Neural Network Constructions 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

July 1, 2022

Abstract

In this dissertation, we explore various neural network construction methods for the problem of approximating pricing functions and their sensitivities (gradients).

We first review standard no-arbitrage derivatives pricing theory, and consider perspective of pricing as a function of a parametric map from an arbitary volatility model parameters and payoff to the price. We discuss the key numerical methods used (MC / AAD, PDE / FD) to obtain prices and sensitivities, their potential limitations.

We then review neural networks as an alternative to basis function regression, and how various neural network construction techniques can incorporate derivatives pricing principles and prior knowledge. We then consider these methods in numerical experiments: We consider two settings, Black-Scholes, in which the neural network is trained 'offline', then used for inference, and a high-dimensional Basket option in the Bachelier Model, where the neural network is used on-the-fly.

In our experiments, using specific neural network construction methods can lead to improved performance relative to other neural network constructions, in specific criterion: errors in the prices, gradient, pricing PDE, and no-arbitrage adherence. We confirm that neural networks can have efficent inference speeds, but a drawback is that the time-versus-error cannot be explicitly controlled. However, this may be only necessary in an online / on-the-fly setting, whereas in an offline setting training time does not matter we can resort to neural architecture search.

Contents

1 Motivation **1**

2 Preliminaries **4**
Pricing as Conditional Expectation 4
Pricing as the solution to the PDE / Stochastic Control 7
Sensitivities 8
Longstaff-Schwartz / Least Squares Monte Carlo 10
No-Arbitrage Constraints 11
Summary 12

3 Neural Networks **14**
Neural Network Definition 14
Neural Networks as Basis Functions 15
Neural Network Theory and Training 17
Overfitting and Effiicent Training 19
Special Architectures 20
Determining Optimal Architecture 21
Summary 22

4 Neural Networks for Derivatives Pricing **23**
Neural Network Architecture - 'Hard Constraints' 23
Loss Functions for Derivatives Pricing and Risks 24
Summary 27

5 Numerical Experiments **28**
European Call - Black-Scholes 29
Basket Option - Arithmetic Brownian Motion 35
European Call - Rough Bergomi 38

6 Conclusion **39**

A Appendix **40**
Pricing for Non-European Payoffs 40
Activation Functions 41
Machine Learning Workflow for Derivatives Modelling 41
Dataset Construction 42
Interpreting and Monitoring Neural Networks 43
Other Applications of Neural Networks for Derivatives Modelling 44
Alternative Methods 45
Black-Scholes European Call 46
Basket Option - Arithmetic Brownian Motion 57
Rough Bergomi, European Calls 64

References **68**

List of Figures

1 Example workflow for using neural networks in derivatives modelling 42
2 Black Scholes Examples, Training Curves 49
3 Black Scholes Example, Feed-Forward Neural Network 50
4 Black Scholes Example, Residual Neural Network 51
5 Black Scholes Example, Gated Neural Network 52
6 Black Scholes Example, Neural PDE 53
7 Black Scholes Example, Differential Neural Network 54
8 Black Scholes Example, Neural Network Ensemble 55
9 Black Scholes Example, Polynomial Regression 56
10 Feed Forward Network 59
11 Residual Network 60
12 Control Variate 61
13 Differential Method 62
14 Control Variate 63
15 Monte Carlo 64
16 Errors for Rough Bergomi Approximations 66
17 Prices for Feed-Foward, Gated, Basis (Cubic Spline) Regression, respectively 66
18 Greeks for Feed-Foward, Basis (Cubic Spline) Regression, Gated, respectively 67

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/ Invocations	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 be obtained in some settings, given explicit convergence rates for these methods.

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 [54]. 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).

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.

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. However, the ideal approximating functions should satisfy several properties. Firstly, the inference time should be faster than the original method, and ideally have a comparable pricing accuracy. 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).

Neural networks are one machine learning method that could be used for approximation. 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, and some leading to industrial applications include: [20] of RiskFuel / Scotiabank, and [35] of Danske Bank, respectively, who used neural network approximations of pricing functions to speed up XVA calculations, and [6] of JP Morgan proposed using neural networks to learn pricing and hedging strategies, applying them 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, different volatility models and payoffs and different criterion: the pricing error, gradients error, no-arbitrage adherence, latency, and performance on a given downstream task.

This dissertation is structured as follows:

- **Chapter 2 - Pricing and Sensitivities**: Relevant mathematical formulations for derivatives pricing: pricing as a conditional expectation and PDE, various methods to obtain sensitivities (Finite Differences, AAD), Least Squares Monte Carlo, no-arbitrage.
- **Chapter 3 - Neural Networks**: Review of neural networks from a machine learning perspective, and why their properties may make them suitable as pricing function approximations for derivatives pricing and risks.
- **Chapter 4 - Neural Networks for Derivative Pricing**: Review of literature on different approaches for neural networks for derivatives pricing / sensitivities approximation, and how different constructions can incorporate and leverage the (no-arbitrage) pricing principles from Chapter 2.
- **Chapter 5 - Numerical Experiments**: We evaluate the various neural network construction methods from Chapter 4, and examine their behaviour in various numerous settings.
- **Chapter 6 - Conclusion**: Review of results and potential next steps for further exploration.

Reader's Guide: Readers familiar with derivatives pricing can skip directly to Chapter 3. Readers also familiar with Neural Networks from a machine learning perspective can skip directly to Chapter 4. Supplementary materials and figures can be found in the Appendix.

2 Preliminaries

In this section, we review the standard formulations of no-arbitrage derivatives pricing and risk calculations based on references such as[58], [11], [26].

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)**.**

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

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 or *market generator* model. 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)**.** *Suppose $h : \mathcal{F} \to \mathbb{R}$ is a Borel-measurable payoff function. 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 $\mathbb{Q}$:*

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

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

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

We consider the most basic family of derivatives in Definition 1: A European option entitles the holder to receive some cashflow or payment $h(\mathbf{S}_T)$ that is a function

of the value of the underlying assets $\mathbf{S}_t$, at the maturity date of the contract $t+\tau = T$. In Definitions 1, we assume that the conditional expectation is well-defined for the valid ranges of $\tau, \mathbf{S}_T, \mathbf{X}$. We also 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 $\mathbf{X}$, then we obtain the second equality, which allows us to we can express the no-arbitrage price as a conditional expectation on $\mathbf{X}$.

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, and the volatility model parameters are the volatility σ and the value of the asset F_t. Thus in general, a pricing function requires two kinds of input: the parameters of the volatility model (which also includes the values of the underlying assets $\mathbf{S}_t$), and the parameters of the payoff structure. 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.

Remark. *Although the neural network approach can be utilised on any general payoff and volatility model, we focus on European options, and specically European Calls for brevity. More generally, we could consider exotic payoffs such as path-dependent or callable payoffs, which we discuss in (Appendix A).*

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

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

If we know the $\mathbb{Q}$-transition density $p(T, \mathbf{S}; \mathbf{S}_t, \mathbf{X})$ of $\mathbf{S}_T$ conditioned on $\mathbf{S}_t$ (and the fixed $\mathbf{X}, \tau$), in other words, the distribution of $\mathbf{S}_T$ given $\mathbf{X}$, 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, by considering the mean of sample payoffs:*

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

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* $h(\mathbf{S_{i,T}})$.

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 $O(\frac{1}{\sqrt{N}})$ arising from the Central Limit Theorem. However, the convergence rate is independent of dimension, thus MC can be used for high-dimensional payoffs. However, 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 in the case of SABR (Example 1). This necessitates approximation methods for the SDE, such as the Euler and Milstein schemes through the discretisation of the timesteps S_{i,τ_j} for $j = 1, \ldots, N_\tau$, which leads to a complexity of $O(d \times N_\tau = O(\text{Dimensionality} \times \text{Timesteps}))$ for a single SDE sample [26].

To summarise, the MC method is powerful in that it is flexible and can be used in high dimensions. However, it is computationally expensive in that we need to simulate many samples of the SDE to achieve a target error. In a parametric pricing context, and computing over all $\mathbf{X}$, we can compute the prices for European calls over some grid of time-to-maturity and strike (τ, K) by simulating evaluating the Call price at each K and timestep. However, we need to re-simulate the paths if we were to choose a different choice of volatility model parameters, in this case σ and value of the underlying S_0, and also if we change the grid of values for (τ, K).

As opposed to pricing via evaluating the conditional expectation 1, an alternative approach is to consider the corresponding *partial differential equation* (PDE) of a volatility model and payoff. First, we consider the principle of no-dynamic-arbitrage. (Definition 4)

Definition 4 (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).*

Pricing as the solution to the PDE / Stochastic Control

Definition 5 (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:*

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

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

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

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

Where $\mathbf{H}$ denotes the Hessian and $\mathcal{L}$ denotes the PDE operator, or generator, for the SDE. In the above, we have $\mathcal{L} = \frac{1}{2}Tr(\sigma^\top \mathbf{H}_f \sigma)$, although in the general case (e.g. under a change of variables) $d\mathbf{S}_t$ may have a drift term under $\mathbb{Q}$.

If we consider a portfolio with position $-\nabla_S$ in each of $\mathbf{S}_t$, and a position of 1 in the payoff Y_t. Then we have

$$\begin{aligned}
dY_t - \nabla_S \cdot dS_t &= \left(-\frac{\partial f}{\partial \tau} + \mathcal{L}f\right)dt \\
(Y_T - Y_t) - (\nabla_S \cdot (\mathbf{S}_T - \mathbf{S}_t)) &= \int_t^T \left(-\frac{\partial f}{\partial \tau} + \mathcal{L}f\right)dt \\
\mathbb{E}^{\mathbb{Q}}\left[(Y_T - Y_t) - \int_t^T \nabla_{\mathbf{S}} d\mathbf{S}_u | \mathbf{X}\right] &= \mathbb{E}^{\mathbb{Q}}\left[\int_t^T -\left(\frac{\partial f}{\partial \tau} + \mathcal{L}f -\right)dt | \mathbf{X}\right] \\
\mathbb{E}^{\mathbb{Q}}[Y_T] - Y_t &= \mathbb{E}^{\mathbb{Q}}\left[\int_t^T -\left(\frac{\partial f}{\partial \tau} + \mathcal{L}f\right)dt | \mathbf{X}\right]
\end{aligned} \tag{8}$$

We use the property that $\mathbf{S}_t$ is a $\mathbb{Q}$-local martingale, then $\mathbb{E}^{\mathbb{Q}}[-\int_t^T \nabla_{\mathbf{S}} d\mathbf{S}_u | \mathbf{X}] = 0$. In order for $f(t, \mathbf{S}_t, \mathbf{X})$ to be free of dynamic arbitrage, the equivalent stochastic process $Y_t|\mathbf{X}$ must be a $\mathbb{Q}$-local martingale. Hence hence the drift term in Equation 5 must be zero $-(\frac{\partial f}{\partial \tau} + \mathcal{L}f) = 0$. Otherwise, we long the dynamically hedged portfolio $dY_t - \nabla_{\mathbf{S}} d\mathbf{S}_t$ when $\mathbb{E}^{\mathbb{Q}}\left[\int_t^T -(\frac{\partial f}{\partial \tau} + \mathcal{L}f)dt | \mathbf{X}\right] > 0$, and short the dynamically hedged portfolio if $\mathbb{E}^{\mathbb{Q}}\left[\int_t^T -(\frac{\partial f}{\partial \tau} + \mathcal{L}f)dt | \mathbf{X}\right] < 0$ to obtain a non-negative terminal wealth in expectation, and thus $\mathbb{Q}$ almost surely.

Hence a sufficient condition for the pricing function $Y_t|\mathbf{X} = f(t, \mathbf{S}, \mathbf{X})$ to be free of dynamic arbitrage is to satisfy the corresponding pricing PDE: $(-\frac{\partial f}{\partial \tau} + \mathcal{L}f) = 0$. This corresponds with the Feynman-Kac Theorem (Definition 6).

Definition 6 (Feynman-Kac Formula)**.** *The solution to the PDE* $\frac{\partial f}{\partial \tau} = \mathcal{L}f$ *is given by the conditional expectation:*

$$f_\tau = \mathcal{L}f, f(0, \mathbf{s}; \mathbf{X}) = h(\mathbf{s}) \quad \forall \tau \in [0, T], \mathbf{X} \in \mathcal{X}, s \in \mathcal{S} \tag{9}$$

$$f(\tau, \mathbf{s}; \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T; \mathbf{X})|\mathbf{S}_t = \mathbf{s}, \mathbf{X}] \tag{10}$$

Lastly, we can also formulate the pricing problem as a stochastic control (and also as a Forward-Backward Stochastic Differential Equation)

Definition 7 (Pricing as Stochastic Control)**.** *We can price by determining the optimal initial price* Y_t *and hedging strategy* $\nabla_{\mathbf{S}}$ *over* $[t, T]$ *for some norm* $\|\|$*:*

$$\inf_{Y_t, \nabla_{\mathbf{S}}(u, \mathbf{S}_u, \mathbf{X}))_{u \in [t,T]}} \left\| \mathbb{E}^{\mathbb{Q}} \left[Y_T - \int_t^T \nabla_{\mathbf{S}} \cdot d\mathbf{S}_u | \mathbf{X} \right] - Y_t) \right\|, Y_T = h(\mathbf{S}, \mathbf{X}) \tag{11}$$

Thus in addition to evaluating the pricing function as a conditional expectation (Definition 1), we can consider it as the solution to a PDE (Definition 6), or a stochastic control problem (Definition 11). Having described different approaches to compute prices $f(\mathbf{X})$, we now turn to different approaches for computing sensitivities $\frac{\partial f}{\partial \mathbf{X}}$:

Sensitivities

For market makers and traders, obtaining the sensitivities, or the differentials of the pricing function f with respect to the input parameters $\mathbf{X}$, is also of interest. The most straightforward application, is that the first order differentials $\frac{\partial f}{\partial \mathbf{X}}$ express a linearised approximation to the derivative price when the input parameters change $\mathbf{X}$, and can thus serve as a measure of key risks. In addition, the sensitivities $\nabla_{\mathbf{S}}$ represent the idealised hedging strategy for the volatility model in continuous time, as in Definition 11. However, given that f is not known analytically known, we need some approximation the differentials of f as well.

