Neural Networks for Derivatives Pricing and Risks

Candidate No:1062797

University of Oxford

A thesis submitted in partial fulfillment of the MSc in

Mathematical and Computational Finance

June 30, 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 an end-user in an investment bank or market maker. An end-user uses a given pricing function to obtain the price(s) for a given derivatives product(s) and parameter(s), and then uses 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 derivative price 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. Some other potential use cases are depicted in Table 1.

	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 a few market models and payoffs (e.g. the Black-Scholes formula and the Heston characteristic transform), which leaves MC and PDE methods for many other market models and payoffs. MC and PDE are themselves numerical methods with an approximation error to the 'true' model price. A controlled trade-off between a requisite level of error and time and computational effort can still be obtained, given explicit convergence rates for many settings.

However, several aspects 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 and - occasionally - second-order sensitivities [47]. This is because the sensitivities represent exposures to the input parameters, as well as the

potential hedging strategy (delta). In some cases, higher-order sensitivities also need to be hedged or accounted for (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 MC and PDE settings, a solution can be obtained over the state and time discretisation. If however the volatility model or payoff parameters change, the numerical method must be re-invoked (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).

An approach to improve speed may be to consider function approximation methods, namely 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. The ideal approximating functions should satisfy several properties. Firstly, the inference method should be faster than the original method, and ideally have a comparable pricing error. Secondly, the approximation must be sufficiently smooth and differentiable for sensitivities to be obtained, and the error in these gradients should ideally be low as well. Lastly, a requisite unique to derivatives modelling is that a pricing approximation must also adhere to no-arbitrage constraints. If violated, the pricing approximation could produce 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 trade-offs between the criterion of interest: pricing error, gradient errors, no-arbitrage adherence, speed, and how these contribute to the 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 Table 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 if the model 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, whilst pricing accuracy is important, some small error in pricing is acceptable, as mispricings may not be well-exploited for illiquid products. However, accurate gradients may be needed in this case for use as the hedging strategy, and to reliably 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 is one approach that allows for a trade-off between these errors, with additional desirable properties. Theoretically, neural networks have universal approximation, and, empirically, an ability to 'learn' non-parametric data-driven relationships. Sensitivities can be obtained quickly with automatic differentiation in an overall very fast inference time. Finally, neural networks can likely overcome the

curse-of-dimensionality, and 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. In the recent, a vast literature on the application of neural networks toward derivatives modelling has emerged. In terms of industrial applications, [19] of RiskFuel / Scotiabank, and [32] 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: The aim of this dissertation is to explore and evaluate neural networks for derivatives pricing and risks in several contexts:

- 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 methods from 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.

Pricing as Conditional Expectation

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 and driving factors. For simplicity, assume that $\mathbf{S}_t$ is an Ito diffusion, although we could also consider Lévy or other processes. We write $\mathbf{S}_t$ as a shorthand $\mathbf{S}_t(\mathbf{X})$, as the evolution of the stochastic process $\mathbf{S}_t$ may depend on some of the parameters in $\mathbf{X}$. Let the parameters $\mathbf{X}$ be $\mathcal{F}_t$-measurable for time t, such that they are fixed and known at time t.

Example 1 (SABR as a Parametric SDE)

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

The choice of dynamics for the stochastic process $\mathbf{S}_t$, for example the SABR in Example 1, represents the quant's assumptions for the dynamics of the underlying asset(s). In this text, we will also refer to this as the choice of *volatility* model (for a single underlying driven by one or more factors), or *market* model (for multiple assets). Suppose that there exists some equivalent local martingale measure (ELMM) $\mathbb{Q} \sim \mathbb{P}$, 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).

Definition 1 (Pricing as Conditional Expectation, European Payoffs) *By the Fundamental Theorem of Asset Pricing, given the existence of some ELMM $\mathbb{Q}$, the no-arbitrage price of a payoff $h(\mathbf{S_T})$ is given by the conditional expectation under $\mathbb{Q}$:*

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

We consider the most simple derivative in Definition 1: a European option on the underlying assets $\mathbf{S}_t$. In the most general case, a European payoff is characterised by a Borel-measurable payoff function $h : \mathcal{F} \to \mathbb{R}$, on the underlying value of the stochastic process $\mathbf{S}_T$, which entitles the holder of the derivative to receive a payment or cash flow of $h(\mathbf{S}_T)$, at time $t + \tau = T$.

In Definition 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 the first equality. In addition, if the initial values $\mathbf{S}_t$ and the fixed time-to-maturity τ are contained in the fixed parameters $\mathbf{X}$, and τ is fixed, then we obtain the second equality. Subsequently, we will include $\tau, \mathbf{S}_t$ into $\mathbf{X}$ for notational simplicity unless otherwise stated.

Definition 2 (Pricing as Integration) *We can express a conditional as a integral of the payoff times the transition density over the domain of* $\mathbf{S}$*:*

$$\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S})|\tau, \mathbf{X}] = \int h(\mathbf{S})p(T = t + \tau, \mathbf{S}; \mathbf{S}_t, \mathbf{X})d\mathbf{S}, \quad \mathbb{E}^{\mathbb{Q}}\left[|h(\mathbf{S}_T)|\tau, \mathbf{X}|\right] < \infty \tag{2}$$

In Definitions 1 and 2, we assume that the conditional expectation is well-defined for the valid ranges of $\tau, \mathbf{S}_T, \mathbf{X}$. If we know the $\mathbb{Q}$-transition density $p(\mathbf{T}, \mathbf{S}; \mathbf{S}_t, \mathbf{X})$ of $\mathbf{S}_T$ conditioned on $\mathbf{S}_t$ (and the fixed $\mathbf{X}, \tau$), we can then express the conditional expectation as an equivalent numerical integral.

The true pricing function f can therefore be expressed as a $\mathcal{F}_t$-conditional expectation of the payoff $h(\mathbf{S}_T)$ under some risk-neutral $\mathbb{Q}$, evaluated for fixed $\mathbf{X}$ in Definitions 1, or the equivalent numerical integral in 2. This motivate the use of *Monte Carlo* methods for derivatives pricing.

Definition 3 (Monte Carlo (MC)) *We can estimate the conditional expectation with Monte Carlo, with:*

$$\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S_T}(\mathbf{X}))|\mathbf{X}] \approx g(\mathbf{X}) = \frac{1}{N}\sum_{i=1}^{N} h(\overline{\mathbf{S_{i,T}}(\mathbf{X})}), \quad \overline{S_{i,T}(\mathbf{X})} \approx \mathbf{S_{i,T}}(\mathbf{X}) \tag{3}$$

Where the transition density function is known, it may be efficient to evaluate the numerical integral 2 for low dimensions. For high dimensions, the number of mesh point required may grow very large; for example, if N points are used in each dimension we would require N^d points. In addition, the transition density function $p(T, \mathbf{S}_T; \mathbf{S}_t, \mathbf{X})$ might be unknown (although we could approximate it in some cases by solving the corresponding Fokker-Planck PDE).

However, we may be able to simulate samples from the transition density, by simulating the corresponding stochastic differential equation $d\mathbf{S}_t$ conditioned on the volatility model parameters in $\mathbf{X}$ (supposing that $\mathbf{S}_t$ is well defined such that a strong solution exists). In Definition 3, suppose we are able to simulate samples $S_{i,T}, i = 1, \ldots, N$. Then, we can estimate the conditional expectation by considering a finite sample mean of the *sample payoffs* $h(\mathbf{S_{i,T}})$, with $\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} h(\overline{\mathbf{S_{i,T}}(\mathbf{X})}) = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S_T})|\mathbf{X}]$. Given that this estimate is only the true value in the asymptotic case, the finite sample MC is itself an approximation for a single parameter set ($\mathbf{X}$), with a convergence rate of $O(\frac{1}{\sqrt{N}})$. Another challenge is that it may be difficult to simulate the true solution to the SDE $\mathbf{S}_{i,T}(\mathbf{X})$, necessitating another approximation $\overline{S_{i,T}(\mathbf{X})} \approx \mathbf{S_{i,T}}(\mathbf{X})$, for example through the Euler and Milstein schemes.

Example 2 (European Call Option, 1 Asset, Black Scholes)

$$\begin{aligned} f(\mathbf{X}) &= \mathbb{E}^{\mathbb{Q}}[(S_T - K)^+|\mathbf{X}], \mathbf{X} = (\tau, K, S_t, \sigma) \\ dF_t &= \sigma S_t dW_t, \quad F_t \in \mathbb{R}, \sigma > 0, K > 0, \tau > 0 \\ g(\mathbf{X}) &= \frac{1}{N}\sum_{i=1}^{N}(S_{i,T}(\mathbf{X}) - K)^+ \end{aligned}$$

As an illustration, in Example 2 we consider the case of vanilla European call options in the Black-Scholes setting, written on a single driftless forward $F_t, r = 0$. A European call entitles the holder to a payoff of $h(S_T) = (S_T - K)^+$ at time $t+\tau = T$. We should note that, as in the case of European Calls options, payoff functions $h(\cdot)$ may more generally have their own contract parameters, so h is shorthand for $h(\cdot; \mathbf{X})$, although the contract parameters do not influence the dynamics of $\mathbf{S}_t$.

In this case, the contract parameters are the time-to-maturity τ and strike K, the volatility model parameter is σ, and the value of the asset is F_t. Thus in general, a pricing function requires three kinds of input: the parameters of the volatility model, the parameters of the payoff structure, and the values of the underlyings. When the parameters of the payoff function are also considered in $\mathbf{X}$, the pricing function f is a *parametric pricing function* for a *family* of payoffs as opposed to for a specific payoff.

In the Monte Carlo approach in Definition 3, we evaluate the price $g(\tau, \mathbf{X})$ for only one choice of $\mathbf{X}$. Under this setting, we can compute the prices for European calls some grid of (τ, K) in one-pass (or more generally some grid of the *contract parameters*), if we consider simulating sample trajectories S_{i,τ_j} for $j = 1, \ldots, N_\tau$ time-to-maturity points.

The advantage of Monte Carlo is that it is scalable to higher dimensions with a convergence rate of $O(\frac{1}{\sqrt{N}})$, but we only obtain the price and sensitivities for one initial state $\mathbf{S}_t$. We also need to re-simulate the paths if we were to choose a different choice of volatility parameters, in this case σ and value of the underlying F_0, or grid (τ, K).

Pricing as the solution to the PDE / Stochastic Control

Definition 4 (Ito's lemma for diffusion processes) *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 (under $\mathbb{Q}$) and apply Ito's lemma, we obtain its dynamics:*

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

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

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

$$= (-\frac{\partial f}{\partial \tau} + \mathcal{L} f) dt + \nabla_{\mathbf{S}} \cdot \sigma(t, \mathbf{S}_t) dW_t \tag{7}$$

Definition 5 (Feynman-Kac Formula)

$$Y_t = f(\tau, \mathbf{S}_t) = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T; \mathbf{X}) | \mathbf{S}_t] \tag{8}$$

Definition 6 (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 (thus given an ELMM, equivalently $\mathbb{Q}$-almost surely). Suppose $\int_t^T -\frac{\partial f}{\partial \tau} + \mathcal{L}f)dt > 0$*, then if we take a dynamic position of* $-\nabla_{\mathbf{S}}$ *in* $\mathbf{S}_t$*, and purchase one unit of the payoff, we have. 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.*

$$\mathbb{E}^{\mathbb{Q}}[(Y_T - Y_0) - \int_t^T \nabla_{\mathbf{S}} d\mathbf{S}_t] = \mathbb{E}^{\mathbb{Q}}[\int_t^T -\frac{\partial f}{\partial \tau} + \mathcal{L}f)dt] > 0 \tag{9}$$

Thus a sufficient condition for no-dynamic arbitrage is if $-\frac{\partial f}{\partial \tau} + \frac{1}{2}(\sigma^\top H_f \sigma)$

In order for the *backward-SDE* to Y_t represent the arbitrage-free pricing function, Y_t must be a $\mathbb{Q}$-local martingale; hence, the drift of Equation 4 must be zero. By the Feynman-Kac Theorem, the corresponding (parametric) pricing PDE is given by:

$$f_\tau = \mathcal{L}f, f(0, s, x) = h(s) \quad \forall \tau \in [0, T], \mathbf{X} \in \mathcal{X}, s \in \mathcal{S} \tag{10}$$

No Dynamic Arbitrage: For Dynamic Arbitrage, a sufficient condition may be that the function g satisfies the relevant pricing PDE:

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

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 continuous, once-differentiable with respect to $\tau = T - t$,and twice differentiable with respect to K.

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

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

Thus we can obtain the pricing function by solving the corresponding pricing PDE. This also connects pricing with *stochastic control* and Forward-Backward Stochastic Differential Equations: if we can solve for Y_t or determine the hedging strategy Δ_S, we can also price the derivative.

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 $\frac{\partial g}{\partial \mathbf{X}} \approx \frac{\partial f}{\partial \mathbf{X}}$ introduces additional error. Thus we also require an approximation for the derivatives:

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

Definition 7 (Finite Differences, 'Bump-And-Revalue') *Suppose we have the pricing function f, then we can approximate the derivatives with some $\epsilon > 0$:*

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

This follows from the definition of differentiation. In this case, we re-evaluate the function for every new $\mathbf{X}$. 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 $\mathbf{X} \in \mathbb{R}^{N_F}$, whereas for the second order sensitivities, we require $4(\frac{N_F^2+N}{2})$ evaluations. However finite differences may fail if the pricing function is non-smooth. We also require sufficiently small ϵ, but with sufficiently small ϵ we may incur *finite-precision* error.

