep learning volatility". In: *Available at SSRN 3322085* (2019).

[31] Brian Norsk Huge and Antoine Savine. "Differential machine learning". In: *Available at SSRN 3591734* (2020).

[32] James M Hutchinson, Andrew W Lo, and Tomaso Poggio. "A nonparametric approach to pricing and hedging derivative securities via learning networks". In: *The journal of Finance* 49.3 (1994), pp. 851–889.

[33] Andrey Itkin. "Deep learning calibration of option pricing models: some pitfalls and solutions". In: *arXiv preprint arXiv:1906.03507* (2019).

[34] Antoine Jack Jacquier and Mugad Oumgari. "Deep PPDEs for rough local stochastic volatility". In: *Available at SSRN 3400035* (2019).

[35] Patrick Kidger et al. "Neural sdes as infinite-dimensional gans". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 5453–5463.

[36] Joerg Kienitz, Nikolai Nowaczyk, and Nancy Qingxin Geng. “Dynamically Controlled Kernel Estimation”. In: *Available at SSRN 3829701* (2021).

[37] Tae-Kyoung Kim, Hyun-Gyoon Kim, and Jeonggyu Huh. “Large-scale online learning of implied volatilities”. In: *Expert Systems with Applications* 203 (2022), p. 117365.

[38] Yann A LeCun et al. “Efficient backprop”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[39] Mahir Lokvancic. “Machine learning SABR model of stochastic volatility with lookup table”. In: *Available at SSRN 3589367* (2020).

[40] Francis A Longstaff and Eduardo S Schwartz. “Valuing American options by simulation: a simple least-squares approach”. In: *The review of financial studies* 14.1 (2001), pp. 113–147.

[41] Ryan McCrickerd and Mikko S Pakkanen. “Turbocharging Monte Carlo pricing for the rough Bergomi model”. In: *Quantitative Finance* 18.11 (2018), pp. 1877–1886.

[42] William McGhee. “An artificial neural network representation of the SABR stochastic volatility model”. In: *Journal of Computational Finance* 25.2 (2020).

[43] Remco van der Meer, Cornelis W Oosterlee, and Anastasia Borovykh. “Optimally weighted loss functions for solving pdes with neural networks”. In: *Journal of Computational and Applied Mathematics* 405 (2022), p. 113887.

[44] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[45] Nikolai Nowaczyk et al. “How deep is your model? Network topology selection from a model validation perspective”. In: *Journal of Mathematics in Industry* 12.1 (2022), pp. 1–19.

[46] Antal Ratku and Dirk Neumann. “Derivatives of feed-forward neural networks and their application in real-time market risk management”. In: *OR Spectrum* (2022), pp. 1–19.

[47] Johannes Ruf and Weiguan Wang. “Neural networks for option pricing and hedging: a literature review”. In: *arXiv preprint arXiv:1911.05620* (2019).

[48] Marc Sabate-Vidales, David Šiška, and Lukasz Szpruch. “Solving path dependent PDEs with LSTM networks and path signatures”. In: *arXiv preprint arXiv:2011.10630* (2020).

[49] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[50] Valentin Tissot-Daguette. *Projection of Functionals and Fast Pricing of Exotic Options*. 2021. DOI: 10.48550/ARXIV.2111.03713. URL: https://arxiv.org/abs/2111.03713.

[51] John N Tsitsiklis and Benjamin Van Roy. “Regression methods for pricing complex American-style options”. In: *IEEE Transactions on Neural Networks* 12.4 (2001), pp. 694–703.

[52] Zhe Wang, Nicolas Privault, and Claude Guet. “Deep self-consistent learning of local volatility”. In: *Available at SSRN 3989177* (2021).

[53] Yongxin Yang, Yu Zheng, and Timothy Hospedales. “Gated neural networks for option pricing: Rationality by design”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.