Definition 8 (Finite Differences, 'Bump-And-Revalue')**.** *Suppose we have the pricing function* f*, then we can approximate the derivatives with some* $\epsilon > 0$*:*

$$\frac{\partial f}{\partial X_i} = \frac{f(\cdots, X_i + \epsilon, \cdots) - f(\cdots, X_i, \cdots)}{\epsilon}$$

$$\frac{\partial^2 f}{\partial X_i \partial X_j} = \frac{f(\cdots, X_i + \epsilon, X_j + \epsilon, \cdots) - f(\cdots, X_i, X_j + \epsilon, \cdots)}{\epsilon^2} - \frac{f(\cdots, X_i + \epsilon, X_j, \cdots) + f(\cdots, X_i, X_j, \cdots)}{\epsilon^2}$$

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 9 (Finite Differences for PDE)**.**

$$\frac{\partial f}{\partial \tau} = \mathcal{L}f, \quad \frac{g(\tau + \Delta\tau, \cdots) - g(\tau, \cdots)}{\Delta\tau} = \mathcal{L}(\Delta x)g, f(0, \mathbf{S}) = g(0, \mathbf{S}) = h(\mathbf{S}; \mathbf{X}) \quad (12)$$

Finite Differences is a key method to solve the pricing problem as a PDE formulation (Equation 9). In effect, we solve the PDE with finite difference approximation by approximating the differentials with finite-differences as in Definition , using a discretisation Δt for the time-to-maturity, and Δs for each of the state variables or underlying assets / factors in $\mathbf{S}$.

This leads to several advantages with PDEs: 1) we also compute the sensitivities (although they may be inaccurate and with a different order of convergence than the order for the price f), with respect to time t and the values of the underlying factors $\mathbf{s}$. Secondly, 2) The solution depends deterministically on the discretisation $(\Delta, \Delta x)$ as opposed to random sampling error in the MC approach. However, the complexity grows with the dimensionality d, and the number of underlyings in S_t, such that PDEs may be too computationally expensive in some cases for high-dimensional problems. Finally 3), the PDE solution 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 $\{0, \Delta\tau, \ldots, N_\tau\Delta\tau = T\}$, but not for the other volatility model parameters and contract parameters in $\mathbf{X}$. Thus in the context of parametric pricing, and if we want a sensitivity with respect to the other or contract parameters, the PDE solver must be reinvoked for new or multiple contract or volatility model parameters.

The finite differences is applicable for both MC and PDE. We now describe some methods for computing sensitivities in the MC setting.

Definition 10 (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 differentials with:*

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

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 [25].

Definition 11 (Adjoint Automatic Differentiation)**.**

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

In 13, we compute the sensitivities with forward differentiation, whereas in 14 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 [25],

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.

Longstaff-Schwartz / Least Squares Monte Carlo

We now consider a baseline method to approximate a given pricing function f and all its differentials with respect to $\mathbf{X}$, over a given range of $\mathbf{X}$. [47] [62] considered approximating pricing functions by approximating the conditional expectation 1 using basis function regression.

Definition 12 (Least Squares Monte Carlo)**.** *Let* Z_i *denote* $i = 1, \ldots, N_B$ *basis functions,* w_i *he weights. Let* $\mathbf{X} \in \mathbb{R}^{N_F}$ *be a matrix denoting* N *sample parameters (vol model, state, contract), and* $\mathbf{y} \in \mathbb{R}$ *be a vector containing the corresponding sample payoffs* $y_j = h(\mathbf{S}_{j,T})|\mathbf{X}_j, j = 1, \ldots, N$

$$g(\mathbf{X}) \approx \mathbb{E}^{\mathbb{Q}}[\mathbf{y}] = \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{X}] \tag{15}$$
$$\mathbb{E}^{\mathbb{Q}}[h(\mathbf{X})|\mathbf{X}], g(\mathbf{X}) = \mathbf{Z}(\mathbf{X})\mathbf{W}, \quad \mathbf{Z}(\mathbf{X}) \in \mathbb{R}^{N \times N_B} \tag{16}$$
$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \|(\mathbf{y} - \mathbf{Z}\mathbf{W})^2\|^2 \approx \arg\min_{\mathbf{W}} \mathbb{E}[(\mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{X}] - \mathbf{Z}\mathbf{W})^2\|^2 \tag{17}$$
$$\mathbf{W}^* = (\mathbf{Z}^\top \mathbf{Z})^\top \mathbf{Z}^\top \mathbf{y} \tag{18}$$

Remark. *The application of [47] is toward pricing American / Bermudan options (Appendix A). They fix the contract parameters and volatility model parameters, and only* $\mathbf{S_t}$ *is varied (hence* $\mathbf{X} = \mathbf{S}_t$*). For our application, we consider the one-timestep-case, which corresponds to Europeans, and consider a* $\mathbf{X}$ *with varying volatility and contract parameters as opposed to just* $\mathbf{S}_t$*.*

Least Squares Monte Carlo sits in the MC family, but is used to obtain an approximation over multiple values of the underlying state $\mathbf{S}_t$ as opposed to a single set. Basis regression has several properties. 1) There are certain theoretical guarantees,

such as the Stone-Weierstrass Theorem. 2) The optimal weights $\mathbf{W}$ and bias can be determined by the least squares solution 18 (supposing $\mathbf{Z}^\top\mathbf{Z}$ is non-singular), 3) For convergence, [47] argues that as the number of basis functions B to infinity, g approximates the true value function (and $N \to \infty$ for MC). To compute the sensitivities, we can apply bump-and-revalue 2 or compute the gradients analytically - however, *all parameters we want sensitivities for, or functions of these parameters, must be in* $\mathbf{X}$. For selecting the basis functions [47], considers Z_i such as Chebyshev and Legendre polynomials, however, any general basis could be considered, such as the Karhunone-Louvre basis in [61]. However basis regression may not necessarily be scalable to high dimensions. In particular, if specify a k-th degree polynomial regression for a N_F-dimensional, the number of basis functions grows at $\binom{N_F+k}{k}$.

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 the approximation g solves the corresponding pricing PDE of f. However, in addition we must consider no static arbitrage.

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 [21] and [10], and 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.

$dg/d\tau \geq 0$	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 [21] and [10]

Definition 14 (Breeden-Litzenberger, Risk Neutral Density)**.** *Suppose that the conditional expectation, and risk-neutral transition density p are well-defined. Differen-*

tiating under the integral, we can extract the transition density:

$$g(T,K) = \int_{\mathbb{R}} (y-K)^+ p(y,T;s,t)dy = \int_K^\infty (y-K)p(y,T;S_t,t,\mathbf{X})dy \tag{19}$$

$$\frac{dg}{dK} = -\int_K^\infty p(y,T;S_t,t,\mathbf{X})dy = -\mathbb{E}^{\mathbb{Q}}[1_{S_T>K}|\mathbf{X}] = -\mathbb{Q}[S_T > K] \tag{20}$$

$$\frac{d^2g}{dK^2} = p(K,T;S_t,t,\mathbf{X}) \tag{21}$$

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 $\frac{d^2g}{dK^2}$ (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:

$$\lim_{K\to\infty} \frac{dg}{dK} = 0, \lim_{K\to 0} \frac{dg}{dK} = -1 \tag{22}$$

From $\frac{d^2g}{dK^2}$, 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:*

$$g^{CALL} - g^{PUT} = \mathbb{E}^{\mathbb{Q}}[(S_T-K)^+ - (K-S_T)^+] = \mathbb{E}^{\mathbb{Q}}[S_T - K] = (S_t - K) \tag{23}$$

Derivatives prices must be also consistent with other prices. In the context of European options, we have the put-call parity relation (23). 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 an open question is how to incorporate these no-arbitrage constraints into an 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 MC 3 and obtain sensitivities via AAD 14, solve the PDE 9 with finite differences **??**, or use numerical integration 2
- These procedures can be computationally expensive, particularly in the case of high dimensionality.
- For one evaluation of MC, we can obtain prices for a range of time-to-maturities and (τ, K), and the sensitivites with respect to all parameters with AAD $\frac{\partial f}{\partial \mathbf{X}}$
- For one evaluation of a PDE solver we obtain prices over time-to-maturities and states (asset prices) $(\tau, \mathbf{s})$, and differentials with respect to $\tau, \mathbf{s}$ only.
- Both MC and PDE must also be re-invoked when (some of) the volatility model and contract parameters in $\mathbf{X}$ change.

- Basis function regression 18, (Least Squares Monte Carlo) can be used as an approximation, such that we can evaluate $g(\mathbf{X})$ and its differentials $\frac{\partial g}{\partial \mathbf{X}}$ analytically over all $\mathbf{X}$.

In the next section, we define neural networks from a machine learning perspective 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 sensitivities problem. A comprehensive outline of neural networks is found in [28], and a reference on practical implementations using `Tensorflow/Keras` can be found in [13]. An overview of neural networks and their applications towards finance can be found in [18].

Neural Network Definition

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 input, 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 generic machine learning problems, the true mapping function $\mathbf{y} = f(\mathbf{X})$ may be unknown, although in the context of derivatives modelling, we know f, but not in analytic closed-form.

Definition 16 (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*

$$\begin{aligned} \mathbf{Z}^1 &= g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{1}(\mathbf{b}^1)^\top) \\ \mathbf{Z}^2 &= g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{1}(\mathbf{b}^2)^\top) \\ &\vdots \\ g(\mathbf{X}; \boldsymbol{\theta}) &= g_{N_H}(\mathbf{Z}^{N_H-1}\mathbf{W}^{N_H} + \mathbf{1}(\mathbf{b}^{N_H})^\top) \end{aligned} \tag{24}$$

Where $\mathbf{W}^i \in \mathbb{R}^{H_{i-1} \times H_i}$ are the real-valued weight matrices for the i-th hidden layer, and $\mathbf{b}^i \in \mathbb{R}^{H_i}$ are the column vectors or bias term for the i-th hidden layer. The product $\mathbf{1}(\mathbf{b}^i)^\top \in \mathbb{R}^{N \times H_i}$ represents adding $\mathbf{b}^i$ to each column of $\mathbf{Z}^{i-1}\mathbf{W}^i$. The activation functions $g_i, i = 1, \ldots, N_H$, for the i-th hidden layer, functions are applied element-wise to the inputs, such that $g_i(\mathbf{Z}_{i-1})_{j,k} = g_i(\mathbf{Z}^{i-1}_{j,k})$.

Remark. *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, [42] mentions frameworks such as the NVIDIA TensorRT, which can further be used to speedup inference of a trained neural network and [30] discusses some considerations to further speed up neural network inference from the framework level.*

Activation Functions: 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 $y_i \in [0,1]$, (e.g. a digital) we could consider applying a final non-linear transformation g_n to constrain output to $[0,1]$. Table 19 displays some common activation functions and their first- and second-order gradients for a single input $x \in \mathbb{R}$.

Automatic Differentiation: As described in Chapter 2, automatic differentiation enables fast computation of gradients, given an *adjoint implementation* of a given function. Adjoint implementations of the activation functions g_i in the underlying deep learning framework of choice lead to efficient evaluations of the gradients of the neural network ∇g, forgoing the need to compute the gradients analytically. Efficient gradients are needed to compute the sensitivities with respect to the neural network parameters $\nabla_{\boldsymbol{\theta}}$ when training the neural network, but can also be applied to obtain efficient gradients for the sensitivities with respect to the input $\frac{\partial g}{\partial \mathbf{X}}$.

Remark. *AAD motivates the potential for using neural networks to obtain fast first-order, and potentially higher-order differentials for derivatives modelling. [54] 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 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.