Definition 8 (Finite Differences for PDE)

$$\frac{\partial f}{\partial \tau} = \mathcal{L} f, \quad \frac{g(\tau + \Delta\tau, \cdots) - g(\tau, \cdots)}{\Delta\tau} = \mathcal{L}(\Delta x) g, f(0, \mathbf{S}) = g(0, \mathbf{S}) = h(\mathbf{S}; \mathbf{X}) \quad (11)$$

Finite Differences is a key method to solve the pricing problem as a PDE formulation. In effect, we solve the PDE with finite difference approximation by approximating the differentials with finite-differences as in Definition , with a discretisation Δt for the time-to-maturity, and Δx for the state or underlying assets $\mathbf{S}$. PDE enables us to obtain a solution across the values of the underlying spatial discretisation $\prod_{i=1}^{d} \{S_{i,min}, S_{i,max}\}$ and time discretisation $\{0, \Delta\tau, \dots, N_\tau \Delta\tau = T\}$, however, not for the varying volatility model parameters and contract parameters in $\mathbf{X}$.

The advantage is that we also compute the sensitivities (although they may be accurate), and obtain a solution that depends on the discretisation $(\Delta, \Delta x)$ as opposed to random sampling error in the MC approach. However, the complexity grows with the dimensionality d, the number of underlyings in S_t, such that PDEs may be too computationally expensive in some cases for high-dimensional problems.

Definition 9 (Pathwise Differentials) *Suppose we have the pricing function f, and the conditional expectation is well defined such that we can interchange the order of integration. Then we can approximate the differentiaals with:*

$$\frac{\partial f}{\partial \mathbf{X}} = (\frac{\partial \mathbf{S}_T(\mathbf{X})}{\partial \mathbf{X}}) \frac{\partial}{\partial \mathbf{S}_T} \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)] = \frac{\partial S_T(\mathbf{X})}{\partial \mathbf{X}} \mathbb{E}^{\mathbb{Q}}[\frac{\partial}{\mathbf{S}_T} h(\mathbf{S}_T)] \quad (12)$$

Equivalently in this case, if we use conditional expectations, we are effectively considering: $\frac{\partial}{\partial \mathbf{X}}\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)]$ or $\frac{\partial \mathbf{S}_T}{\partial \mathbf{X}}\frac{\partial}{\partial \mathbf{S_T}}\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)]$. Then supposing we can interchange of the order of integration and differentiation, we can compute. An alternative is the adjoint automatic differentiation method [24].

Definition 10 (Adjoint Automatic Differentiation)

$$\frac{\partial f}{\partial \mathbf{X}} = \mathbb{E}^{\mathbb{Q}}[\frac{\partial}{\partial \mathbf{S}_T}h(\mathbf{S}_T)] \cdot (\frac{\partial \mathbf{S}_T(\mathbf{X})}{\partial \mathbf{X}}) \tag{13}$$

In 12, we compute the sensitivities with forward differentiation, whereas in 13 we compute the sensitivities of f with respect to $\mathbf{X}$ backward via the application of the reverse chain rule. *Adjoint* implementations of operators in the programming framework make the backward mode differentiation more efficient, as it involves vector-matrix products instead of matrix-matrix products for the pathwise approach. To quote [24], leads to a worst-case time-complexity no worse than 'a factor 4 greater than the cost of the original calculation, regardless of how many sensitivities are being computed'. As we will describe in Chapter 3, adjoint automatic differentiation is a key tool that enables the training of neural networks.

Example 3 (Payoff Smoothing) *We can approximate the payoff digital payoff* $h(S;K) = 1_{S>K}$ *with a piecewise linear approximation or call spread* $h_{ramp}(S;K,\epsilon)$*:*

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

An issue with the adjoint method is that it fails for non-smooth payoff functions, which we can address to some extent with *payoff smoothing*. In Example 3, we consider approximating a digital option, which has a non-differentiable payoff, with a piecewise linear approximation or *call spread*. This allows us to apply the adjoint or pathwise differential method, although this comes at the cost of introducing some *bias* in the estimation of the sensitivity.

Pricing for Other Payoffs

If we are able to price a single European payoff, then 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.

Definition 11 (Path-Dependent Payoff) *A path-dependent payoff depends on the entire history of* $\mathbf{S_t}$ *over the start and maturity* $\in [0, T]$

$$f(\tau, (\mathbf{S}_u)_{u\in[0,t]}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[h_2((\mathbf{S_u})_{u\in[0,T]})|(\mathbf{S}_u)_{u\in[0,t]}\mathbf{X}\right] \tag{14}$$

Example 4 (Asian and Barrier options)

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

$$f(\tau, S_t, M_t) = \mathbb{E}^{\mathbb{Q}}\left[(S_T - K)^+ 1_{M_T \leq b}|S_t, M_t\right], M_t = \max_{u\in[0,t]}\{S_u\}, t \in [0, T] \quad \text{(Barrier)}$$

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 $A_t = \frac{1}{t}\int S_u du$, or in the case of a knock-out barrier, the running maximum. $\max_{u\in[0,t]}\{S_u\}$ 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 has error, and may not be arbitrage free.

Definition 12 (Callable Payoffs) *A callable payoff allows the holder to enter / exit a payoff at some time* $T^* \in \mathcal{T}$*, over a set of possible exercise times* $\mathcal{T}$*.*

$$f(\mathbf{X}) = \sup_{T^*\in\mathcal{T}} \mathbb{E}[h(\mathbf{S}_{T^*})|\mathbf{X}] \tag{15}$$

Another additional complexity is that payoffs may also have callability features, in this case, there is an additional complexity as the optimal exercise time T^* must also be determined.

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 a given pricing function f. [41] considered use the of sequential regressions as a method to approximate the value function for Americans.

$$g(T - t, \mathbf{X}) \approx f(T - t, \mathbf{X}), \quad f(T - t, \mathbf{X}) = g(\mathbf{X}) + \epsilon \tag{16}$$

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 $\mathbf{X}$.

For a single timestep (period), they consider approximating the true value function, or conditional expectation, f 1 using basis function regression.

[41] [53] considered basis function regression. In [41], they consider basis functions such as Chebyshev and Legendre polynomials, however, any general basis could

be considered, such as the Karhunone-Louvre basis in [52] 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.

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

$$g_{t_i}(\mathbf{S}_{t_i}) = \sum_{i=1}^{N_B} w_i \phi_i(\mathbf{S}_{t_i}) + b, \mathbf{S}_{t_i} \in \mathbb{R}^d \tag{17}$$

Where in the one-step case:

$$g_{t_{N_T-1}}(\mathbf{S}_{t_{i-1}}) \approx \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_{t_{N_T-1}}] \tag{18}$$

The to price a Bermudan option they use sequential regressions:

$$g_{t_i}(t_i, \mathbf{X}) \approx \max\{\mathbb{E}^{\mathbb{Q}}[g(\mathbf{S}_{t_{i+1}})|\mathbf{S}_t]\} \tag{19}$$
$$f(X_{t_i}) = \max\{\mathbb{E}^{\mathbb{Q}}[f(X_{t_{i+1}})\}|X_{t_i}], h(X_{t_i})\}, \quad f(X_T) = h(X_T)$$

In the application of [41], 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

[41] 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.

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

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

Then by the tower property of expectation

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

No-Arbitrage Constraints

When we consider an approximation g, it is not guaranteed to have the same properties as the true pricing function f. In particular, we are concerned with the possibility of arbitrage. In the previous subsections, we described that no-dynamic arbitrage is attained when g solves the corresponding pricing PDE of f. However, in addition we must consider no static arbitrage.

$dg/d\tau \geq 0$	carry, time-value
$dg/dK \leq 0$	decreasing in strike
$d^2g/dK^2 \geq 0$	convexity in strike
$0 \leq (S_t - K)^+ \leq g(\tau, K) \leq S_t$	lower, upper bounds
$\lim_{K\to\infty} g(\tau, K, \ldots) = 0$	Strike Asymptotic
$\lim_{\tau\to 0} g(\tau, K, \ldots) = (S_T - K)^+$	Exercise Value

Table 2: No Static Arbitrage for European Call Options from [20] and [10]

Definition 13 (No Static-Arbitrage Constraints) *No Static-Arbitrage can be defined as a self-financing trading strategy with zero initial wealth that does not require dynamic re-balancing, which leads to $\mathbb{P}$-almost surely positive wealth.*

We focus on the case for European call options on a single underlying, as the *no static-arbitrage* bounds are explicitly known [20] and [10], some of which are depicted in Table 2. These no static-arbitrage relations hold for European Calls hold for any strike, maturity, and underlying $K, T, S_t \in [0, \infty)$, and for any general volatility model (thus they are *model-free*). Thus ideally any approximating function $g(\mathbf{X})$ for a European Call option should satisfy these properties.

Definition 14 (Breeden-Litzenberg) *Suppose that the transition density p is well-defined. Then differentiating under the integral, we can extract the transition density:*

$$g(T, K) = \int_{\mathbb{R}} (y - K)^+ p(y, T; s, t) dy = \int_K^\infty (y - K) p(y, T; S_t, t, \mathbf{X}) dy \tag{20}$$

$$\frac{dg}{dK} = -\int_K^\infty p(y, T; S_t, t, \mathbf{X}) dy = -\mathbb{E}^{\mathbb{Q}}[1_{S_T > K} | \mathbf{X}] = -\mathbb{Q}[S_T > K] \tag{21}$$

$$\frac{d^2 g}{dK^2} = p(K, T; , S_t, t) \tag{22}$$

The case of the European Calls is also particularly relevant as we can extract the risk-neutral density (Definition 14). Thus if we are able to approximate f and its second derivatives with g accurately, we can extract the transition CDF and transition PDF from the derivatives $1 - \frac{dg}{dK}$ and the transition density $\frac{d^2 g}{dK^2}$ (under $\mathbb{Q}$). This also leads gives additional no-arbitrage constraints on $\frac{dg}{dK}$ such that $1 - \frac{dg}{dK}$ is a valid CDF:

$$\lim_{K\to\infty} \frac{dg}{dK} = 0, \lim_{K\to 0} \frac{dg}{dK} = -1 \tag{23}$$

From $\frac{d^2 g}{dK^2}$, we can then price all European options from Definition 2.

Definition 15 (Put-Call Parity) *For the pricing functions g^{CALL} and g^{PUT}, for European Calls and Puts, we must have:*

$$g^{CALL} - g^{PUT} = \mathbb{E}^{\mathbb{Q}}[(S_T - K)^+ - (K - S_T)^+] = \mathbb{E}^{\mathbb{Q}}[S_T - K] = (S_t - K) \tag{24}$$

Derivatives prices must be also consistent with other prices. In the context of European options, we have the put-call parity relation (24). For non-European options, in some cases, we could replication arguments to obtain no-arbitrage lower and upper bounds, for example for Barriers (Example 5):

Example 5 (Barrier Replication) *For a down-and-out call and up-and-in call with the same barrier level b,maturity T, and strike K, we have;*

$$\mathbb{E}[(S_T - K)^+ 1_{M_T<b}] + \mathbb{E}[(S_T - K)^+ 1_{M_T>b}] = \mathbb{E}^{\mathbb{Q}}[(S_T - K)^+]$$
$$V_{down} + V_{up} = V_{call}$$

Or we could consider inequality relations, for example for Callables (Example 6):

Example 6 (Callable Bounds) *For callables with the same underlying* $\mathbf{S}_t$*, terminal maturity* T*, and payoff function* h*, the value is increasing in the number of exercise dates :*

$$\sup_{T^* \in [0,T]} \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T^*)] \geq \sup_{T^* \in \{T_1, T_2, \ldots, T\}} \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T^*)] \geq \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)]$$
$$f^{American}(\mathbf{X}) \geq f^{Bermudan}(\mathbf{X}) \geq f^{European}(\mathbf{X})$$

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.

Summary

In this sections, we described some of the standard approaches to obtain derivatives pricing and sensitivities. We can approximate the conditional expectation 1 with 3 and obtain sensitivities via AAD, solve the PDE with finite differences, or use numerical integration 2 (which could also involve a PDE solution to the Fokker-Planck Equation). In some cases, these procedures can be computationally expensive, particularly in the case of high dimensionality, and must be re-invoked for new parameters $\mathbf{X}$. Basis function regression, or Least Squares Monte Carlo can be used as an approximation, but the approximation is not guaranteed to satisfy the no-arbitrage constraints without special construction. In the next section, we define neural networks as an alternative to basis function regression.