Example 4 (Neural Networks as basis function regression)**.** *Suppose* g_{N_H} *is the identity function, and all parameters except* $\mathbf{W}^{N_H}$ *and* $\mathbf{b}^{N_H}$ *are fixed. Then the columns of* $\mathbf{Z}^{N_H-1}$ *represent fixed basis functions, and the neural network corresponds to basis function regression.*

$$\mathbf{Z}^{N_H-1}(\mathbf{X})\mathbf{W}^{N_H} + \mathbf{1}(\mathbf{b}^{N_H})^\top \tag{25}$$

Another way of viewing a neural network may be to consider it as non-parametric 'learning' of a flexible 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. [35] and [32] use this construction (4) to draw comparison with the Least-Square Monte Carlo method (Equation 18), and in this case the theoretical guarantees for linear regression apply. However, neural networks are flexible in that the basis functions $\mathbf{Z}^{N_H-1}$ (the parameters $\boldsymbol{\theta}$) are *not* fixed, and can be determined based on the dataset and objective. However, this leads to a lack of guarantees in convergence.

Neural Networks as dimensionality reduction: In addition, we could also consider neural networks as a non-linear form of PCA or dimensionality reduction. If we select the final hidden layer size to be less than the input dimension $H_{N-1} < N_F$, the basis functions $\mathbf{Z}^{N_H-1}(\mathbf{X})$ 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.

Multi-output neural networks: Another implication is that we could consider a multi-output neural networks as a composition of neural networks one-dimensional output with a shared set of efficient basis functions. In Equation 16, 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:

$$g_n(\mathbf{Z}^{N_H-1}\mathbf{W}^{N_H}) = \begin{pmatrix} g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,1} + \mathbf{b}) & \cdots & g_n(\mathbf{Z}^{N_H}\mathbf{W}^{N_H}_{;,H_{N_H}} + \mathbf{b}) \end{pmatrix} \tag{26}$$

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, [49] considers using a neural network with a $H_{N_H} = 10$-dimensional output of SABR-implied volatilities at fixed strikes, and [34] 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 contrasts with standard basis functions regression, where the same collection of basis functions may not be optimal for each of the H_{N_H} outputs, and thus a regression per output may be needed.

Example 5 (Ensemble)**.** *Suppose g_{N_H} is the identity function. Then an ensemble of N_E neural networks, with weighting $\alpha_i, i = 1, \dots N_E$ can be expressed as:*

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

Example 6 (Neural Network Confidence Intervals)**.** *We can obtain confidence intervals from N_E neural networks, by considering:*

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

One way to address the non-determinism and uncertainty in choice of neural networks could be consider an *ensemble* of neural networks instead of a single one (Example 5). In this regard, we use a MC estimator of neural networks with weightings α_i (which could be fixed e.g. $\alpha_i = \frac{1}{N_E}$, or further determined). We could also think of this as a new neural network, with a penultimate dimension $N^*_{H-1} = N_E \times N_{H-1}$. Naturally, the drawback of this approach is that it may lead to a more complex implementation, and reduce inference speed, given the need to evaluate n neural networks instead of 1 (although this could be parallelised). Uncertainty bounds can also be

obtained by considering multiple neural networks (although in general only a single price may be needed). The authors of [24], consider multiple random initialisations to obtain a confidence interval on prices.

Transfer Learning: Transfer learning is a machine learning technique that can enable faster training of a model, using another pre-trained neural network. In the derivatives pricing context, [3, 23] explore this application; we assume that the learnt basis functions in the final layer $\mathbf{Z}^{H_N-1}(\mathbf{X})$ in a neural network for European calls may also be useful for European puts, and then retrain and determine the optimal weights for the basis functions $\mathbf{W}^{\mathbf{N}_H}$ for the new pricing function. This corresponds exactly to Example 4.

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.

Definition 17 (Universal Approximation, Hornik (1990) [33])**.** *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 [33] 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.

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).

Definition 18 (Neural Network optimality as loss function minimisation)**.** *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. Let $\boldsymbol{\theta}$ denote the learnable parameters for the neural network, and consider a loss (objective) function L. Then the optimal neural network is given by:*

$$\arg\min_{\theta\in\Theta} L(\mathbf{y}, g(\mathbf{X})), \quad \boldsymbol{\theta} = (\mathbf{W}^1, \dots \mathbf{W}^n, \mathbf{b}^1, \dots, \mathbf{b}^n), \Theta = \prod_{i=1}^{n} \{\mathbb{R}^{H_{i-1}\times H_i}\} \prod_{i=1}^{n} \mathbb{R}^{H_i} \tag{28}$$

Example 7 (MAE, RMSE)**.** *The expected Mean Absolute Error (MAE) corresponds to the absolute bias, or L^1 norm, and the expected Mean Squared Error (MSE) corresponds to the variance, or squared L^2 norm (Root Mean Squared Error (RMSE))*

$$MAE(\mathbf{y}, g(\mathbf{X};\boldsymbol{\theta})) = \mathbb{E}[\|\mathbf{y} - g(\mathbf{X};\theta)\|] \approx \frac{1}{N}\sum_{i=1}^{N} \|y_i - g(\mathbf{X})_i\| \tag{29}$$

$$MSE(\mathbf{y}, g(\mathbf{X};\boldsymbol{\theta})) = \mathbb{E}[\|\mathbf{y} - g(\mathbf{X};\theta)\|^2] \approx \arg\min_{\theta\in\Theta} \frac{1}{N}\sum_{i=1}^{N} \|y_i - g(\mathbf{X})_i\|^2 \tag{30}$$

In effect, determining the optimal neural network for a fixed architecture amounts to determining the optimal parameters $\boldsymbol{\theta}$, which amounts to solving a non-linear optimisation problem. (Definition 18. The 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.

In Example 7 we depict two common loss functions L for machine learning regressions; in particularly the Mean Absolute Error (MAE) and Mean Squared Error (MSE). These have the straightforward interpretation of being the absolute prediction errors (similar to the bias), and the squared prediction errors (similar to the variance). In general, any arbitrary loss function may be considered, but different losses lead to different results. For example, the MSE penalises large deviations more than the MAE. Also note that the input parameters $\mathbf{X}$ are no longer fixed in Example 7, but a $\mathbb{R}^{N_F}$ random variable over some probability space $(\mathcal{X}, \mathcal{F}^{\mathcal{X}}, \mathbb{P}^*)$. In effect, we want the minima of the $L^p \times \mathbb{P}^{\mathcal{X}}$ norm over the *entire* sample space for inputs $\mathbf{X}$. 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.

Algorithm 1 Mini-Batch Stochastic Gradient Descent

```
Initialise parameters θ_0 randomly via PRNG, t = 0
while t ≤ EPOCHS × ⌈N/BatchSize⌉ or NOT StoppingCriterion do
  for batch = ⌈1, ..., N/BatchSize⌉ do
    Compute loss L(y^batch, g(X^batch))
    Compute gradient w.r.t. loss ∇L via AAD
    t ← t + 1
    Set θ_{t+1} ← θ_t − η_t ∇_θ L
  end for
  if StoppingCriterion then Break
  end if
end while
```

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* (Algorithm 1). In short, we perform iterative updating to

the neural network parameters $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \lambda \nabla_{\boldsymbol{\theta}_t} L$ based on the direction of the loss. 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).

Weight Initialisation: Firstly, we randomly initialise the parameters $\boldsymbol{\theta}_0$. Typically, the weights are drawn from some random distribution, e.g. $\mathbf{W}^i_{j,k} \sim N(0, \sigma^i)$. The parameter 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 $\nabla_{\boldsymbol{\theta}}$ of the loss with respect to the parameters $\boldsymbol{\theta}$ are computed via AAD in the neural network framework. 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 $\nabla_\theta.L$

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 the gradient estimate $\nabla_\theta L$. The Adam optimizer is one method [43] that incorporates previous gradient estimates, and has been shown empirically to outperform SGD in many settings.

Overfitting and Effiicent Training

From 7, losses represent estimates of the error across the sample space $\mathcal{X}$. The neural network may potentially fit the values of $\mathbf{X}$ well, but have high errors for unseen $\mathbf{X}^{new} \in \mathcal{X}$. This is referred to as *overfitting*. We consider several methods to address overfitting below. In practice, several neural network techniques have been found empirically to improve training speed and generalisation, some of which are outlined in [45].

- **Early Stopping**: Early Stopping terminates training when the generalisation error on $\mathcal{X}$ no longer decreases. Partition the dataset $(\mathbf{X}, \mathbf{y})$ into $\mathbf{X}^{train}, \mathbf{y}^{train}, \mathbf{X}^{val}, \mathbf{y}^{val}$, or sample another dataset from some (new) parameter space $(\mathcal{X}^{val}, \mathbf{y}^{val})$. Withold $\mathbf{X}^{val}$ from training (i.e. use in gradient descent), but on every epoch, evaluate the performance of g on $\mathbf{X}^{val}, \mathbf{y}^{val}$. Cease training when the performance on $(\mathbf{X}^{val}, \mathbf{y}^{val})$ no longer improves. In practice, we need only tune the initial learning rate λ_t and batch size BatchSize and set a large number of epochs, and let EarlyStopping terminate training. This also means that the time budget EPOCHS is only upper bound, as training may converge or fail before EPOCHS is reached.

- **Batch Normalisation**: BatchNormalisation may enable faster training. The outputs of each layer are normalised element-wise by batch $(\mathbf{Z} \ominus \mu_i) / \oslash \sigma_i$, which ensures that the gradient updates have a consistent scale and are more stable, which may encourage more efficient training / reduce overfitting.

- **Dropout**: [60] 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, such that $\mathbf{Z}' \leftarrow \mathbf{Z} \otimes \mathbf{R}\frac{1}{p}, R_{ij} \sim \text{Bernoulli}(p)$. We can think of this as evaluating a sub-network on every epoch, with the inntended effect of preventing 'over-dependence' on a particular basis function.

- **Weight Regularisation**: We set a penalty on the weights $\mathbf{W}^i$, which may introduce sparsity. This amounts to modifying the loss function, for example: $L + \lambda \sum_{i=1}^{n} \|\mathbf{W}^i\|_2$.

Special Architectures

Another consideration is whether to use special neural network architectures in place of feed-forward neural networks 16. We describe several hidden layer, or 'block architectures', that could be potentially used.

Example 8 (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.*

$$\mathbf{Z}^1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{1}(\mathbf{b}^1)^\top) \tag{31}$$
$$\mathbf{Z}^2 = g_2(\mathbf{X}\mathbf{W}^2\mathbf{1}(\mathbf{b}^2)^\top) \tag{32}$$
$$\mathbf{Z}^3 = \mathbf{Z}_1 \otimes \mathbf{Z}_2 \qquad \text{(gated block)}$$

The Gated Unit can facilitate the learning of multiplicative interactions between the basis functions (or 'latent factors'). [65] uses gated units to construct a neural network that has the requisite monotonicity and convexity constraints with respect to strike and moneyness.

Example 9 (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.*

$$\mathbf{Z}_1 = g_1(\mathbf{X}\mathbf{W}^1 + \mathbf{1}(\mathbf{b}^1)^\top) \tag{33}$$
$$\mathbf{Z}_2 = g_2(\mathbf{Z}_1\mathbf{W}^2 + \mathbf{1}(\mathbf{b}^2)^\top) + \mathbf{Z}_1 \qquad \text{(residual block)}$$

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:

$$\mathbf{Z}^3 = \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top \tag{34}$$

$$L(\mathbf{y}, g(\mathbf{X})) = \|(\mathbf{y} - \mathbf{Z}^2\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top)\|^2 \tag{35}$$

$$= \|\mathbf{y} - \left((g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{1}(\mathbf{b}^2)^\top) + \mathbf{Z}^1)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top\right)\|^2 \tag{36}$$

$$= \|\mathbf{y} - (g_2(\mathbf{Z}^1\mathbf{W}^2 + \mathbf{y} - \mathbf{1}(\mathbf{b}^\top)\mathbf{W}^3 + \mathbf{1}(\mathbf{b}^3)^\top] - (\mathbf{Z}^1\mathbf{W}^3)\|^2 \tag{37}$$

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 at the cost of no extra parameters, (but slightly more complicated inference), as the upper layers are also connected to the prediction, which may alleviate the vanishing gradients issue

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 [31] 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.

Definition 19 (AutoML)**.** *Consider some space of possible neural network architectures $\mathcal{N}$, for example, in terms of the number of hidden layers or hidden units. Then consider some finite subset of $\mathcal{N}_{sub} \subset \mathcal{N}$ and determine:*

$$\arg \min_{g_i \in \mathcal{N}_{sub}} \min_{\theta \in \Theta_{g_i}} L \tag{38}$$

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).

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.[52] describes 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 10 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.

Example 10 (Example Grid Search)**.**

$$H_i \times (N_H - 1) \times g_i \in \{32, 64, 128\} \times \{32, 64, 128\} \times \{ELU, Swish\} \tag{39}$$

For a larger search space, for example, with continuous parameters, and 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 [27] considered in their paper for using neural networks to solve parametric pricing PDEs.