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 [26], 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:

$$\begin{aligned} \mathbf{Z}^1 &= g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{1}(\mathbf{b}^1)^\top) \\ \mathbf{Z}^2 &= g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{1}(\mathbf{b}^2)^\top) \\ &\vdots \\ g(\mathbf{X}; \boldsymbol{\theta}) &= g_n(\mathbf{Z}^{N_H-1}\mathbf{W}^n + \mathbf{1}(\mathbf{b}^n)^\top \end{aligned} \tag{25}$$

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 $\mathbf{b}^i$ 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 obtained via dedicated hardware such as Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). In this direction, [38] mentions frameworks such as the NVIDIA TensorRT, which can further be used to speedup inference of a trained neural network and [28] 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 3 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 3: 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 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. [47] discusses some potential speed-ups 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 engineering side, further investigation 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 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 17). Additionally, when $N_H = 1$ (i.e. no hidden layers) and g_1 is the identity function, the neural network is *exactly* a 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 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 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 Equation 25, 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:

$$g_n(\mathbf{Z}^{N_H-1}\mathbf{W}^{N_H}) = \left(g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,1} + \mathbf{b}) \quad \cdots \quad g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,H_{N_H}} + \mathbf{b})\right) \tag{26}$$

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, [43] considers using a neural network with a $H_{N_H} = 10$-dimensional output of SABR-implied volatilities at fixed strikes, and [31] 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. 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; we may also be unsure about 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:

$$\sum_{i=1}^{N_E} a_i g(\mathbf{X};\boldsymbol{\theta}^1) = \frac{1}{N_E}\begin{pmatrix}\mathbf{Z}^{N_{H-1},1} & \mathbf{Z}^{N_{H-1},2} & \dots & \mathbf{Z}^{N_{H-1},N_E}\end{pmatrix}\begin{pmatrix} a_1\mathbf{W}^{N_H,1} \\ a_2\mathbf{W}^{N_H,2} \\ \vdots \\ a_{N_E}\mathbf{W}^{N_H,N_E}\end{pmatrix} \tag{27}$$

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 be 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.

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

A drawback of this approach is that this may potentially lead to a more complex implementation, and reduce inference speed , given the need to evaluate n neural networks instead of 1. Another implication is that we could aggregate neural networks, for example 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 theorem of [30] 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 its derivatives. The theorem suggests that, for our derivatives modelling application, we must use a continuous k-times differentiable C^k activation function g in order to obtain k-th order sensitivities. This rules out the commonly used Rectified Linear Unit (ReLU) activation function, and the LeakyReLU function.

However, the universal approximation theorem only proves the 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:

$$\arg\min_{\theta \in \Theta} L(\mathbf{y}, g(\mathbf{X})) \tag{28}$$

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

$$\arg\min_{\theta \in \Theta} MAE(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \mathbb{E}[\|\mathbf{y} - g(\mathbf{X}; \theta)\|] \tag{29}$$

$$\arg\min_{\theta \in \Theta} MSE(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \mathbb{E}[\|\mathbf{y} - g(\mathbf{X}; \theta)\|^2], \tag{30}$$

In Equation (32), the input parameters $\mathbf{X}$ are no longer fixed, but a $\mathbb{R}^f$ random variable over some probability space $(\mathcal{X}, \mathcal{F}^{\mathcal{X}}, \mathbb{P}^*)$. 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.

$$\arg\min_{\theta \in \Theta} \mathrm{MAE}(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} |y_i - g(\mathbf{X}_i)| \tag{31}$$

$$\arg\min_{\theta \in \Theta} \mathrm{MSE}(\mathbf{y}, g(\mathbf{X}; \boldsymbol{\theta})) = \arg\min_{\theta \in \Theta} \frac{1}{N} \sum_{i=1}^{N} (y_i - g(\mathbf{X})_i)^2, \tag{32}$$

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 $t \leq$ EPOCHS $\times \lceil \frac{N}{\text{BatchSize}} \rceil$ or NOT StoppingCriterion **do**
  for batch $= \lceil 1, \ldots, \frac{N}{\text{BatchSize}} \rceil$ **do**
    Compute loss $L(y^{batch}, g(\mathbf{X}^{batch}))$
    Compute gradient w.r.t. loss ∇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. Consequently, 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 to $O(BatchSize)$ as opposed to the entire dataset $O(N)$. Furthermore, the procedure can be parallelised, or further speed up 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 [39].

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: [51] 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 'over-dependence' on a particular basis function.

Weight Regularisation: We set 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 25. We describe several hidden layer, or 'block ar-

chitectures', 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 $\mathbf{Z}_1, \mathbf{Z}_2$ have the same dimension. This can facilitate learning multiplicative interactions between the basis functions / feature. [55] uses gated units to construct a neural network that has the requisite monotonicity and convexity constraints with respect to strike and moneyness.

$$\mathbf{Z}^1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{b}^1\mathbf{1}^\top) \tag{33}$$
$$\mathbf{Z}^2 = g_2(\mathbf{X}\mathbf{W}^m athbf b^2\mathbf{1}^\top) \tag{34}$$
$$\mathbf{Z}^3 = \mathbf{Z}_1 \otimes \mathbf{Z}_2 \qquad \text{(gated block)}$$

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.

$$\mathbf{Z}_1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{b}^1\mathbf{1}^\top) \tag{35}$$
$$\mathbf{Z}_2 = g_2(\mathbf{Z}_1\mathbf{W}^2 + \mathbf{b_2}\mathbf{1}^\top) + \mathbf{Z}_1 \qquad \text{(residual block)}$$

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:

$$\mathbf{Z}^3 = \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top \tag{36}$$
$$L(\mathbf{y}, g(\mathbf{X})) = \|(\mathbf{y} - \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top)\|^2 \tag{37}$$
$$= \|\mathbf{y} - \left((g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{b_2}\mathbf{1}^\top) + \mathbf{Z}^1)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top\right)\|^2 \tag{38}$$
$$= \|\mathbf{y} - (g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{b_2}\mathbf{1}^\top)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top] - (\mathbf{Z}^1\mathbf{W}^3)\|^2 \tag{39}$$

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, an 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 [29] 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; namely, the number of hidden units, layers, and other hyperparameters.

AutoML, Hyperparameter Tuning: From the Universal Approximation theorem, 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 a neural network, the depth and 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:

$$\arg \min_{g_i \in \mathcal{N}_{sub}} \min_{\theta \in \Theta_{g_i}} L \tag{40}$$

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.[46] 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 illustration, 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 $i = 1, \ldots, N_H - 1$ hidden layers.

$$H_i \times (N_H - 1) \times g_i \in \{32, 64, 128\} \times \{32, 64, 128\} \times \{\text{ELU}, \text{Swish}\}$$
(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 [25] considered in their paper for using neural networks to solve parametric pricing PDEs.

Summary

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.

4 Neural Networks for Derivatives Pricing

In the previous sections we considered the problem formulation of approximating derivatives prices and sensitivities, the characteristics and construction of neural networks. In this section, we review some of the existing literature and methods on how to apply neural networks in derivatives pricing and risks from the perspective of the entire workflow. [48] , which presents an excellent survey article on neural networks towards derivatives modelling, describes several approaches by authors to approach derivatives modelling through neural networks. A key theme is that it may be possible to achieve better performance on the neural network , by constructed 'hand-crafted' neural networks with prior knowledge, through the choice of an appropriate architecture or loss functions, which we examine in this section.

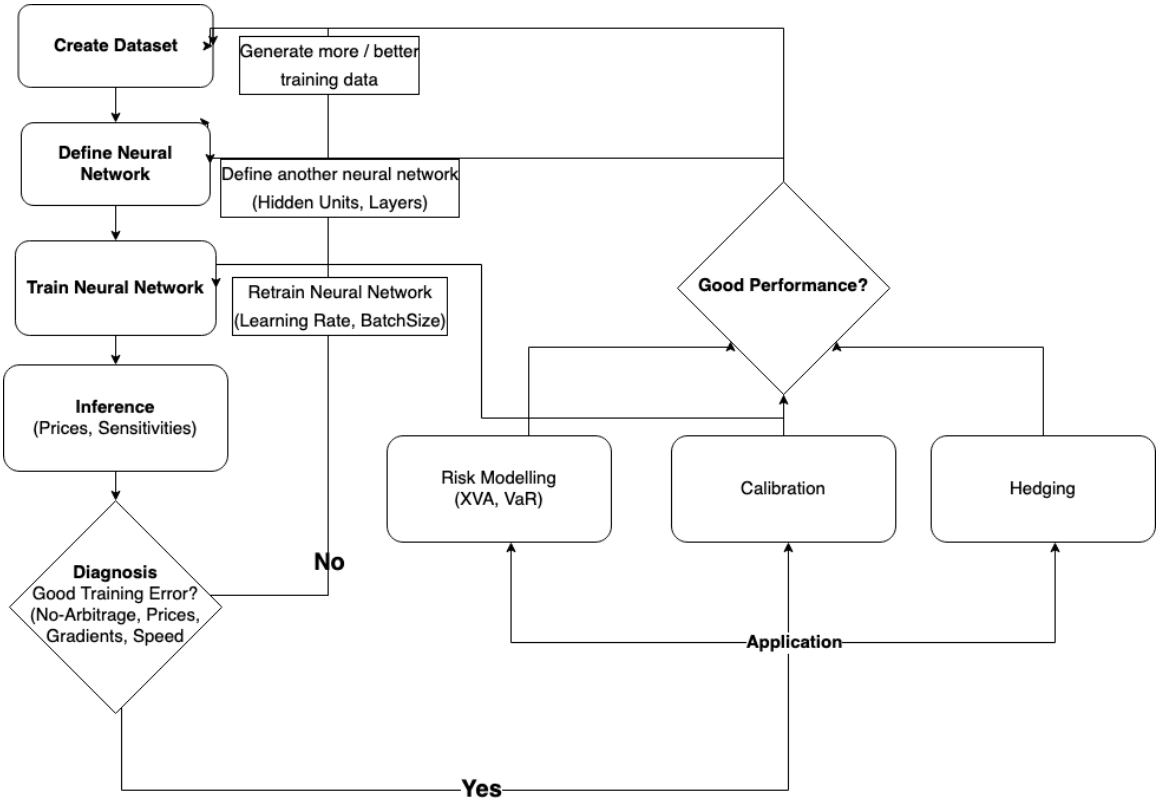


Figure 1: Example workflow for using neural networks in derivatives modelling

We consider two potential settings for pricing function approximations. For example: [43] [31] [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'. On the other hand [32] uses a neural network to solve a complex pricing problem on-the-fly (high-dimensional / callable/ path-dependent), with some fixed parameters (volatility, payoff parameters).

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 elaborated on how to generate a dataset of $\mathbf{X}, \mathbf{y}$.

Choice of y: The first question is what to choose as a definition of price $\mathbf{y}$. Although we may wish to approximate the true pricing function $\mathbf{y} = f(\mathbf{X})$, we can do so through defining different choices of $\mathbf{y}$. The most straightforward approach could just be to let $\mathbf{y}$ be the prices under some volatility model and payoff family, for example $\mathbf{y} = f^{BS,Call}(\mathbf{X})$. On the other hand, [43] let $\mathbf{y}$ to be implied volatilities, such that we can apply $f^{BS}(\sigma = y, X = (S, T, K, \cdots))$ to obtain the corresponding price. 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 [31] and [43].

Simulating y. If we have a closed-form solution, then we can use it to generate the prices $\mathbf{y}$ (although these cases would be trivial and only useful to investigate the empirical performance of neural networks, as in this paper). This leaves us to generate $\mathbf{y}$ uing MC or FD / PDE, in which we can leverage the known best practices for each setting, for example Quasi Monte Carlo and control variates for MC, or implicit schemes or sparse grids for PDEs. In the MC setting [32] lets $\mathbf{y}$ to be sample Monte Carlo payoffs (i.e. $N_{SAMPLES} = 1$, such that y_i is only an unbiased estimate of the true pricing function $\mathbb{E}[y_i] = f(\mathbf{X}_i)$. [19] also explores the use of MC payoffs, and comments that the neural network may be able to 'denoise' the data.

Simulating X: The *model risk* of the neural network is also present through the dataset. 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 [29] 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). However, the neural network may learn well at the dense clusters of the sampling distribution (e.g. near the mean of each sampling distribution), but poorly at the boundaries of the training dataset. For example the neural network could fail to learn the relationships for deep-in-the-money or out-the-money calls. A potential solution to address this in [32] is to simply simulate more points at these regions (equivalently, this could be thought of as applying weights for each point in the loss function). How to efficiently generate a dataset could be a direction for further exploration.

Data Preprocessing: Given that we need a numerical method to generate the price labels $\mathbf{y}$, the dataset may contain noise $\boldsymbol{\epsilon}$ $\mathbf{y} = f(\mathbf{X}) + \boldsymbol{\epsilon}$, and may contain arbitrageable prices. As a result, the neural network could learn these arbitragable prices. As a possible solution, [14] proposes a method to remove arbitrage in the call prices by projecting the call prices to a polytope of linear no-static-arbitrage constraints.

Neural Network Architecture - 'Hard Constraints'

The 'hard-constrained' or Gated Neural Network approach refers to imposing architectural constraints to restrict the outputs, and pre-define some gradient relationships, similar to the use of special blocks in Chapter 3.

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 [34] [11], which arises from the Universal Approximation Theorem **??**. This excludes the commonly used ReLU if continuous first-order gradients are needed, and the ELU function in Table 3. The softplus, swish, and gelu activation functions may be appropriate given that they are C^∞, 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. [34] 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.

Example 7 (Loss Function with Price, Mean Squared Error (MSE))

$$??L_{price}(y_i, g(X_i)) = (y_i - g(\mathbf{X}_i))^2 \tag{41}$$

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

Example 8 (Loss Function with Implied Volatility)

$$L_{price}(y_i, f^{BS}(g(X_i))) = (y_i - f^{BS}(g(\mathbf{X}_i)))^2, L_{vol}(y_i, g(X_i)) = (y_i - g(\mathbf{X}_i))^2 \tag{42}$$

As mentioned, we can also consider predicting implied volatility, as considered in [43] In this, setup we could either have the neural network minimise the error in implied volatility. A potential advantage of the latter 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.

Control Variate: Another approach could be to consider a control-variate approach. 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 [27]. In this context, the neural network is used to 'correct' the error of another pricing approximation.

Example 9 (Control Variate)

$$L_{price}(y_i, g(X_i)) = (y_i - ControlVar_i - g(\mathbf{X}_i))^2, g_{N_H}(x) = \exp(-\frac{x^2}{2}) \tag{43}$$

Given that the cubic spline interpolation has the correct asymptotics, their proposed architecture leverages a specific activation function, the radial basis function (RBF) from Table 3 to ensure that the predicted price $ControlVar + g(\mathbf{X_i})$ has the correct asymptotics. 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 as described in Chapter 2; thus in this case the predicted price will be guaranteed to satisfy the intrinsic lower value bound.

Example 10 (Deep Hedging, FBSDE)

$$L(y_i; g) = \frac{1}{N_{PATHS}} \left\| h(\mathbf{S_T}) - \sum_{i=1}^{N_\tau} g(t_i, \mathbf{X_{t_i}})(\mathbf{S_{t_{i+1}}} - \mathbf{S_{t_i}}) \right\| \tag{44}$$

A similar method could be to price through a hedging / replicating strategy; as in [6], which also has a connection with the stochastic control / FBSDE approach for pricing. In the *Deep Hedging* approach of, in this case, the neural network represents the *delta* or the positions to take in each one of the underlying assets, and the loss function for a single defined as being the norm of the pathwise hedging error over some MC sample paths. We can then price by considering the mean hedging error or through superhedging, or perhaps using another neural network to predict the initial price. In [6], they examine only fixed set of volatility model and payoff parameters, although the approach can be extended to non-fixed parameters. However, a potential drawback is that to price, we require MC simulations for the values of the factors S_{t_i}.

Example 11 (Soft Penalty)

$$L_{soft\ penalty}(y_i, g) = \lambda_0 L_{price} + \sum_{i=1}^{P} \lambda_i L_{cons_i} \tag{45}$$

Another approach is to incorporate no-arbitrage constraints into the loss function, similar to the penalty method in convex optimisation. [34] describes this as a *soft penalty* approach. In the most general case, let $L_{cons_1}, \dots L_{cons_P}$ denote P soft penalties; one example could be a penalty for being beneath the intrinsic call bound $((S-K)^+ - g(S))^+$. The soft penalty approach is able to penalise linear constraints in the pricing approximation, 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, only that they are penalised, unlike a hard constrained neural network.

Example 12 (Differential Machine Learning)

$$\lambda L_{price}(g(X_i), f(X_i)) + \lambda L_{greeks}\left(\frac{\partial g(X_i)}{\partial X}, \frac{\partial g(X_i)}{\partial X}\right), \lambda \geq 0 \tag{46}$$

The proposed method from [32] is to consider a joint loss function, an approach they describe as *differential machine learning*. In this case, we also need to generate the differentials $\frac{\partial y}{\partial \mathbf{X}}$[32] in the training dataset. The advantage of this approach may be that it encourages convergence in the gradients as well, which is important if we also want accurate sensitivities. However, the need to compute differentials leads to an increase total training time, which des not matter in the offline setting, but may be significant in the on-the-fly setting.

Example 13 (Neural PDE)

$$L_{price}(g(X_i), f(X_i)) + \lambda L_{PDE}(\mathcal{L}g) \tag{47}$$

We could also consider the corresponding PDE associated with the pricing problem, which can be described as a *Neural PDE* approach. Use of neural networks to solve (high-dimensional) PDEs as in *Physics-Inspired Neural Network*, and [50] formulated a Neural PDE approach to solve high-dimensional PDE problems called the *Deep Galerkin Method*. Here $\mathcal{L}$ denotes the corresponding PDE operator for the volatility model. [54] 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. [25] considered an extension to [50] to solve parametric PDEs, as opposed to a PDE for a fixed vol model and payoff. We note that we do not need to generate differentials in the training dataset for this method to work, in contrast to the differential method in Example 12. In [50], they outline a training method where price labels $\mathbf{y}$ do not need to be pre-computed, instead, we may only need to simulate the state space $\mathbf{X}$ and generate samples for the boundary conditions $g(0, \mathbf{X}) = h(\mathbf{X})$. However the neural PDE cannot be time-to-maturity τ is not an input.

Example 14 (Multi-Objective)

$$\lambda L_{price} + \lambda_1 L_{greeks} + \lambda_2 L_{PDE}(\mathcal{L}g) + \sum_{i=1}^{P} \lambda_i L_{cons_i} \tag{48}$$

In the most general case, we could have a complicated loss function that combines multiple losses. The additional challenges are now that the problem becomes multi-objective optimisation problem, and we must determine the optimal weightings λ for each objective. [32] argues that the λ_1 is not significant and they set it to $\lambda_1 = 1$, but this could be determined by considering prior knowledge of the relative scales or through trial-and-error. [44] proposes a method to determine the relative weightings for λ_2 between the pricing error and the PDE error.

Summary

To summarise, different authors have proposed various construction methods, which can incorporate prior knowledge of a derivative and market model into the neural network. In the subsequent section, we aim to explore these different constructions in various settings, the behaviour of these methods, and whether any method can produce consistently better results in our numerical experiments.

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 [45], 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

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

$$g(\mathbf{y}, \mathbf{X}^{contract}) = \mathbf{X}^{vol}, f(\mathbf{X}^{vol}, \mathbf{X}^{contract}) = \mathbf{y} \tag{49}$$

In this case, we determine the unknown volatility model parameters from the known values of the assets and contract parameters, and revert to the original MC / PDE pricing method.

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:

$$\mathbf{S}_t = g(\mathbf{S}_t; \boldsymbol{\theta}) d\mathbf{W}_t \tag{50}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} [y_i(\mathbf{X}_i) - \mathbb{E}[h_i(\mathbf{S}_T; \mathbf{X})] \| \tag{51}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} \left[y_i(\mathbf{X}_i) - \sum_{j=1}^{N_B} h_i(\mathbf{S}_T; \mathbf{X}; W_j) \right] \| \tag{52}$$

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) [36]. 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

Chebyschev / Tensor Methods: [2] Ruiz, Glau, 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 for future exploration.

Gaussian Processes / Kernel methods: [15] [37] 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 drawback 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 very low prediction errors for some applications. However, more advanced implementations of Gaussian Processes could be potentially explored and compared against the performance of neural networks as a future direction.

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: [40] 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.

5 Numerical Experiments

We consider three settings: Black-Scholes and Rough Bergomi. In all cases, we fix the architecture, to explore the empirical impact.

We conduct our numerical experiments in a similar workflow as the 1: we construct a dataset, define the neural networks, and then evaluate, prediction, gradient, PDE, and no-arbitrage error where available, and the model and time complexity as well.

The code, which leverages open source libraries will be available at github.com/XXX. `Python` is used as the programming language. In particular, we leverage the `numpy` library for MC / SDE simulation, the `Jax` library to obtain gradients in MC via AAD, and the `Tensorflow/Keras` API for constructing neural networks. We benchmark the results against polynomial regression using the implemention from the `scikit-learn` framework. In the Rough Bergomi experiment, we use the implementation for the Turbocharged Rough Bergomi scheme [1] from [42]. For the basket example, we consider the notebook examples [2] for [32] as a reference.

In Equation 48 we could consider incorporating multiple losses. For our numerical experiments we consider each construction method individually, where $\lambda^{method} = 1$ if the method is the method to be evaluated, and 0 otherwise.

Geometric Brownian Motion / Black-Scholes

[INCOMPLETE, fix table, add graphs, tidy]

Setup: We consider the most simple case: European Call pricing in 1D Black-Scholes. Although the neural network approximation is trivial given the availability of the Black-Scholes formula, this example is a setting where we can examine the behaviour of the various construction methods in a setting where we can train and test against the true price. The European Call case is also particularly relevant, as if we can accurately approximate European Calls accurately, we can price all European payoffs. In this case, we aim to learn a neural network 'clone' of the Black-Scholes formula over a wide range of parameters. In the true setup for this case, we would determine the optimal neural network architecture via extensive hyperparameter search.

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

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

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

[2] Accessed from : github.com/differential-machine-learning/notebooks

We apply the change of variables to $M_t = \log(F_t/K)$ as suggested by [48]. As mentioned in Chapter 3, inputs and outputs that are standardised may lead to better training. The Log-moneyness is exactly gaussian in this case, whereas, the forward or normalised forward F_t/K would be log-normal with skew

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

$$\frac{\partial \sigma\sqrt{\tau}}{\partial \tau} = \frac{\sigma}{2\sqrt{\tau}} = \frac{\sigma^2}{2\sigma\sqrt{\tau}}, \quad \frac{\partial m}{F} = \frac{\partial m}{F} = \frac{1}{F} \tag{53}$$

$$\frac{\partial}{\partial F}(\frac{\partial g}{\partial m}\frac{\partial m}{\partial F}) = \frac{-1}{F^2}\frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}\frac{1}{F^2}0 \qquad = -g_\tau - \frac{\sigma^2}{2}g_m + \frac{\sigma^2}{2}g_{mm}, g \tag{54}$$

$$0 = \frac{\sigma^2}{2}[-g_{\sigma\sqrt{\tau}}\frac{1}{\sigma\sqrt{\tau}} - g_m + g_{mm}], g(\tau, m) = (\exp(m) - 1)^+ \tag{55}$$

$$\tag{56}$$

The Black-Scholes PDE under the change of variables given by:

$$0 = \frac{\partial g}{\partial \sigma\sqrt{\tau}}\frac{1}{\sigma\sqrt{\tau}} - \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}, \quad g(\tau, m) \tag{57}$$

No Arbitrage Bounds: The no-arbitrage call bounds under the change of variables is given by:

$$\frac{\partial g}{\partial \tau} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}}\frac{\sigma}{2\sqrt{\tau}} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}} > 0 \tag{58}$$

$$\frac{\partial g}{\partial K} < 0 \implies \frac{\partial g}{\partial m}\frac{-1}{K} < 0 \implies \frac{\partial g}{\partial m} > 0 \tag{59}$$

$$\frac{\partial^2 g}{\partial K^2} > 0 \implies \frac{1}{K^2}\frac{\partial g}{\partial K} + \frac{\partial^2 g}{\partial m^2}\frac{1}{K^2} < 0 \implies \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2} > 0 \tag{60}$$

$$(\exp(m) - 1)^+ \leq g(\sigma\sqrt{\tau}, m) \leq \exp(m) \tag{61}$$

$$\lim_{m\to-\infty} g(\sigma\sqrt{\tau}, m) = 0, \lim_{m\to\infty} = \exp(m) \tag{62}$$

$$\lim_{\sigma\sqrt{\tau}\to 0} g(\sigma\sqrt{\tau}, m) = (\exp(m) - 1)^+ \tag{63}$$

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 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. 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. 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 [33] [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

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

$$\mathbb{E}^{\mathbb{Q}}[\frac{(F_t}{K} - 1)^+] = \frac{\mathbb{E}^{\mathbb{Q}}[(F_T - K)^+ | S_t, K, \sigma]}{K} \tag{64}$$
$$\lambda C(F_t, K, \sigma, \tau) = C(\lambda F_t, \lambda K, \sigma, \tau) \tag{65}$$

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 differentials $\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$ for the differential method. We consider two datasets as in [1], one with the identical parameter space which we denote Test1, and one with a larger parameter space to test extrapolation. We note that in the 'digital clone' application, it may suffice to use the approximation only on the sample space it is trained on, but consistency in extrapolation and asymptotics may potentiaally lead to better predictive performance oveall.

	Number ofSamples	$m = \log(F/K)$ Log-Moneyness	$\sigma\sqrt{\tau}$ (Time-scaled Implied Volatility)
Train	65,536	Uniform $[-1, 1]$	Uniform $[0.2, 0.6]$
Test1	65,536	Uniform $[-1, 1]$	Uniform $[0.2, 0.6]$
Test2	65,536	Uniform $[-1.5, 1.5]$	Uniform $[0.1, 0.8]$

Table 4: Parameter space for the Black-Scholes European calls example

Models: We first consider a polynomial basis regression (PolyReg) as a benchmark method, with degree 7 such that the number of basis functions is $\binom{7+2}{7}$. We use the same construction for the feed-forward neural network, Residual Network (ResNet), Differential Network (DiffNet), and the Neural PDE (NPDE) approaches: a rectangular neural networks with $N_H - 1 = 4$ hidden layers, and $H_i = 36$ hidden units, in each layer, such that both Polynomial Regression and the Neural Networks consider the same number of basis functions. For the Gated network, we consider an amount of hidden units $H_i = 36^2$ in the single hidden layer, such that the total number of parameters in the network is approximately the same. compensate for the lack of depth with width. Finally, we consider an average of the neural network models (Ensemble). For shorthand let FFN denote the feed-forward neural network 25, ResNet denote the residual network **??**, Gated denote the Gated Network **??**, NPDE denote the Neural PDE approach **??**, and PolyReg denote Poylnomial Regression

	$N_H - 1$	H_i	g_i
Feed-Forward Network (FFN)	4	36	softplus
Residual Network (ResNet)	4	36	softplus
Differential (DiffNet)	4	36	softplus
Neural PDE (NPDE)	4	36	softplus
Gated	1	$1296 = 36 \times 36$	softplus, sigmoid
PolyReg	-	$36 = \binom{7+2}{7}$	-

Table 5: Hyperparaameters for the Neural Network Architectures

The other hyperparameters are:

- Epochs = 30, BatchSize = 256, Learning Rate $\lambda_0 = 10^{-4}$, Optimizer: Adam, EarlyStopping with patience = 5,

  See github.com/XXX/notebook1 for the example notebook.