Summary

- Neural Networks are similar to basis function regressions, but the basis functions are flexible and determined from data.
- The Mini-batch SGD enables training at scale. We can fix a time budget proportional to the maximum number of iterations to consider $O(\lceil \frac{N}{\text{BatchSize}} \rceil \times$ EPOCHS), and the memory usage is proportional to $O(BatchSize)$ as opposed to the entire dataset $O(N)$. Furthermore, training can be parallelised, or further speed up using dedicated hardware such as TPUs or GPUs.
- Inference time, which includes time to compute both predictions and sensitivities, can be relatively fast after training.
- The disadvantage is that there is no guarantee of convergence in even the first order (pricing). We cannot explicitly control the trade-off between training time and error, only indirectly control the training time - error tradeoff by expending computational effort on Neural architecture search.

Finally, there is no guarantee that an arbitrary neural network construction has the desired properties (e.g. no-arbitrage) of derivatives pricing functions. In the next section, we review the literature towards apply neural networks for derivatives pricing.

4 Neural Networks for Derivatives Pricing

In the previous sections we considered the problem formulation of approximating derivatives prices and sensitivities, as well as 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. [55], 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 constructing 'hand-crafted' neural networks with prior knowledge, through the choice of an appropriate architecture or loss functions, which we examine in this section.

We consider two potential settings for pricing function approximations. For example: [49] [34] [20] 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 [35] 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).

Remark. *In this section, we mainly consider the neural network construction aspects. However, the entire machine learning pipeline (depicted in Figure 1), may be of interest, in particular the aspects of dataset construction A before the model is trained, and interpretability and monitoring after the model has been trained A.*

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 [38] [12], which arises from the Universal Approximation Theorem 17. This excludes the commonly used ReLU if continuous first-order gradients are needed, and the ELU function in Table 19. The softplus, swish, and gelu activation functions may be appropriate given that they are C^∞, although the latter two are non-monotonic.

[12] [65] define a specific neural network architecture to satisfy some no-arbitrge constraints. *softplus* activation $\log(1 + e^x)$ is used for the moneyness, and sigmoid σ 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 and convex in moneyness (thus strike), and monotonic in time-to-maturity. We can extend this to incorporate other inputs if we choose a non-negative activation function:

Example 11 (Gated Network, Hard Constraints)**.**

$$g(m,\tau,\mathbf{X}) = \sum_{i=1}^{H_1} w_i \sigma(w_{\tau,i}\tau)\log(1+\exp(w_{m,i}m))\log(1+\exp(X_i\mathbf{W}_i)), w_{\tau,i}>0, w_i>0 \tag{40}$$

Loss Functions for Derivatives Pricing and Risks

We now consider modifying the loss functions for the supervised learning task, and considering either Residual Networks or Feed-Forward Neural Networks as opposed to a specific architecture. Let L denote some loss function, for example the MSE or MAE 7

Example 12 (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 13 (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 [49] 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.

Example 14 (Control Variate)**.**

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

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 [22], the neural network is used to correct the errors of the SABR approximation of [29]. In this context, the neural network is used to 'correct' the error of another pricing approximation. 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 19 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 15 (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 [6], 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], the authors 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 16 (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. [38] 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 17 (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 [35] 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 $\frac{\partial y}{\partial \mathbf{X}}$[35] 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 does not matter in the offline setting, but may be significant in the on-the-fly setting.

Example 18 (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. [53] considered incorporating PDE terms to a neural network loss function *Physics Informed Neural Network*, and [59] formulated a Neural PDE approach to solve high-dimensional PDE problems for pricing called the *Deep Galerkin Method*. Here $\mathcal{L}$ denotes the corresponding PDE operator for the volatility model. [64] 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), $\mathcal{L}g = 0$, then there is no dynamic arbitrage. [27] considers an extension to [59] 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 17. In [59], the authors 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 approach cannot be used when time-to-maturity τ is not an input.

Example 19 (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 a multi-objective optimisation problem, and we must determine the optimal weightings λ for each objective. [35] argues that λ_1 is not significant and sets $\lambda_1 = 1$. However, this could be determined by considering prior knowledge of the relative scales or through trial-and-error. [50] 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 incorporate prior knowledge of a derivative and market model into the neural network:

- Hard Constraints: Constrain the architecture, weights, hidden units of a neural network to match the properties of the payoff
- Soft Constraints: Penalise static arbitrage bound violations
- Control Variate: Use neural network to correct another output, e.g. SABR approximation.
- Differential: Incorporate knowledge of the differentials into the neural network, could lead to accurate gradients
- Neural PDE: Incorporate knowledge of the pricing PDE into the neural network, could lead to g being a solution of the PDE and hence no dynamic arbitrage

Each approach addresses some of the criteria of interest: no-arbitrage errors, gradients, matching the PDE, but could potentially lead to improvements in all errors. 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. In Equation 48 we could consider incorporating multiple losses. For our subsequent numerical experiments we consider each construction method individually, where $\lambda^{method} = 1$ if the method is the method to be evaluated, and 0 otherwise.

5 Numerical Experiments

We consider two settings: European Call in 1D Black-Scholes and a high-dimensional Basket Call in the Bachelier model. We conduct our numerical experiments in a similar workflow as in Figure 1. We construct a dataset, define the neural networks, and then evaluate prediction, gradient, PDE, no-arbitrage error (where available), as well as the model and time complexity. In all cases, we fix the architecture to explore the empirical impact, although this is not the correct 'true' setup in the Black-Scholes case.

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. For the basket example, we consider the notebook examples [1] for [35] as a reference.

Abbreviations:

- Feed-Forward Neural Network (FFN) 16
- Residual Network (ResNet): Network with residual blocks in hidden layers 9
- Gated: The Hard-constrained neural network with gated units
- Differential (DiffNet): Neural Network trained with first order differentials 17
- Neural PDE (NPDE): Neural Network trained with pricing PDE 18
- Polynomial Regression (PolyReg) 18
- Ensemble: Average of all neural network approaches 5
- Mean Absolute Error (MAE): L1 Norm, Absolute Bias 29
- Mean Squared Error (MSE): Square of L2 Norm, Variance 30
- Root Mean Squared Error (RMSE): L2 Norm, dispersion
- Maximum Absolute Error (Max): Maximum of Absolute Error, L^∞ norm
- LowerBound: Proportion of samples below the intrinsic lower bound in the dataset $(S_t - K)^+$
- Monotonicity: Proportion of samples with negative monotonicity in moneyness $\frac{\partial g}{\partial m} < 0, \frac{\partial g}{\partial s} < 0$

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

- TimeValue: Proportion of samples with negative monotonicity in time-to-maturity: $\frac{\partial g}{\partial \tau} < 0$, or negative monotonicity in time-scaled implied volatility $\frac{\partial g}{\partial(\sigma\sqrt{\tau}} < 0$
- Convex: Proportion of samples with negative convexity in moneyness in the dataset $\frac{\partial^2 g}{\partial m^2} < 0$
- TrainTime: Time to train the neural network
- InferenceTime: Unless otherwise stated, time to compute prices, first order sensitivites, and second order sensitivities with respect to moneyness.

European Call - Black-Scholes

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

Context: We consider the most simple case of European call pricing in 1D Black-Scholes. Although the neural network approximation is trivial given the availability of the Black-Scholes formula, this example provides a setting where we can examine the behaviour of the various construction methods, as well as train and test against the true price. The European call case is also particularly relevant, since if we can accurately approximate European calls, then 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.

Results at a glance:

- NN inference time is fast, 1.068×10^{-5}/sample for $65,536$ samples.
- Pricing accuracy is order 10^{-3}.
- DiffNet is the best neural network in pricing and gradient errors.
- NPDE is the best neural network in PDE errors
- All neural networks underperform PolyReg on a testing set with the same parameter range as training (interpolation). PolyReg with bump and re-value also much faster in this case.
- Visually, neural networks produce knot like errors. See Appendix A for the visualisation of predictions and errors.

Dataset: See Appendix A for further derivations. 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 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 potentially lead to better predictive performance overall.

	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 3: 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 FFN, DiffNet,ResNet, and 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; in effect we explore the tradeoff between depth with width.

	$N_H - 1$	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	-	$36 = \binom{7+2}{7}$	-

Other hyperparameters: $\text{Epochs} = 30, \text{BatchSize} = 256,$

Table 4: Hyperparameters for the Neural Network Architectures

Prediction Errors: The DiffNet is the best performing neural network approach in prediction errors, however, polynomial regression outperforms it in all the pricing errors in the testing set Test1. However, the DiffNet is the best performer in MAE and RMSE, 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 5: Black-Scholes Example, Prediction Errors

Gradient Errors: We obtain the same performance for the gradient errors as the prediction errors. Polynomial regresion outperforms the neural network approaches in the Test1 dataset, but DiffNet is the best performer in terms of both MAE and RMSE in the Test2 dataset. Interestingly, the NPDE approach attains the best Max-error in the Test2 dataset, which could suggest that solving the PDE leads to consistency throughout the entire sample space.

	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 6: Black-Scholes Example, Gradient Errors

PDE Errors: We additionally consider the AbsMean PDE error, defined as the absolute value of the averaged signed error in the dataset (whereas the MAE is the average of the absolute errors in the dataset). AbsMean represents the absolute value of the bias of the PDE. It may be potentially acceptable to have small dynamic arbitrage errors that 'cancel' out throughout the sample space. In this case, the Neural PDE approach outperforms the other approaches in both Test1 and Test2. As expected, the errors in the extrapolation dataset Test2 are higher in all cases.

	AbsMean, Test1	MAE, Test1	RMSE, Test1	Max, Test1
FFN	1.62e-01	8.29e-01	1.43	1.32e+01
ResNet	7.44e-02	3.13e-01	4.44e-01	2.26
Gated	2.28e-01	7.12e-01	9.43e-01	2.57
Differential	3.89e-02	1.24e-01	1.77e-01	7.22e-01
NPDE	**5.98e-04**	**5.29e-03**	**6.40e-03**	**2.65e-02**
Ensemble	3.58e-02	2.73e-01	3.71e-01	2.50
PolyReg	3.80e-02	9.19e-02	1.97e-01	1.32

Table 7: Black-Scholes Example, PDE Errors, Test1 (Interpolation)

	AbsMean, Test2	MAE, Test2	RMSE, Test2	Max, Test2
FFN	2.20	5.16	1.16e+01	9.62e+01
ResNet	1.29e-01	1.69	3.01	2.30e+01
Gated	5.38e-01	8.80e-01	1.28	4.29
Differential	1.11e-01	3.14e-01	4.93e-01	2.05
NPDE	**7.94e-02**	**8.54e-02**	**2.38e-01**	**1.40**
Ensemble	2.68e-01	1.06	2.58	2.28e+01
PolyReg	8.68e-01	1.97	4.47	3.57e+01

Table 8: Black-Scholes Example, PDE Errors, Test2 (Extrapolation)

No-Arbitrage Errors: 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 11, 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 in Test1, 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, Test1	d/dm, Test1	$d/d\sigma\sqrt{\tau}$, Test1	d^2/dm^2, Test1
FFN	1.15e-01	9.28e-02	2.55e-01	4.83e-02
ResNet	1.37e-01	3.65e-02	1.41e-01	1.11e-01
Gated	1.35e-01	**0**	**0**	**0**
Differential	1.96e-01	7.52e-02	8.58e-02	6.99e-03
NPDE	2.70e-01	1.60e-01	**0**	**0**
Ensemble	1.77e-01	9.10e-02	5.66e-02	, 2.27e-03
PolyReg	**7.24e-02**	4.93e-02	6.66e-02	4.83e-02

Table 9: Black-Scholes Example, No Arbitrage Errors, Test1 (Interpolation)

	LowerBound, Test2	d/dm, Test2	$d/d\sigma\sqrt{\tau}$, Test2	d^2/dm^2, Test2
FFN	2.25e-01	1.47e-01	3.77e-01	1.39e-01
ResNet	2.81e-01	1.04e-01	2.71e-01	1.71e-01
Gated	2.50e-01	**0**	**0**	**0**
Differential	3.75e-01	1.27e-01	3.55e-01	6.94e-02
NPDE	**1.80e-01**	2.72e-01	1.71e-03	**0**
Ensemble	2.74e-01	1.66e-01	2.29e-01	6.59e-03
PolyReg	2.03e-01	1.22e-01	4.16e-01	1.58e-01

Table 10: Black-Scholes Example, No Arbitrage Errors, Test2 (Extrapolation)

Model Complexity: See Appendix Figure 2 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 significantly fast inference time - **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.

Remark. *Training time does not necessarily matter in the setting of producing a clone for a pricing function, although ideally we would prefer it to be shorter. However, a shorter training time 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.*

	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 11: Black Scholes Example, Model Complexity

Remark. *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.*

Basket Option - Arithmetic Brownian Motion

Context: We consider the first example from [35], a standard European call on a high-dimensional 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 the Longstaff-Schwartz formulation. We benchmark the various neural network constructions against polynomial regression, and run a Monte Carlo on each sample path. We consider a similar setup as the first code example from [35]. Time-to-maturity $\tau = T - t$, strike K, and the covariance $\mathbf{L}\mathbf{L}^\top$ are fixed.

Results at a Glance:

- DiffNet is the best performing neural network in terms of all errors, and in this case outperforms polynomial regression.
- Polynomial Regression and the other neural network approaches results in predictions that are close to random noise. See A for the visualisations
- This example may still be too simplistic, as a MC simulation on each path + AAD outperforms the neural network in all errors by approximately $\times 10$, whereas the neural network has a speedup (training + inference) of around $\times 6$

Dataset: For details on the derivations, see Appendix A. We generate samples $i = 1, \ldots, N_{\text{samples}} = 10,000$ samples of $\mathbf{S_{i,0}}, \mathbf{W}_{i,0} \in \mathbb{R}^d$, simulating $\mathbf{S_{i,T}} = \mathbf{S}_{i,t} + \mathbf{L}\mathbf{W_{i,T}}$ exactly from 76. 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 without further processing.

In our setup, we consider a fixed parameter space Table 12. We simulate a random covariance matrix by taking a Cholesky decomposition of a single posiive-definite sample of matrix of standard normal random variables: $\mathbf{L}\mathbf{L}^\top = 0.2\mathbf{Z}^\top\mathbf{Z}, Z_{i,j} \sim N(0,1)$. Given we do not vary time-to-maturity in this case, we cannot use the Neural PDE method.

d (No. Assets)	τ (Time-to-Maturity)	K (Strike)	$\mathbf{w}$ (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 12: Fixed Parameters for Basket Example

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

Table 13: Sample Parameter Space for Basket Example

Models: We first consider a polynomial basis regression (PolyReg) as a benchmark method, with a low degree (2) such that the number of basis functions is $5151 = \binom{10+2}{7}$. We use the same construction for the FFN, DiffNet,ResNet approaches: a rectangular neural networks with $N_H - 1 = 4$ hidden layers, and $H_i = 100$ hidden units. In place of the Gated Network, we consider a Control Variate model 43, which learns the residual between the call price and the intrinsic value of the basket.

	$N_H - 1$	H_i	g_i
Feed-Forward Network (FFN)	4	100	swish
Residual Network (ResNet)	4	100	swish
Differential (DiffNet)	4	100	swish
Neural PDE (NPDE)	4	100	swish
ControlVar	4	100	swish, rbf (final)
PolyReg	-	$5151 = \binom{100+2}{100}$	-

Other hyperparameters: $\text{Epochs} = 100, \text{BatchSize} = 30,$

Table 14: Hyperparameters for the Neural Network Architectures

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, and model complexity.

Prediction Errors: The best performing neural network approach in this case is this **DiffNet** approach, as depicted in Table 15. The DiffNet outperforms the other neural network constructions, and unlike 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 running a Monte Carlo on each sample path.

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

Table 15: Bachelier Basket Example - Prediction Errors

Gradient Errors: Table 16 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 is likely 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
MAE	2.275×10^{-3}	2.820×10^{-3}	$\mathbf{6.036 \times 10^{-4}}$	2.257×10^{-3}	1.346×10^{-1}	$\mathbf{3.017 \times 10^{-5}}$
RMSE	2.883×10^{-3}	3.531×10^{-3}	$\mathbf{7.623 \times 10^{-4}}$	2.816×10^{-3}	1.724×10^{-1}	$\mathbf{3.437 \times 10^{-5}}$
Max	1.292×10^{-2}	1.481×10^{-2}	$\mathbf{4.639 \times 10^{-3}}$	1.130×10^{-2}	1.027	$\mathbf{6.816 \times 10^{-5}}$

Table 16: Bachelier Basket Example - Gradient Errors

No-arbitrge Errors: Table 17 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
Pred, Lower	3.109×10^{-1}	3.182×10^{-1}	$\mathbf{6.280 \times 10^{-3}}$	2.649×10^{-1}	3.216×10^{-1}	**0**
Grad, Lower	5.622×10^{-2}	1.246×10^{-1}	$\mathbf{4.000 \times 10^{-5}}$	5.514×10^{-2}	4.860×10^{-1}	**0**

Table 17: Bachelier Basket Example - No-Arbitrage Violations

Model Complexity: Table 18 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. However, we are unable to explicitly control the time spent training when using EarlyStopping.

	CV	FFN	DiffNet	ResNet	PolyReg	MC
InfTime	7.288 $\times$ 10^{-2}	**6.337** $\times\mathbf{10^{-2}}$	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 18: Bachelier Basket Example - Time Complexity

Remark. *As per the hypothesis of [35], 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 Arithmetic 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, in this case, there is a neural network construction that outperforms polynomial regression. At least for our particular choice of sample seed and the given neural network architecture, all neural network approaches underperform a MC estimate on each sample path 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.*

European Call - Rough Bergomi

We also briefly examine a work-in-progress Appendix A. As of the submission date, the results are incomplete, but preliminary results suggest similar behaviour as the Black-Scholes setting. The neural network approaches underperform against basis regression, although in this case, the dimensionlity is still low (5).

6 Conclusion

In our experiments, we have seen that neural networks are able to achieve fast inference speeds in terms of obtaining prices and gradients. However, a significant problem is the inability to explicitly control the convergence rate, which could be an issue in the on-the-fly setting.

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.

In terms of the various construction methods, we saw that incorporating differential and PDE terms into the loss function can lead to increased performance towards the gradient errors and PDE error. On the other hand, a pre-specified architecture, the hard constrained approach or Gated Network prevented some of the no-arbitrage conditions, but came at the cost of much lower accuracy or expressivity.

For further improvements, we could consider improving the whole workflow as in Figure 1. In terms of *dataset generation*, 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 applications, we could consider further investigation into more realistic payoffs, for example high-dimensional callable products such as Bermudan swaptions as in [44]. For further comparisons against *other methods*, we could compare against the Cheybschev / Tensor based approaches of [2], as no convenient `Python` implementation exists yet, or investigate efficient implementations of Gaussian processes (see Appendix A).

Alternatively, another approach could be to experiment with using the neural network itself as a volatility model (Neural SDEs, GANs Appendix A), or to compare direct pricing with the hedging via replication (Deep Hedging) [6], or the FBSDE approach of [63]

Finally, we explore extensions of neural networks, in particular quantum neural networks and their application for derivatives modelling [57].

A Appendix

Pricing for Non-European 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 20 (Path-Dependent Payoff)**.** *A path-dependent payoff depends on the entire history of* $\mathbf{S_t}$ *over the start and maturity* $\in [0, T]$

$$f(\tau, (\mathbf{S}_u)_{u\in[0,t]}, \mathbf{X}) = \mathbb{E}^{\mathbb{Q}}\left[h_2((\mathbf{S_u})_{u\in[0,T]})|(\mathbf{S}_u)_{u\in[0,t]}\mathbf{X}\right] \tag{49}$$

Example 20 (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 given the original $\mathbf{S}_t$, as it may depend on the history $(S_u)_{u\in[0,t]}$. 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 $M_t = \max_{u\in[0,t]}\{S_u\}$. Thus we can think of some path-dependent options as a European option on a non-traded process. Otherwise, we could consider approximating the true pricing function.

Definition 21 (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* $\mathcal{T}$*.*

$$f(\mathbf{X}) = \sup_{T^*\in\mathcal{T}} \mathbb{E}[h(\mathbf{S}_{T^*})|\mathbf{X}] \tag{50}$$

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.

Definition 22 (Least Squares Monte Carlo)**.** *Let* Z_i *denote* $i = 1, \ldots, N_B$ *basis functions,* w_i *he weights, and* b *the bias term or intercept. Suppose* $\mathbb{E}[g(\mathbf{S}_{t_i})^2] < \infty$

the payoff is L^2 integrable. Let $\mathbf{X} \in \mathbb{R}^{N_F}$ be a matrix denoting N sample parametrs, and $\mathbf{y} \in \mathbb{R}$ be a vector containing the corresponding sample payoffs $h(\mathbf{S}_{j,T})|\mathbf{X}$

$$g_{t_{N_T-1}}(\mathbf{S}_{t_{i-1}}) \approx \mathbb{E}^{\mathbb{Q}}[h(\mathbf{S}_T)|\mathbf{S}_{t_{N_T-1}}] \tag{51}$$

$$g_{t_i}(\mathbf{S}_{t_i}) = \max\{h(\mathbf{S}_{t_i}), \mathbb{E}^{\mathbb{Q}}[g_{t_{i+1}}(\mathbf{S}_{t_{i-1}})|\mathbf{S}_{t_i}] \tag{52}$$

$$g_{t_i}(\mathbf{S}_{t_i}) = \sum_{i=1}^{N_B} w_i Z_i(\mathbf{S}_{t_i}) + b\mathbf{1} = \mathbf{Z}(\mathbf{S}_{t_i})\mathbf{W} + b\mathbf{1}, \mathbf{W} \in \mathbb{R}^{N_B}\mathbf{S}_{t_i} \in \mathbb{R}^N \tag{53}$$

$$\arg\min_{\mathbf{W},b} \|\mathbb{E}^{\mathbb{Q}}[\left(\mathbb{E}^{\mathbb{Q}}[g_{t_{i+1}}(\mathbf{S}_{t_{i-1}})|\mathbf{S}_{t_i}] - g_{t_i}(\mathbf{S}_{t_i})\right)^2\|^2 \tag{54}$$

$$\mathbf{W} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{X}^\top\mathbf{y}] \tag{55}$$

No-arbitrage bounds for Exotic Payoffs: In general, the no static-arbitrage bounds may not be known. For non-European options, in some cases, we could replication arguments to obtain no-arbitrage lower and upper bounds, for example for Barriers (Example 21):

Example 21 (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 22):

Example 22 (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 :*

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

Activation Functions

Some common activations are depicted below. Different activations have different properties, and may influence training. For example, the Tanh and Sigmoid activations suffer from the vanishing gradients problem.

Machine Learning Workflow for Derivatives Modelling

A reference for designing machine learning systems is [37]. A potential workflow for applying machine learning and neural networks is depicted in Figure 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 19: Activation Functions and their Derivatives

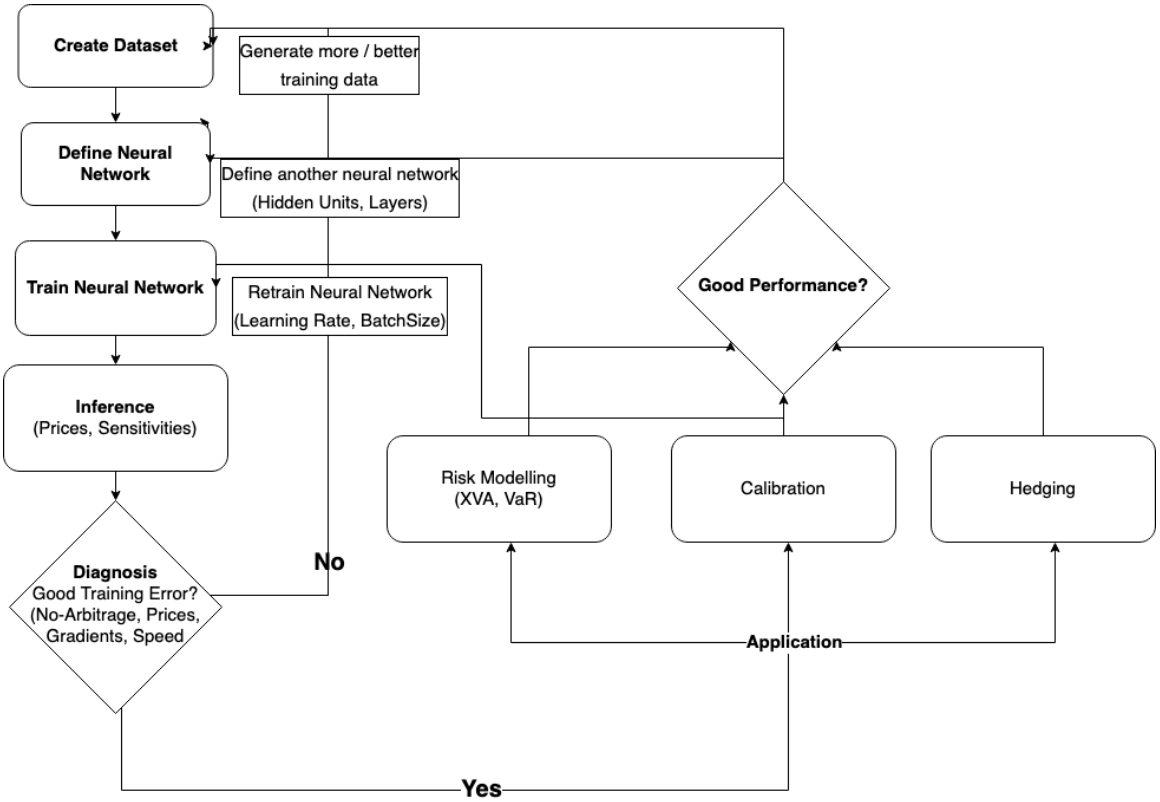


Figure 1: Example workflow for using neural networks in derivatives modelling

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 is 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, [49] let $\mathbf{y}$ 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 a 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 [34] and [49].