  Results: In terms of prediction errors, the DiffNet is the best performing neural network approach. However, polynomial regression outperforms it in all the pricing errors for the test dataset with the same parameter space as training. However, we also consider the extrapolation testing set is also considered, the DiffNet is the best performer in Mean Absolute Error and Root Mean Squared Error, but the NPDE is the best performer in the Maximum Error.

	MAE, Test1	RMSE, Test1	Max, Test1	MAE, Test2	RMSE, Test2	Max, Test2
FFN	8.97e-03	1.14e-02	6.62e-02	1.11e-01	2.12e-01	1.19
ResNet	6.03e-03	8.18e-03	3.45e-02	4.71e-02	8.79e-02	5.10e-01
DiffNet	**3.88e-03**	**5.06e-03**	**1.92e-02**	**2.11e-02**	**4.77e-02**	3.32e-01
NPDE	1.29e-02	1.70e-02	5.54e-02	4.04e-02	5.74e-02	**2.01e-01**
Gated	5.67e-02	7.41e-02	2.93e-01	1.42e-01	2.92e-01	1.33
Ensemble	9.21e-03	1.27e-02	4.29e-02	3.82e-02	6.45e-02	5.29e-01
PolyReg	**6.79e-04**	**9.73e-04**	**5.48e-03**	5.33e-02	1.38e-01	1.46

Table 6: Black-Scholes Example, Prediction Errors

We obtain the same rankings for the gradient errors, with polynomial regresion outperforming the neural network approaches in the Test1 dataset, but DiffNet being the best performer in terms of MAE and RMSE, and NPDE for the Max erorr in the Test2 dataset.

	MAE, Test1	RMSE, Test1	Max, Test1	MAE, Test2	RMSE, Test2	Max, Test2
FFN	2.83e-01	2.51e-01	2.64	1.80	1.64	1.15e+01
ResNet	1.22e-01	1.03e-01	7.26e-01	6.98e-01	5.89e-01	4.98
DiffNet	**5.66e-02**	**4.45e-02**	2.01e-01	**1.84e-01**	**1.46e-01**	1.45
NPDE	1.69e-01	1.32e-01	3.66e-01	2.44e-01	1.87e-01	**4.85e-01**
Gated	3.79e-01	3.14e-01	1.51	6.23e-01	5.38e-01	3.41
Ensemble	1.03e-01	8.42e-02	5.85e-01	3.93e-01	3.28e-01	3.28
PolyReg	**1.98e-02**	**1.60e-02**	**1.75e-01**	7.49e-01	5.86e-01	1.37e+01

Table 7: Black-Scholes Example, Gradient Errors

We define additionally the Mean PDE error to be the absolute value of the averaged signed error in the dataset (whereas the MAE is the average of the absolute errors in the dataset). It may be potentiaally acceptable to have small dynamic arbitrage errors that 'cancel' out throughout the sample space. As may be expected, the errors in the extrapolation dataset Test2 are higher in all cases. In this case, the Neural PDE approach outperforms the other approaches.

We let LowerBound denote the proportion of samples for which $g(\mathbf{X}) \leq (\exp(m) - 1)^+$, Montonoticity to be the proportion of samples for which $dC/dm < 0$, and likewise for TimeValue and Convexity, the proportions for which $dC/d\sigma\sqrt{\tau} < 0$, $\frac{d^2C}{dm^2} < 0$. As described in **??**, the Gated approach has no no-arbitrage in Montonicity, TimeValue, and Convexity for any test set. Interestingly, in Test1 and Test2, the Neural PDE also obtains zero error in TimeValue and Convexity, but not for Monotonicity. The Neural PDE and Polynomial Regression have the first and second lowest errors in the LowerBound error, whereas in the Test2 Dataset, Polynomial Regression and the standard feed-forward are the top two. We note that the Gated network is not guaranteed to be above the intrinsic value lower bound.

	LowerBound	Monotonicity	TimeValue	Convex	1.15e-01	9.28e-02
				FFN		
2.55e-01	4.83e-02	2.25e-01	1.47e-01	3.77e-01	1.39e-01	
ResNet	1.37e-01	3.65e-02	1.41e-01	1.11e-01	2.81e-01	1.04e-01
2.71e-01	1.71e-01					
Differential	1.96e-01	7.52e-02	8.58e-02	6.99e-03	3.75e-01	1.27e-01
3.55e-01	6.94e-02					
NPDE	2.70e-01	1.60e-01	0.00	0.00	1.80e-01	2.72e-01
1.71e-03	0.00					
Ensemble	1.68e-01	8.52e-02	5.18e-02	3.80e-03	2.87e-01	1.47e-01
2.81e-01	9.27e-02					
PolyReg	7.24e-02	4.93e-02	6.66e-02	4.83e-02	2.03e-01	1.22e-01
4.16e-01	1.58e-01					
Gated	1.35e-01	**0.00**	**0.00**	**0.00**	2.50e-01	**0.00**
0.00	**0.00**					

Table 8: Black-Scholes Example, PDE Error

Training time does not necessarily matter in the setting of producing a clone for a pricing function, although ideally we would prefer a shorter training time. However, a shorter training could indicate the training has failed time standard MC / PDE, we cannot necessarily control the error convergence with increased training. In the 'true' setup for creating a clone of a payoff function, we should instead resort to neural architecture search to determine the optimal architecture instead of a fixed architecture. See Appendix Figure 4 for the training curves for this example.

We define the inference time as the time required to produce predictions, compute all first order greeks $\frac{\partial g}{\partial \mathbf{x}}$, and the second order greeks with respect to $\nabla_{\mathbf{x}} \frac{\partial g}{\partial S}$ for the dataset. There may be some small randomness from the CPU, but the results indicate that neural network approximations can indeed offer a significant speedup - the average time per sample would be roughly 1.068×10^{-5} seconds per sample. However, in this application, Polynomial Regression with simple bump and revalue is approximately $\times 6 - 7$ faster than the neural network approaches, and even faster than the analytic formula with AAD for the sensitivities. We also note that the Gated Network is much slower, likely due to the additional complexity required in its implementation.

	No. Parameters $\lvert\boldsymbol{\theta}\rvert$	InferenceTime	TrainTime
FFN	4433	0.857371	18.5371
ResNet	4433	0.674041	21.1667
Differential	4433	0.716184	28.0614
NPDE	4433	0.743568	37.4137
Gated	6484	13.4928	35.2946
PolyReg	36	0.126842	0.117382
Analytic	-	0.1989	-

Table 9: Black Scholes Example, Model Complexity

Summary: In this example, we have seen that neural networks can achieve very fast inference in pricing and first / second order sensitivities, and can potentially achieve some degree of accuracy. However, pricing errors of order 10^{-3} are likely too high for a production setting, and the parameter space may not be sufficiently large. In addition, we could consider combinations of these constructions instead, for example a construction combining ResNet + Differential + NeuralPDE.

On the other hand, polynomial regression outperformed all neural network approaches in interpolation, but had poor extrapolation behaviour. It may also be possible to attain further performance improvements for the basis regression approach; we could consider another basis (e.g. cubic splines) for better extrapolation, implement a differentiable version in `Tensorflow`, or incorporate weight regularisation. It may also be that this example is too simplistic of a setting for neural networks. In the next example, we consider the setting of a more complicated volatility model.

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. [31] used a two step approach.

MC and SDE:

The Rough Bergomi model has dynamics:

$$1 \tag{66}$$

Experiment: Our parameter space consists of:

$\mathbf{S_0}$	$\log(K)$	τ	$\alpha = \frac{1}{2}(H - \frac{1}{2})$	ρ	V_0	ξ
Underlying	Strike	Time-to-maturity	Roughness	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 10: 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 [42], 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 11: 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 12: 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
train_time	NaN	NaN	NaN	21.75

Table 13: Rough Bergomi Example - Model Complexity

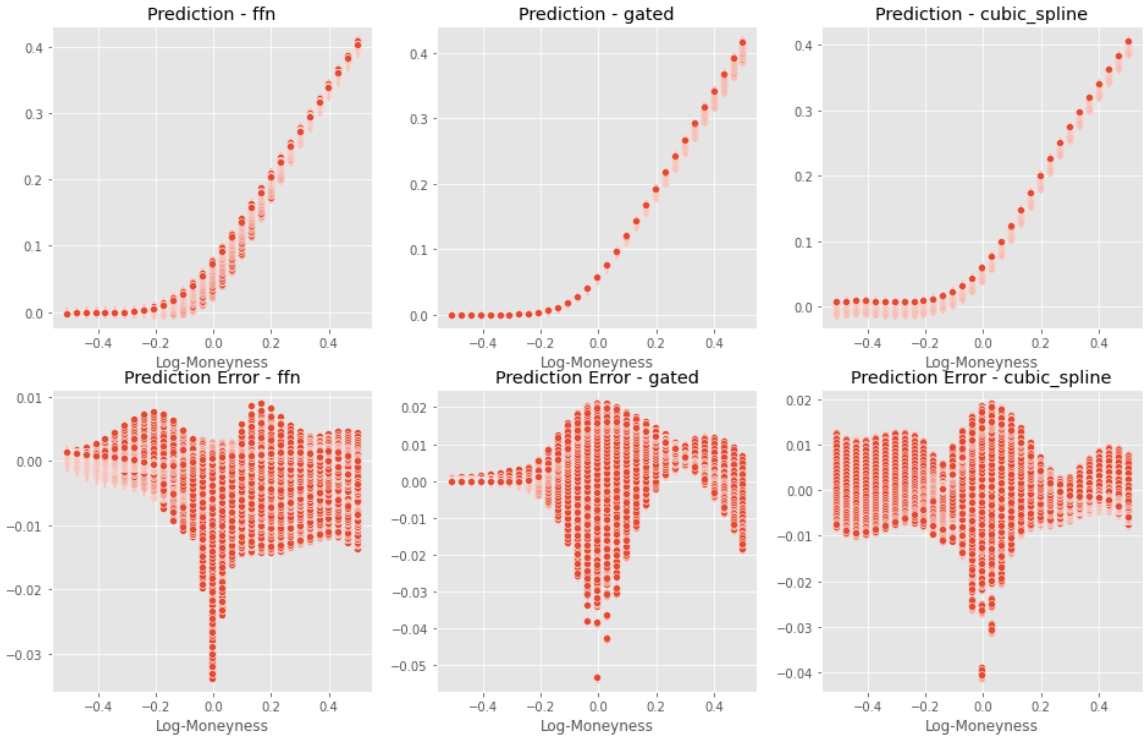


Figure 2: Errors for Rough Bergomi Approximations

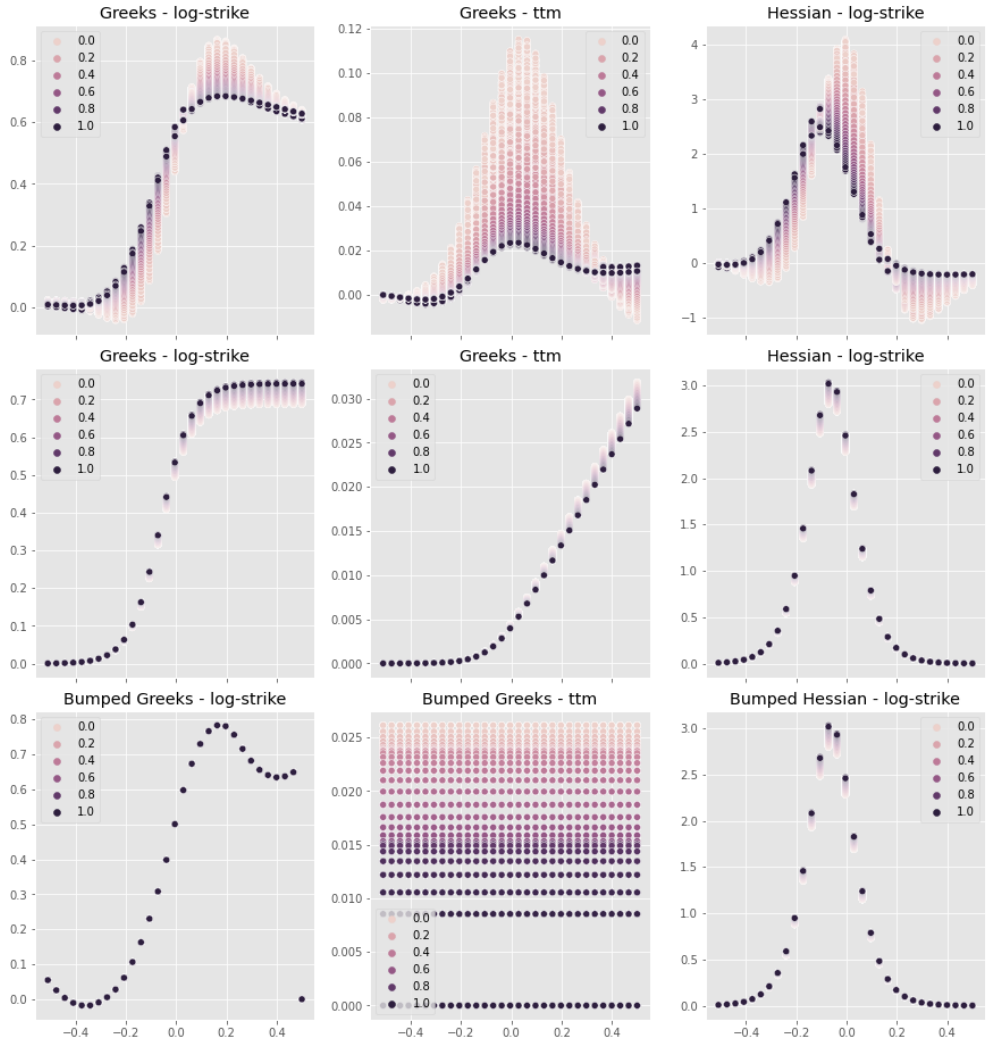


Figure 3: Greeks for Feed-foward, Gated, Cubic-spline respectively

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 [35] and [49]

Basket Option - Arithmetic Brownian Motion

[INCOMPLETE, restructure and fix graphs]

Setup: We consider the first example from [32], 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 from [32]. 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:

$$d\mathbf{S}_t = \mathbf{L}d\mathbf{W}_t, \mathbf{S}_t = \mathbf{S}_0 + \mathbf{L}\mathbf{W}_t, \mathbf{S}_t \sim N(\mathbf{S}_0, \mathbf{L}\mathbf{L}^\top t) \tag{67}$$

$$d\mathbf{X}_t = \mathbf{w}^\top d\mathbf{S}_t = \sigma dW_t^\top, \mathbf{X}_t = \mathbf{w}^\top \mathbf{S}_t, \mathbf{X}_t \sim N(\mathbf{w}^\top \mathbf{S}_0, \mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w}) \tag{68}$$

$$h(\mathbf{w}, \mathbf{S}_t, K) = (\mathbf{w}^\top \mathbf{S}_T - K)^+ = (\mathbf{X}_T - K)^+ \tag{69}$$

$$f(\tau, \mathbf{S}_t, K, \mathbf{w}) = \mathbb{E}[(\mathbf{w}^\top \mathbf{S}_t - K)^+ | \mathbf{S}_t, \mathbf{w}, \tau, K] \tag{70}$$

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 $w_i S_i$, 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:

$$df(T - t, S_t; \ldots) = -\frac{\partial f}{\partial t}dt + \sum_{i=1}^{d} \frac{\partial f}{\partial S_i} dS_{i,t} + \sum_{i=1}^{d}\sum_{j=1}^{d} \frac{1}{2}\frac{\partial V}{\partial S_i S_j} d[S_i, S_j] \tag{71}$$

$$0 = -\frac{\partial f}{\partial \tau} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} \frac{\partial V}{\partial S_i S_j}, f(0, \mathbf{S}) = (\mathbf{w}^\top \mathbf{S} - K)^+ \tag{72}$$

$$df(T - t, X_t; \ldots) = -\frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dX_t + \frac{1}{2}d[X]_t \tag{73}$$

$$= \mathbf{w}^\top \mathbf{L}d\mathbf{W}_t + \frac{1}{2}\mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w}dt \tag{74}$$

$$= -\frac{\partial f}{\partial \tau} + \frac{\sigma^2}{2}\frac{\partial f}{\partial x^2}, f(0, \mathbf{X}) = (\mathbf{X} - K)^+ \tag{75}$$

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:

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

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:

$$
f(\mathbf{X}_0, \sigma, \tau, K) = \sigma\sqrt{\tau}[\phi(d_1) + d_1\Phi(d_1)], d_1 = \frac{\mathbf{X}_0 - K}{\sigma\sqrt{\tau}} \tag{76}
$$

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

$$
\frac{\partial f}{\partial x} = \sigma\sqrt{\tau}[-\frac{d_1}{\sigma\sqrt{\tau}}\phi(d_1) + \frac{1}{\sigma\sqrt{\tau}}\Phi(d_1) + \phi(d_1)\frac{d_1}{\sigma\sqrt{\tau}}] = \Phi(d_1) \tag{77}
$$

$$
\frac{\partial f}{\partial \tau} = \frac{f}{2\tau} + \sigma\sqrt{\tau}[\frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1}{2\sqrt{\tau}}\Phi(d_1)] = \frac{\sigma}{2\sqrt{\tau}}\phi(d_1) \tag{78}
$$

$$
\frac{\partial f}{\partial x^2} = \frac{1}{\sigma\sqrt{\tau}}\phi(d_1) \tag{79}
$$

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 **??**, and a neural network trained on the pathwise differentials 46, 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 $\mathbf{S_{i,T}} = \mathbf{S}_{i,t} + \mathbf{LW_{i,T}}$ exactly from 69. 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 14: 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 15: 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 16: 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 17: Bachelier Basket Example - Gradient Errors

Table 17 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	3.109×10^{-1}	3.182×10^{-1}	6.280×10^{-3}	2.649×10^{-1}	3.216×10^{-1}	0
LowerBound						
Grad	5.622×10^{-2}	1.246×10^{-1}	4.000×10^{-5}	5.514×10^{-2}	4.860×10^{-1}	0
LowerBound						

Table 18: Bachelier Basket Example - No-Arbitrage Violations

Table 18 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 $\times 10^{-2}$	**6.337** $\mathbf{\times 10^{-2}}$	6.739 $\times 10^{-2}$	7.450 $\times 10^{-2}$	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 19: Bachelier Basket Example - Time Complexity

Finally, table 19 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 [32], 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

In our experiments, we have seen that neural networks can achieve higher speeds, but not necessarily any guarantees on convergence. In a low dimension setting of approximating a European call on a single asset under a given volatility model, all neural net constructions underperformed polynomial basis construction in terms of pricing, gradient, no-arbitrage,PDE error and speed, but had better extrapolation properties. However for the setting of offline cloning of a neural network, we used a fixed architecture, whereas the practical approach may be to consider neural architecture search over a range of models, and with a much larger dataset.

In another context of solving a higher multi-dimensional problem on-the-fly, neural networks were able to outperform polynomial regression in terms of both speed, and pricing, no-arbitrage, and gradient errors, but underperformed a simple Monte Carlo evaluation for every path in the state space. However, our experiment may be too simplistic as the problem can be simulated exactly, reduced to a single factor, and only considers,one timestep.

For further improvements, we could consider improving the whole workflow. In our experiments, we considered arbitrary ranges of parameters, but further investigation into dataset construction could be considered. For example, we could consider sampling from more efficient parameter spaces, such as basing them from parameter distributions and parameter correlations from historical data.

In terms of the various construction methods, we saw that incorporating differential and PDE terms into the loss function can lead to increased performance. The hard constraints approach or Gated Network indeed prevents some no-arbitrage conditions, but comes at the cost of much lower expressivity. Empirically, it seems that if pricing , gradient, and PDE errors can be minimised, then no-arbitrage errors may also reduce. In addition, we could consider post-processing of the predicted

In terms of applications, we could consider further investigation into more realistic payoffs, for example high-dimensional callable products such as Bermudan swaptions.

For further comparisons against other methods, we could compare against the Cheybschev / Tensor based approaches of [2], Potz, Glau, as no convenient `Python` implementation exists yet, or investigate efficient implementations of Gaussian processes. Investigation into Quantum Deep Learning and potential applications for derivatives modelling.

A Appendix

Black-Scholes European Call

Training: We note that the Feed-Forward and ResNet terminate training as the train and validation loss appear to plateau. The Gated Network appears to be training, but terminates training with error in the order of 10^{-1}, whereas the other models have order 10^{-2}.

Pricing Errors: In all cases, the approximated function produes an upward sloping curve, but some samples lie below the intrinsic value lower bound. The errors indicate a knot-like pattern, which might be due to the use of or 36 basis functions, or hidden units pattern. The colour denotes the, which suggest that the neural networks do not necessarily capture the relationship between time-to-maturity and price.

Gradient Errors: The Gradient errors with respect seem to be upward sloping, in some cases, which suggests that the neural networks are unable to capture sufficient convexity.

PDE Errors: In general, there appears to be greater errors at the low and high values in the range for moneyness, particularly in the case of polynomial regression. This suggests there may be poor extrapolation across the boundaries of the domain, There seems to be higher errors for low time-scaled implied volatilities for the feed-forward and residual networks