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 $\mathbf{y}$ using 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 [35] lets $\mathbf{y}$ be sample Monte Carlo payoffs (i.e. $N_{SAMPLES} = 1$, such that y_i is only an unbiased estimate of the true pricing function $\mathbb{E}[y_i] = f(\mathbf{X}_i)$. [20] 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. normal uniform). One such 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 [31] 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 [35] 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, [15] proposes a method to remove arbitrage in the call prices by projecting the call prices to a polytope of linear no-static-arbitrage constraints.

Interpreting and Monitoring Neural Networks

We briefly discuss some issues with neural networks during the downstream applications phase in Figure 1. The first issue is the 'black-box' nature and lack of inter-

pretability. 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 [14]. 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 [51], 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:

$$g(\mathbf{y}, \mathbf{X}^{contract}) = \mathbf{X}^{vol}, f(\mathbf{X}^{vol}, \mathbf{X}^{contract}) = \mathbf{y} \tag{56}$$

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 [24], we represent the dynamics of some stochastic process $\mathbf{S}_t$ with a neural network, for example consider:

$$\mathbf{S}_t = g(\mathbf{S}_t; \boldsymbol{\theta}) d\mathbf{W}_t \tag{57}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} [y_i(\mathbf{X}_i) - \mathbb{E}[h_i(\mathbf{S}_T; \mathbf{X})] \| \tag{58}$$

$$\arg\min_{\boldsymbol{\theta} \in \Theta} \| \sum_{i=1}^{N_C} \left[y_i(\mathbf{X}_i) - \sum_{j=1}^{N_B} h_i(\mathbf{S}_T; \mathbf{X}; W_j) \right] \| \tag{59}$$

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 context 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 are [8] [7]. GANs also have a connection with Neural SDEs, in that Neural SDEs can be considered to be an infinite-dimensional GAN [40]. 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 an 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: [16] [41] 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. Further, 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 [18], 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 led to highly competitive results on machine learning competitions such as *Kaggle*. Some papers, for example [19] [17], have considered their application towards pricing, given their predictive performance. 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; for applications that require pricing sensitivities, tree-based methods are not feasible given that the tree is nowhere differentiable.

Lookup Table: [46] 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.

Black-Scholes European Call

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:

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

We apply the change of variables to $M_t = \log(F_t/K)$ as suggested by [55]. As mentioned in Chapter 3, inputs and outputs that are standardised may lead to better training. The kog-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

$$\frac{\partial \sigma\sqrt{\tau}}{\partial \tau} = \frac{\sigma}{2\sqrt{\tau}} = \frac{\sigma^2}{2\sigma\sqrt{\tau}}, \quad \frac{\partial m}{F} = \frac{\partial m}{F} = \frac{1}{F} \tag{60}$$

$$\frac{\partial}{\partial F}(\frac{\partial g}{\partial m}\frac{\partial m}{\partial F}) = \frac{-1}{F^2}\frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}\frac{1}{F^2} \tag{61}$$

$$0 = -g_\tau - \frac{\sigma^2}{2}g_m + \frac{\sigma^2}{2}g_{mm}, g(\tau, m) = (\exp(m) - 1)^+ \tag{62}$$

$$0 = \frac{\sigma^2}{2}[-g_{\sigma\sqrt{\tau}}\frac{1}{\sigma\sqrt{\tau}} - g_m + g_{mm}], g(\tau, m) = (\exp(m) - 1)^+ \tag{63}$$

$$\tag{64}$$

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

$$0 = \frac{\partial g}{\partial \sigma\sqrt{\tau}}\frac{1}{\sigma\sqrt{\tau}} - \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}, \quad g(\tau, m) \tag{65}$$

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

$$\frac{\partial g}{\partial \tau} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}}\frac{\sigma}{2\sqrt{\tau}} > 0 \implies \frac{\partial g}{\partial \sigma\sqrt{\tau}} > 0 \tag{66}$$

$$\frac{\partial g}{\partial K} < 0 \implies \frac{\partial g}{\partial m}\frac{-1}{K} < 0 \implies \frac{\partial g}{\partial m} > 0 \tag{67}$$

$$\frac{\partial^2 g}{\partial K^2} > 0 \implies \frac{1}{K^2}\frac{\partial g}{\partial K} + \frac{\partial^2 g}{\partial m^2}\frac{1}{K^2} < 0 \implies \frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2} > 0 \tag{68}$$

$$(\exp(m) - 1)^+ \leq g(\sigma\sqrt{\tau}, m) \leq \exp(m) \tag{69}$$

$$\lim_{m\to-\infty} g(\sigma\sqrt{\tau}, m) = 0, \lim_{m\to\infty} = \exp(m) \tag{70}$$

$$\lim_{\sigma\sqrt{\tau}\to 0} g(\sigma\sqrt{\tau}, m) = (\exp(m) - 1)^+ \tag{71}$$

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 $-\frac{\partial g}{\partial m} + \frac{\partial^2 g}{\partial m^2}$ 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 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. We exploit this first-order positive homogeneity in the underlying and strike, as in [36], so we can eliminate one of F_t, K by letting $\lambda = \frac{1}{K}$ and fixing the other.

Although this does not necessarily extend to all volatility models (e.g. the homogenity holds for the Heston model, not for SABR 1), we can in general exploit dimensionality reduction techniques or model factors instead of the original problem, as in the case of MC/ PDEs.

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

$$\mathbb{E}^{\mathbb{Q}}[\frac{(F_t}{K} - 1)^+] = \frac{\mathbb{E}^{\mathbb{Q}}[(F_T - K)^+ | S_t, K, \sigma]}{K} \tag{72}$$
$$\lambda C(F_t, K, \sigma, \tau) = C(\lambda F_t, \lambda K, \sigma, \tau) \tag{73}$$