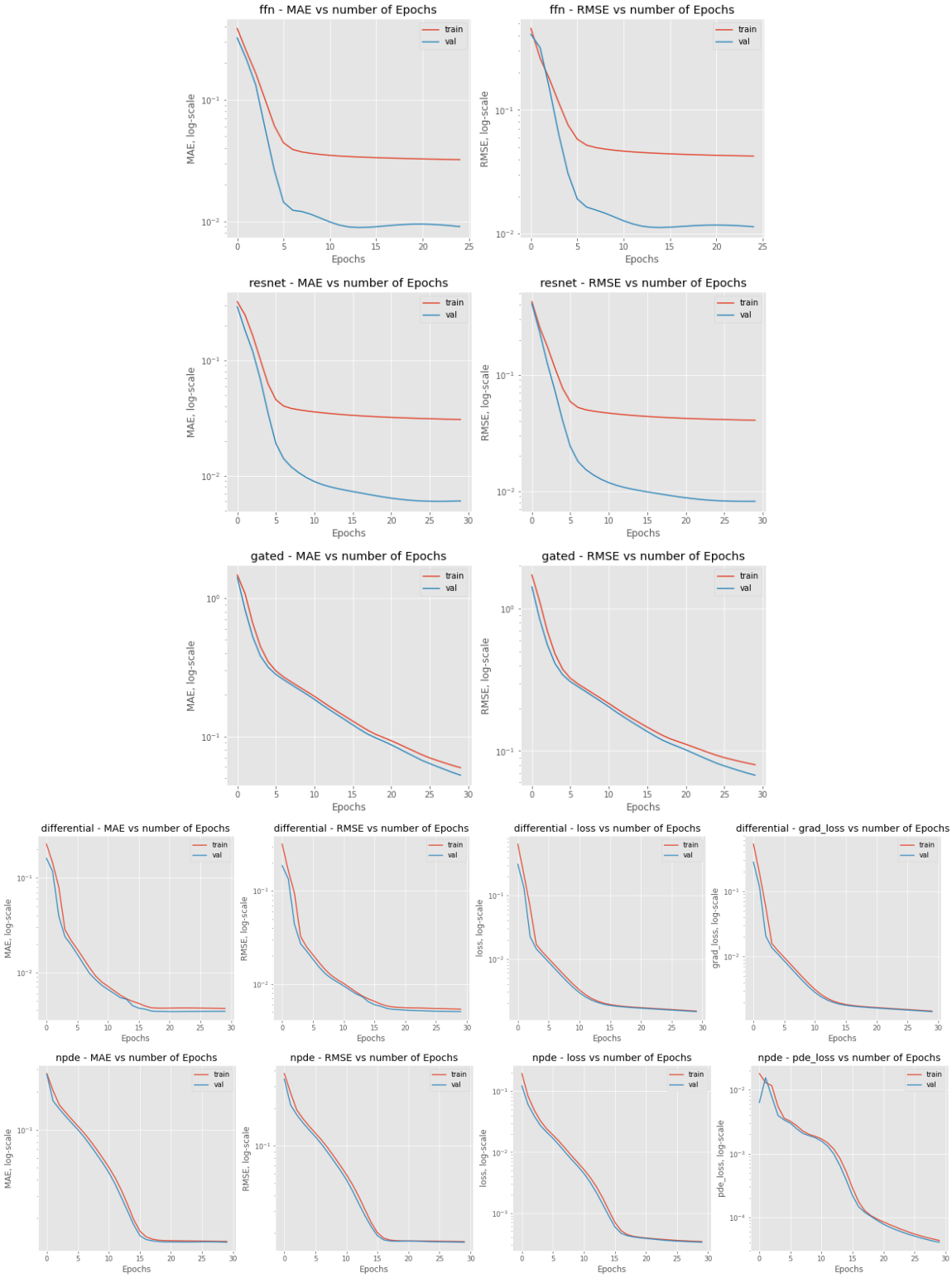


Figure 4: Black Scholes Examples, Training Curves

Figure 5: Black Scholes Example, Feed-Forward Neural Network

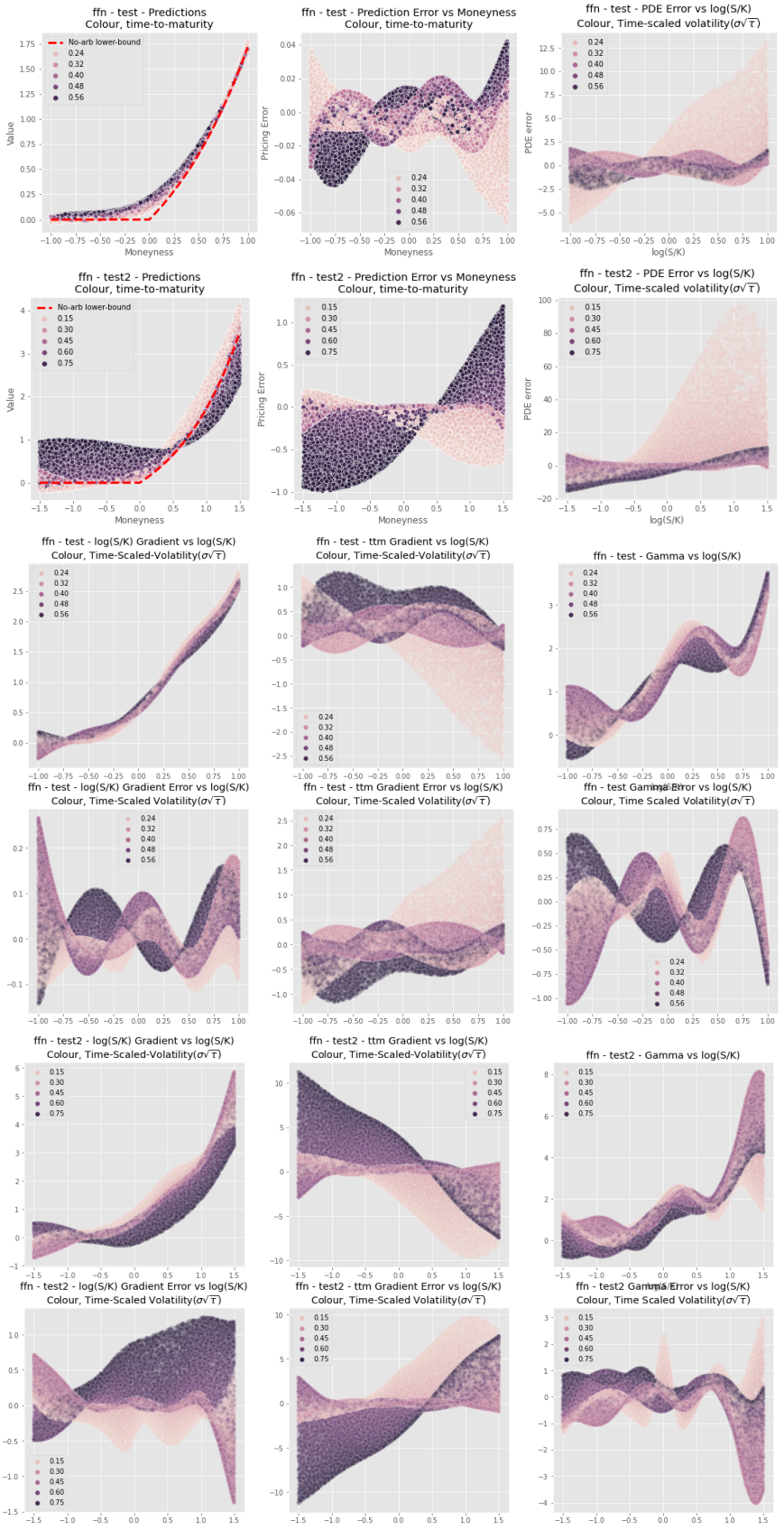


Figure 6: Black Scholes Example, Residual Neural Network

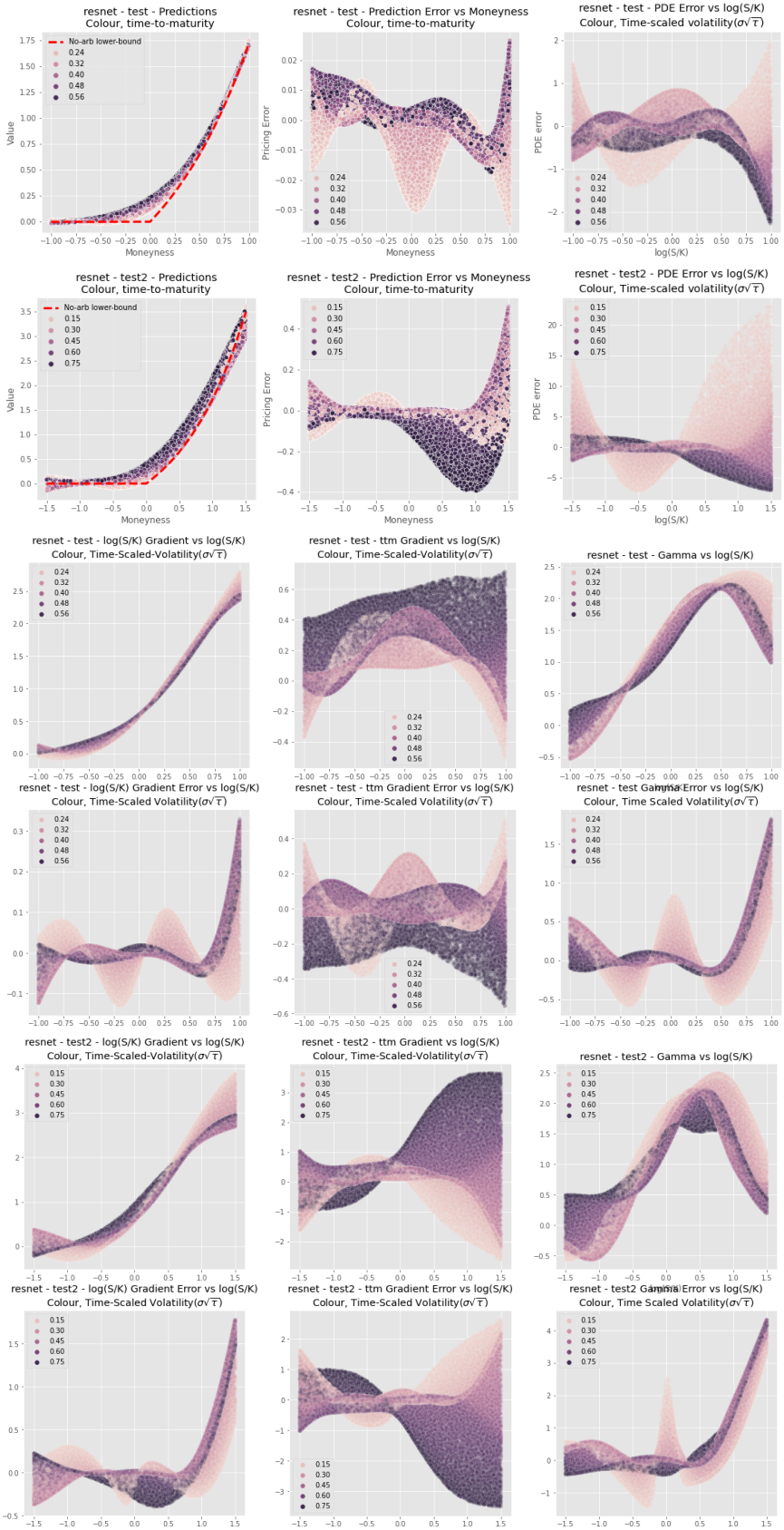


Figure 7: Black Scholes Example, Gated Neural Network

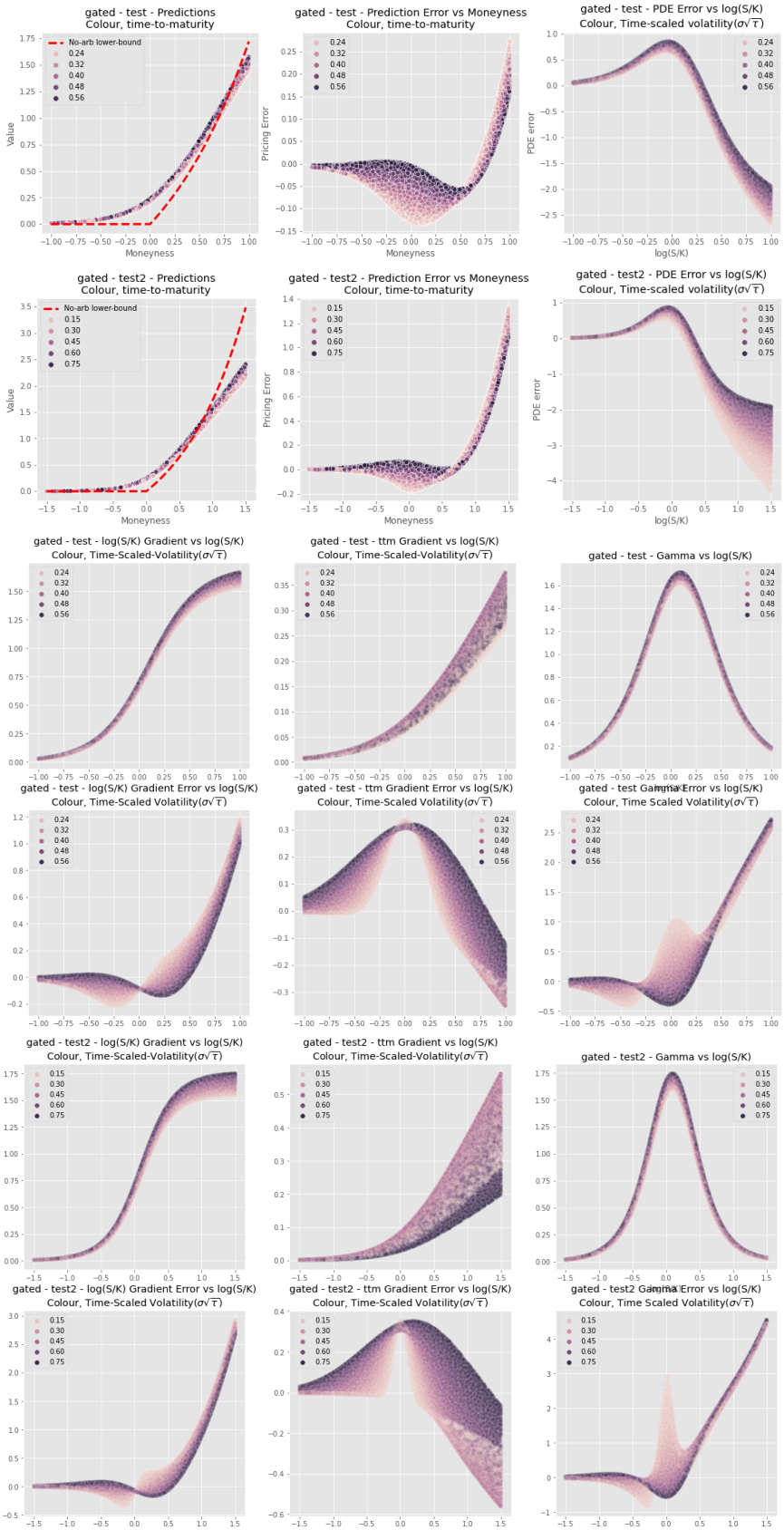


Figure 8: Black Scholes Example, Neural PDE

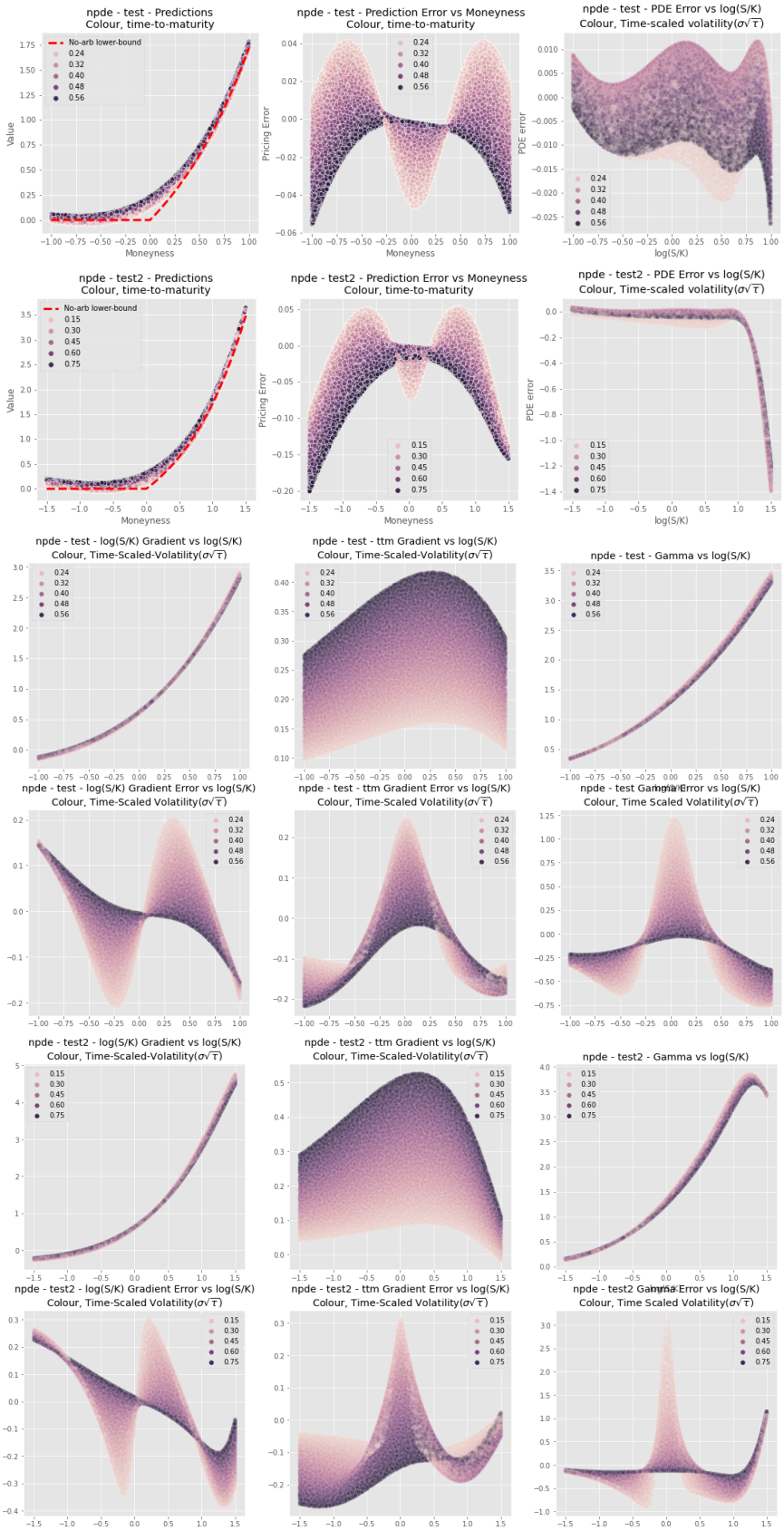


Figure 9: Black Scholes Example, Differential Neural Network

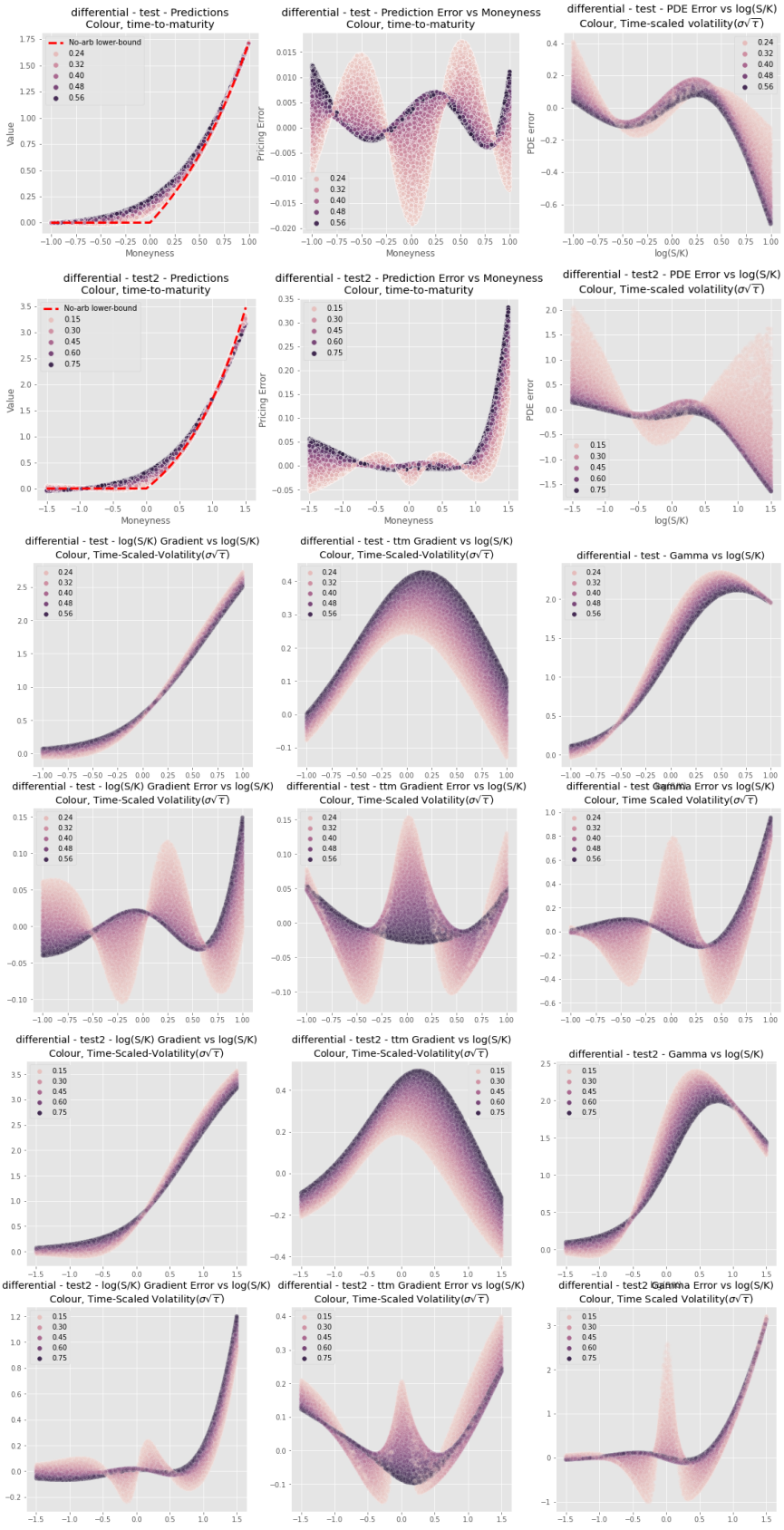


Figure 10: Black Scholes Example, Neural Network Ensemble

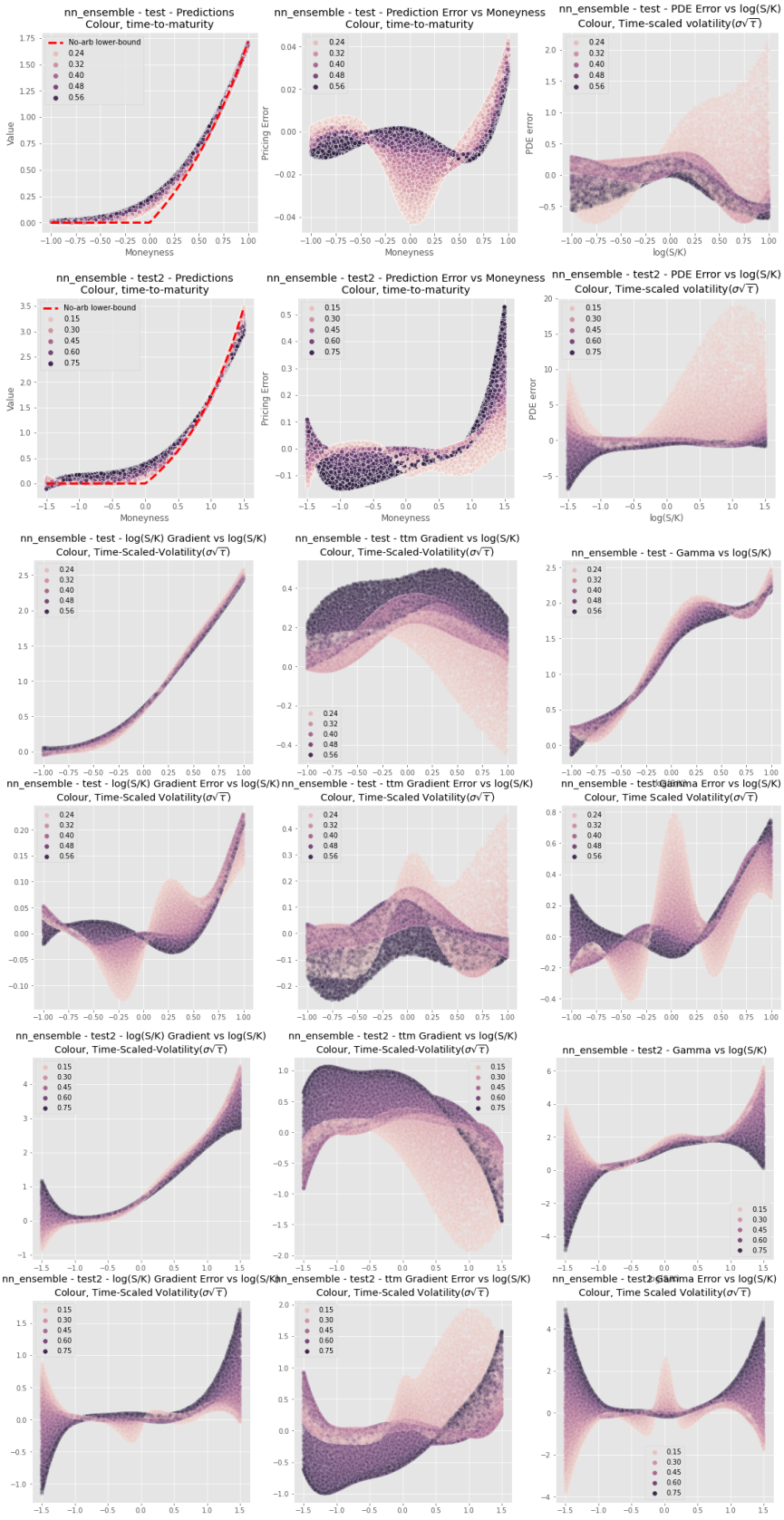
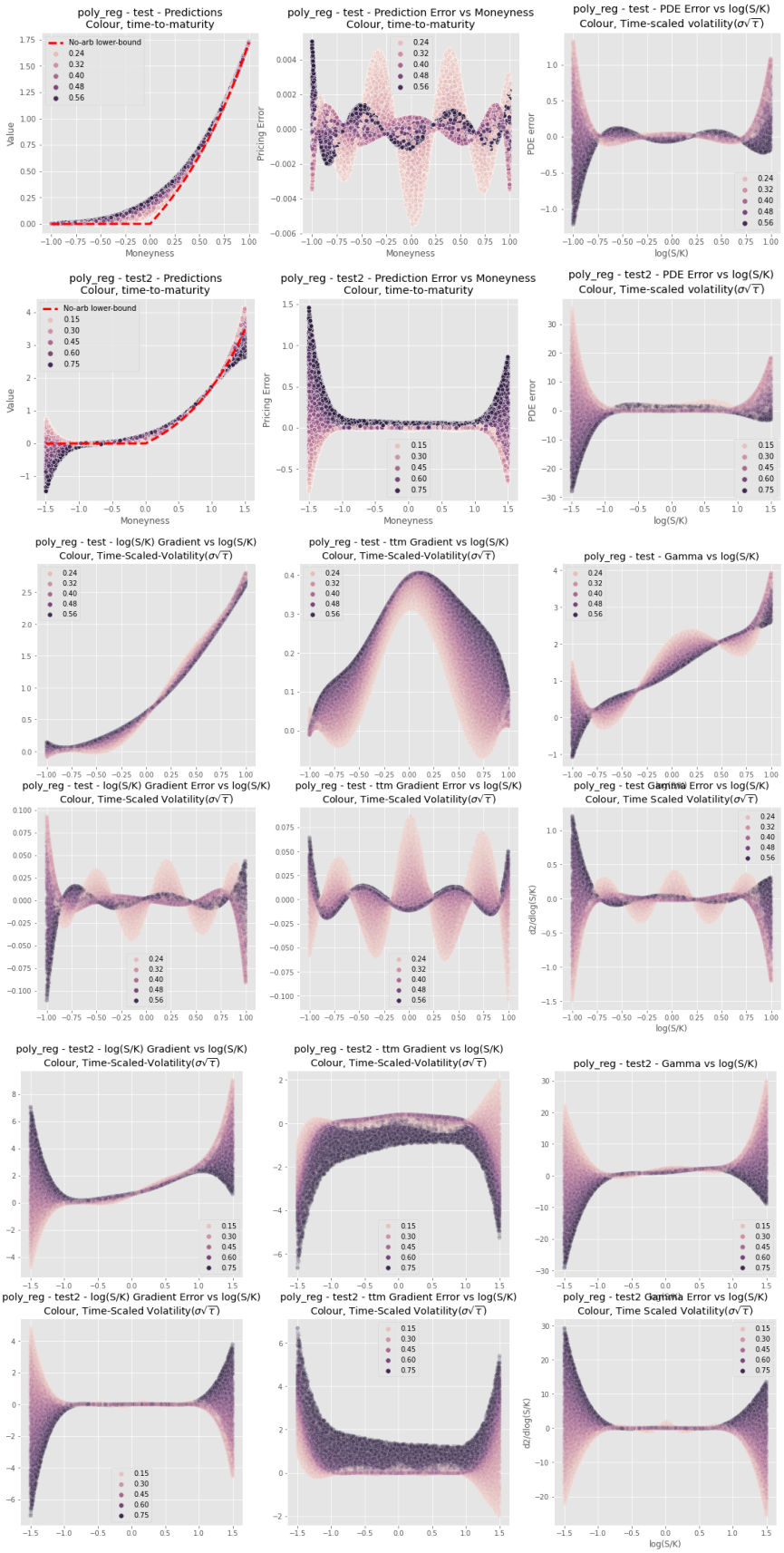


Figure 11: Black Scholes Example, Polynomial Regression



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] Mike Giles and Paul Glasserman. "Smoking adjoints: Fast monte carlo greeks". In: *Risk* 19.1 (2006), pp. 88–92.

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

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

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

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

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

[30] 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.

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

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

[33] 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.

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

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

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

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

[38] 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.

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

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

[41] 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.

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

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

[44] 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.

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

[46] 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.

[47] 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.

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

[49] 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).

[50] Justin Sirignano and Konstantinos Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of computational physics* 375 (2018), pp. 1339–1364.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.