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

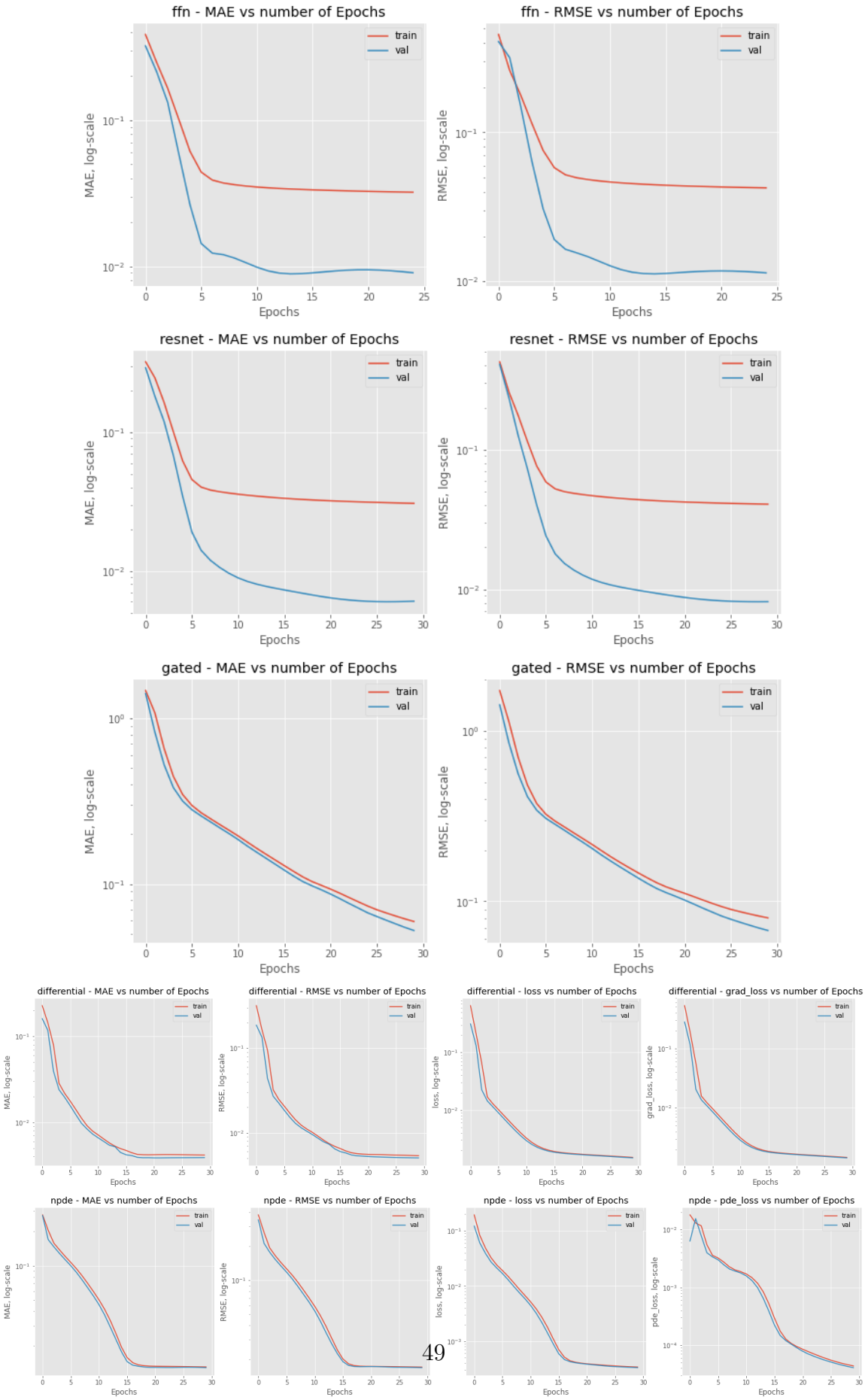


Figure 2: Black Scholes Examples, Training Curves

Figure 3: Black Scholes Example, Feed-Forward Neural Network

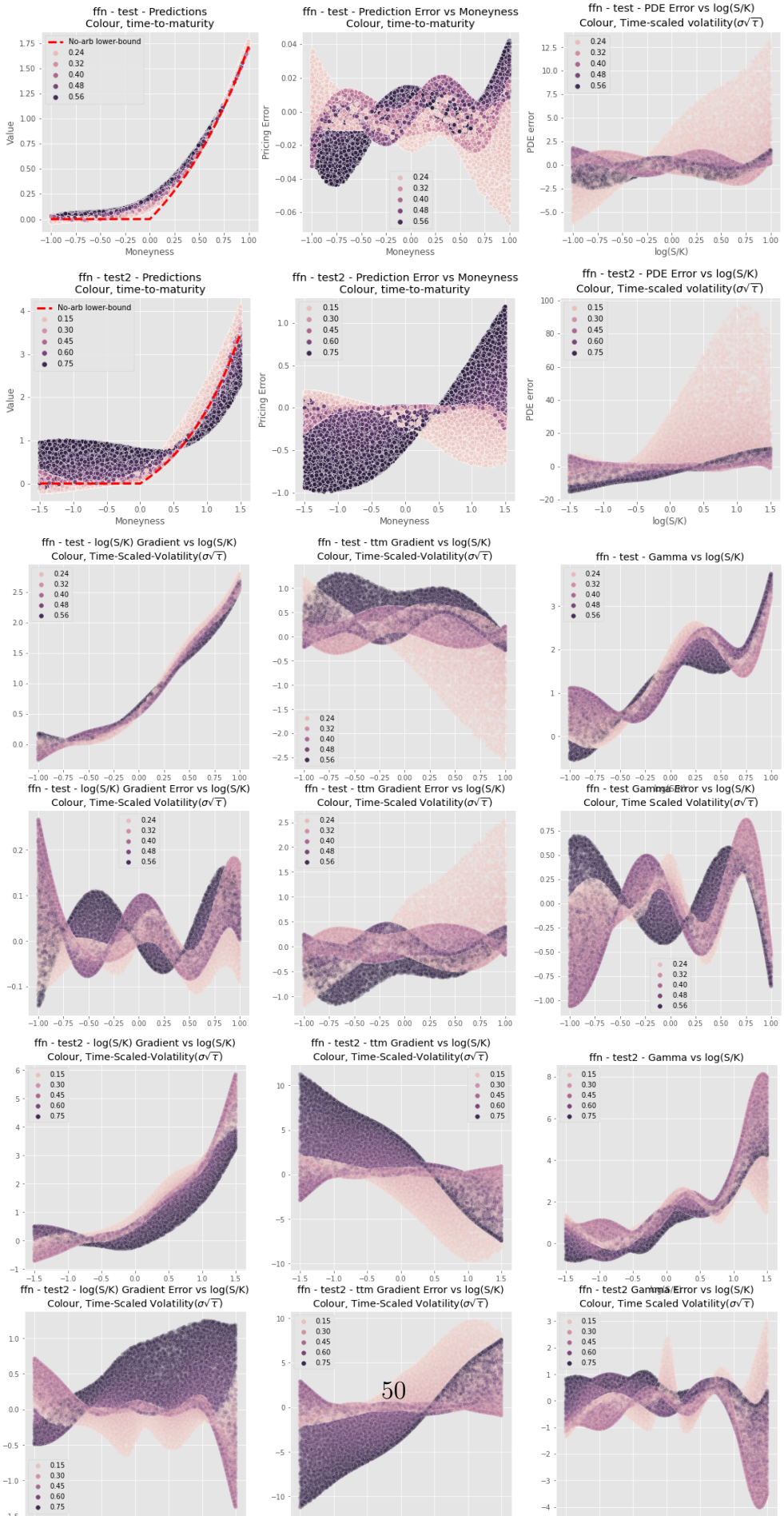


Figure 4: Black Scholes Example, Residual Neural Network

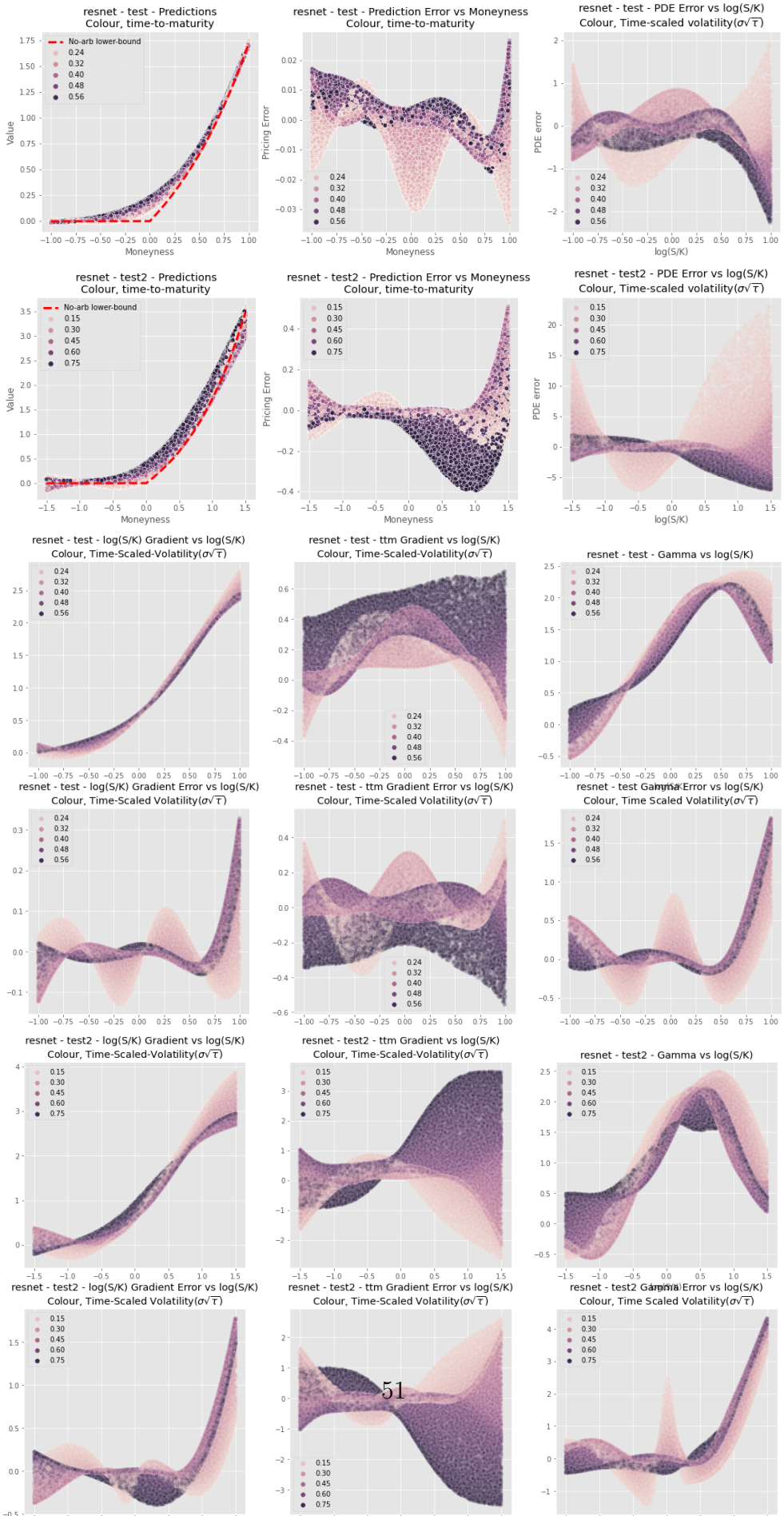


Figure 5: Black Scholes Example, Gated Neural Network

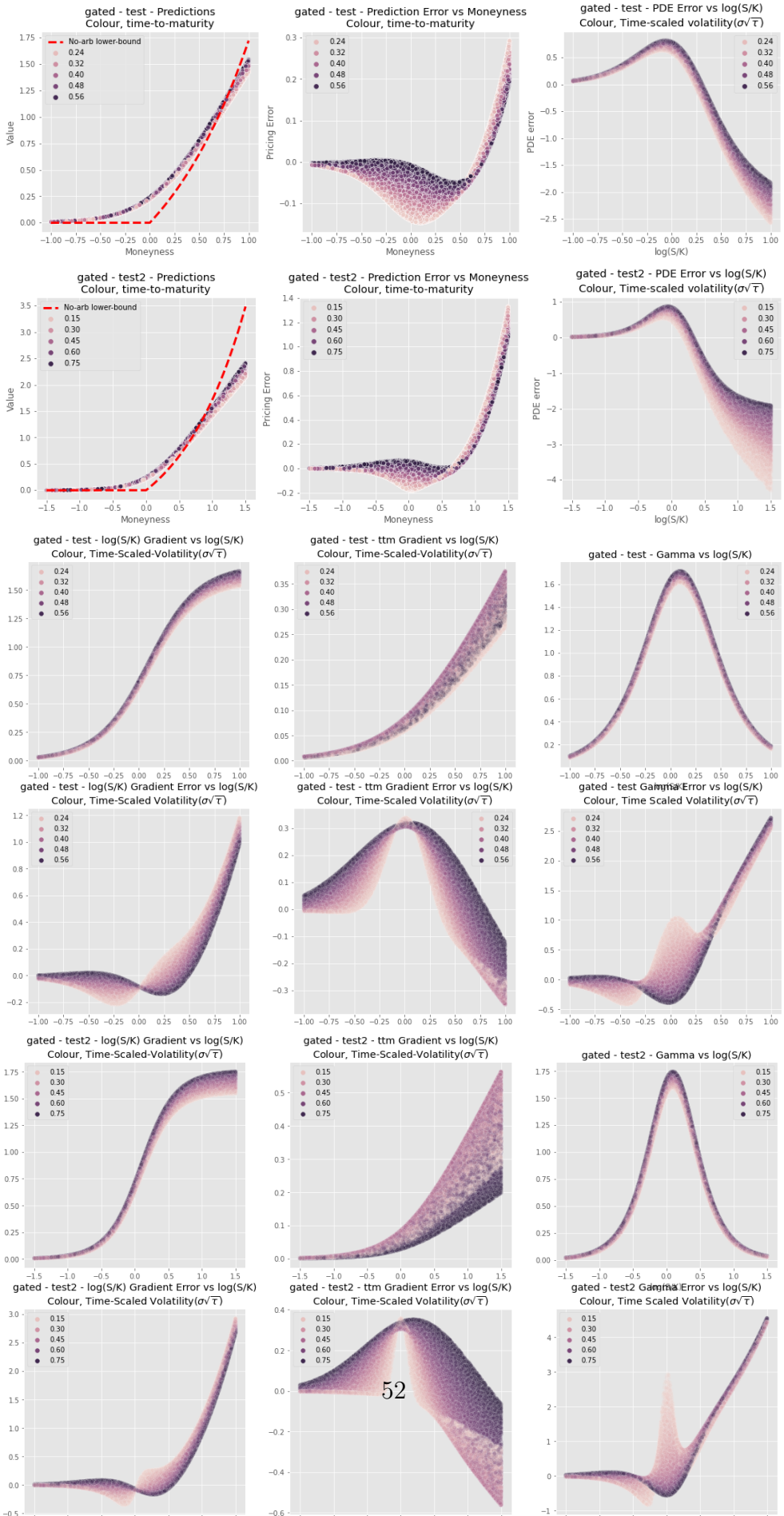


Figure 6: Black Scholes Example, Neural PDE

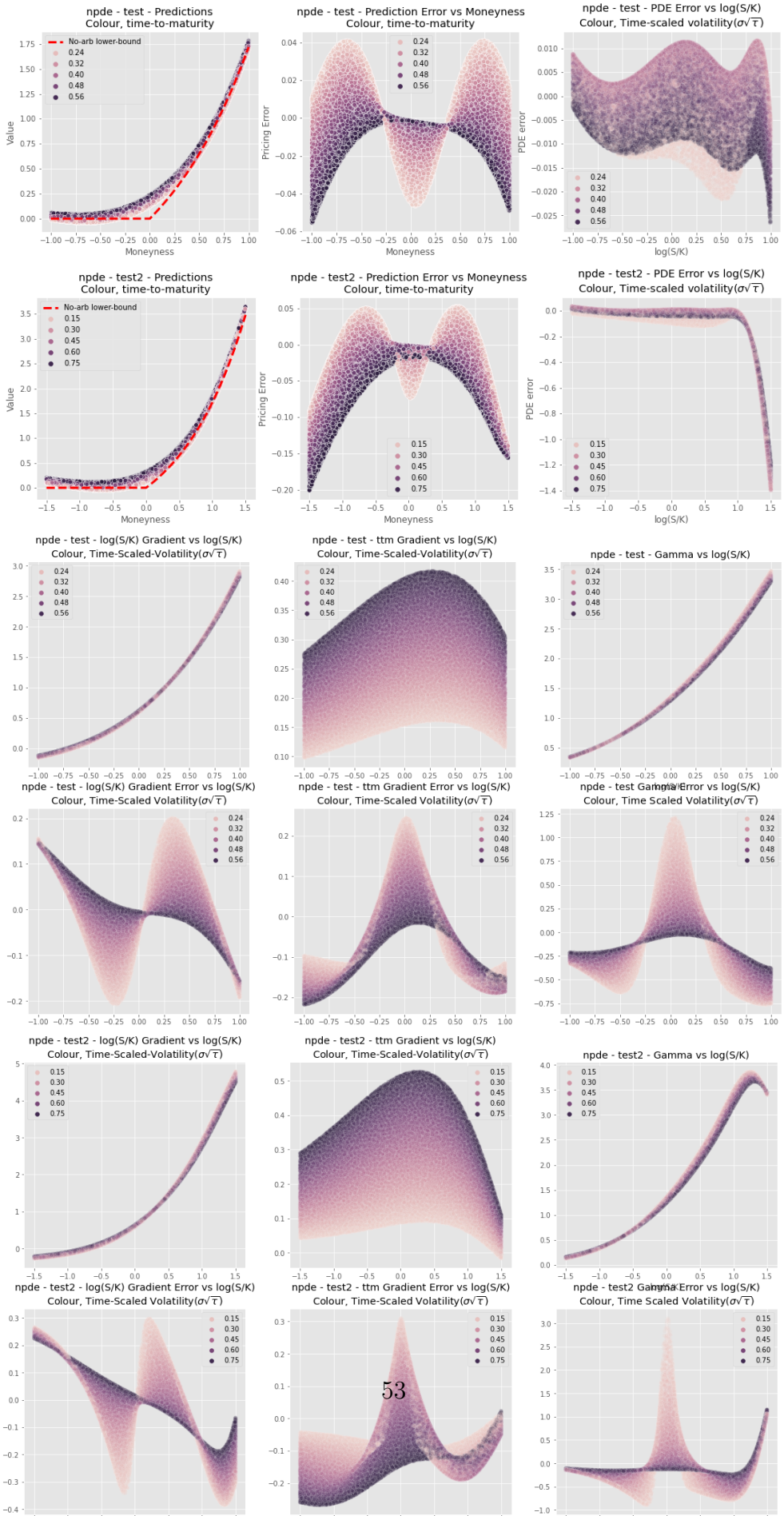


Figure 7: Black Scholes Example, Differential Neural Network

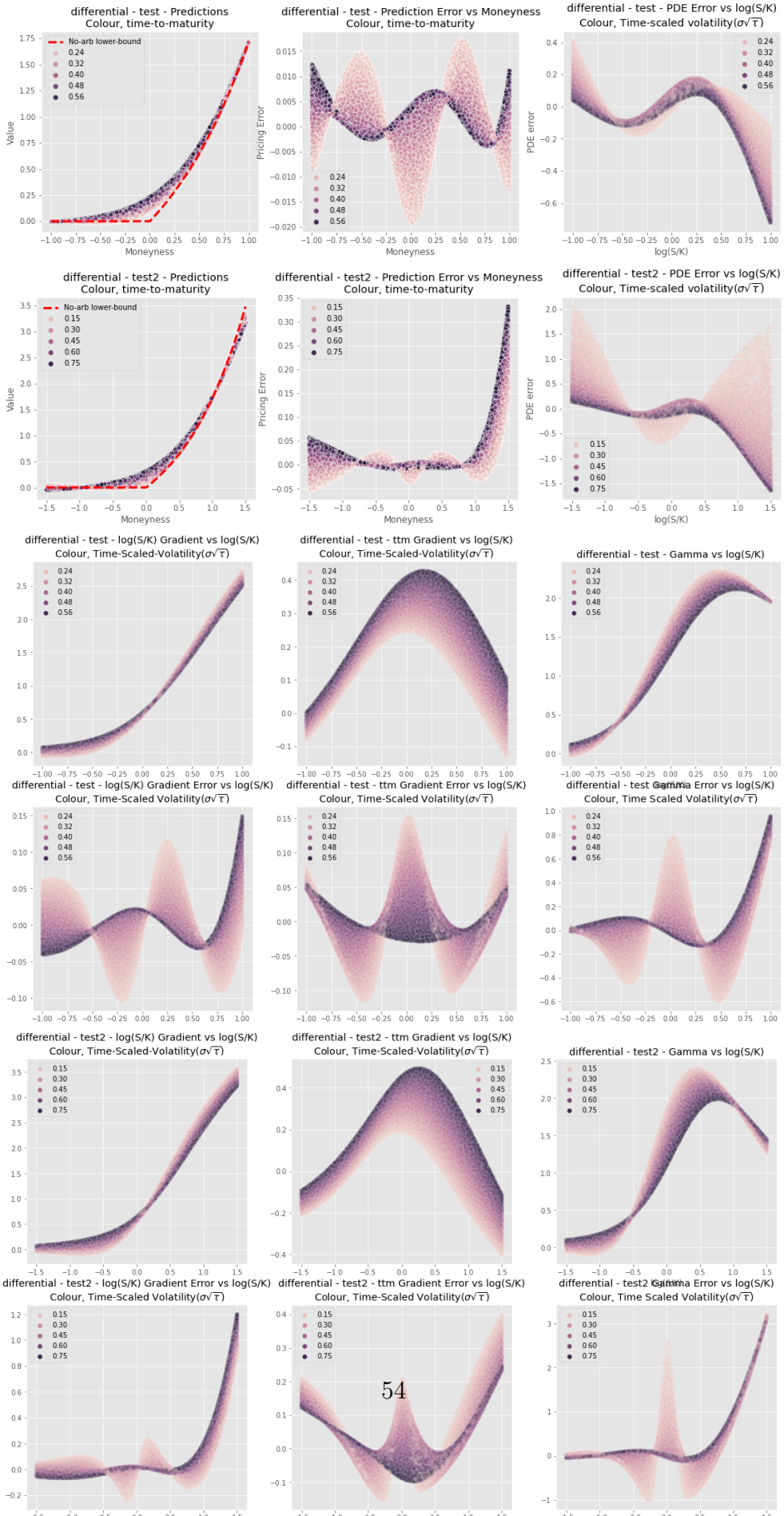


Figure 8: Black Scholes Example, Neural Network Ensemble

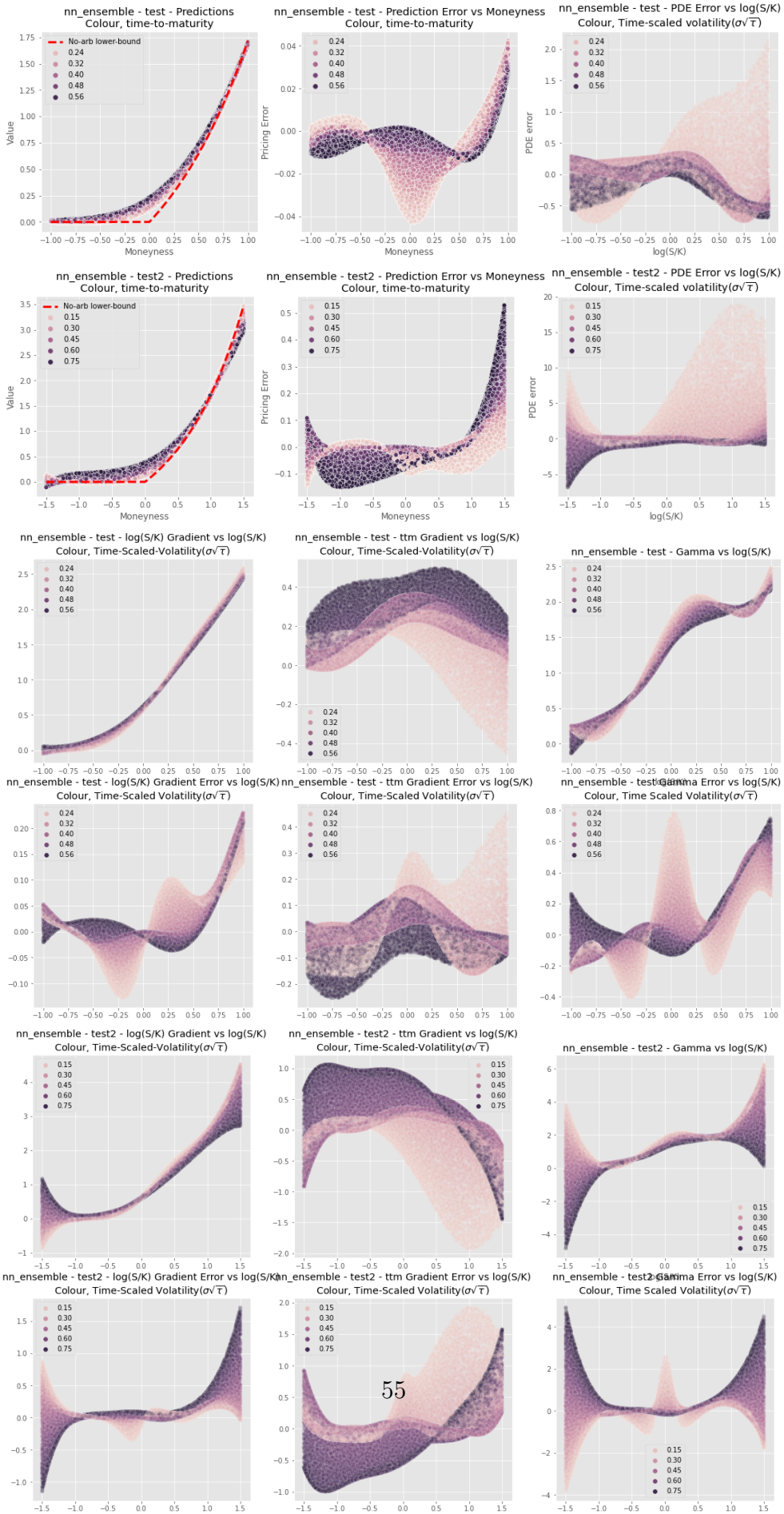
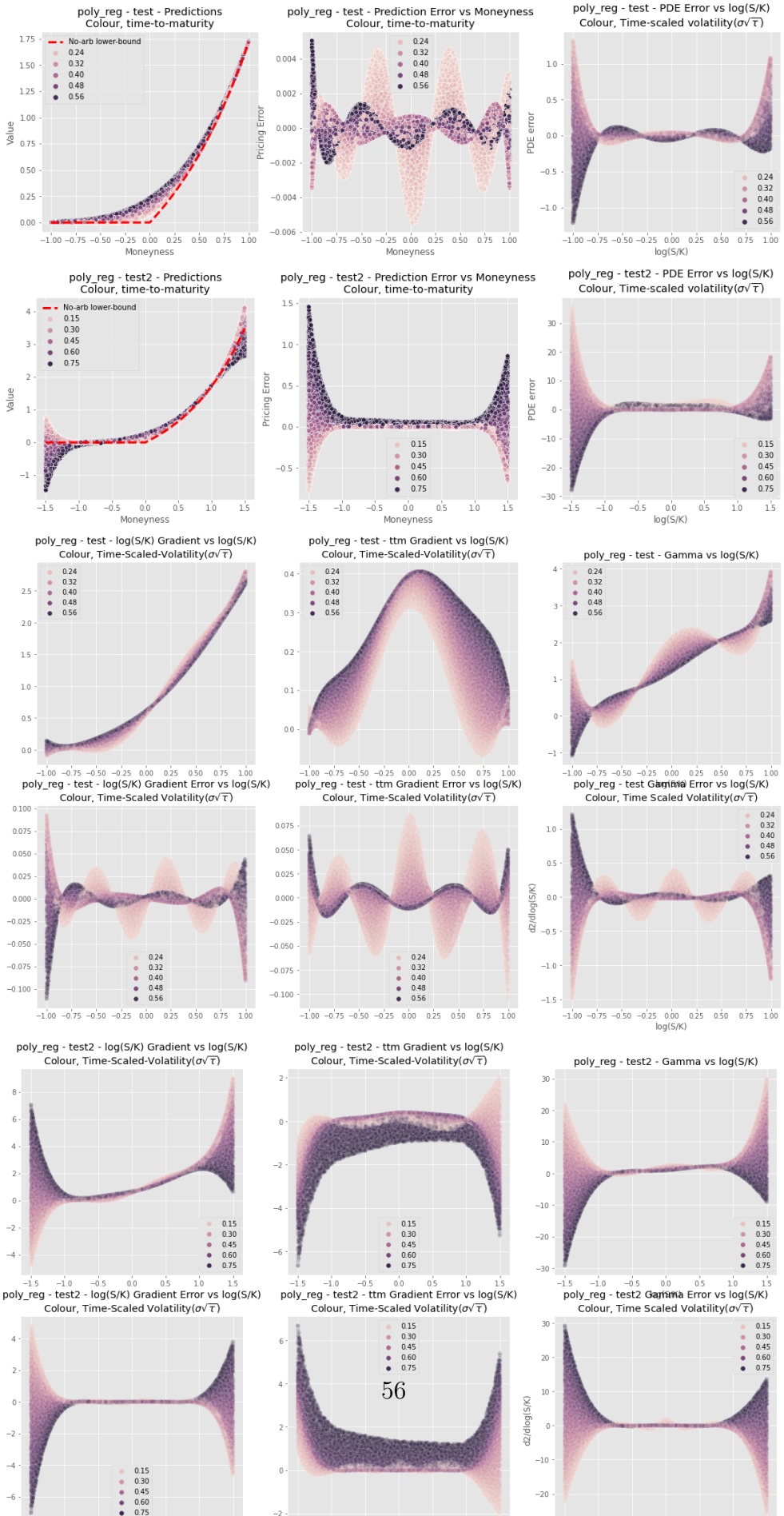


Figure 9: Black Scholes Example, Polynomial Regression



Basket Option - Arithmetic Brownian Motion

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:

$$d\mathbf{S}_t = \mathbf{L}d\mathbf{W}_t, \mathbf{S}_t = \mathbf{S}_0 + \mathbf{L}\mathbf{W}_t, \mathbf{S}_t \sim N(\mathbf{S}_0, \mathbf{L}\mathbf{L}^\top t) \tag{74}$$

$$d\mathbf{X}_t = \mathbf{w}^\top d\mathbf{S}_t = \sigma dW_t^\top, \mathbf{X}_t = \mathbf{w}^\top \mathbf{S}_t, \mathbf{X}_t \sim N(\mathbf{w}^\top \mathbf{S}_0, \mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w}) \tag{75}$$

$$h(\mathbf{w}, \mathbf{S}_t, K) = (\mathbf{w}^\top \mathbf{S}_T - K)^+ = (\mathbf{X}_T - K)^+ \tag{76}$$

$$f(\tau, \mathbf{S}_t, K, \mathbf{w}) = \mathbb{E}[(\mathbf{w}^\top \mathbf{S}_t - K)^+ | \mathbf{S}_t, \mathbf{w}, \tau, K] \tag{77}$$

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:

$$df(T - t, S_t; \ldots) = -\frac{\partial f}{\partial t}dt + \sum_{i=1}^{d} \frac{\partial f}{\partial S_i} dS_{i,t} + \sum_{i=1}^{d}\sum_{j=1}^{d} \frac{1}{2} \frac{\partial V}{\partial S_i S_j} d[S_i, S_j] \tag{78}$$

$$0 = -\frac{\partial f}{\partial \tau} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} \frac{\partial V}{\partial S_i S_j}, f(0, \mathbf{S}) = (\mathbf{w}^\top \mathbf{S} - K)^+ \tag{79}$$

$$df(T - t, X_t; \ldots) = -\frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dX_t + \frac{1}{2}d[X]_t \tag{80}$$

$$= \mathbf{w}^\top \mathbf{L} d\mathbf{W}_t + \frac{1}{2}\mathbf{w}^\top \mathbf{L}\mathbf{L}^\top \mathbf{w} dt \tag{81}$$

$$= -\frac{\partial f}{\partial \tau} + \frac{\sigma^2}{2}\frac{\partial f}{\partial x^2}, f(0, \mathbf{X}) = (\mathbf{X} - K)^+ \tag{82}$$

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:

$$
f(\mathbf{X}_0, \sigma, \tau, K) = \sigma\sqrt{\tau}[\phi(d_1) + d_1\Phi(d_1)], d_1 = \frac{\mathbf{X}_0 - K}{\sigma\sqrt{\tau}} \tag{83}
$$

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

$$
\frac{\partial f}{\partial x} = \sigma\sqrt{\tau}[-\frac{d_1}{\sigma\sqrt{\tau}}\phi(d_1) + \frac{1}{\sigma\sqrt{\tau}}\Phi(d_1) + \phi(d_1)\frac{d_1}{\sigma\sqrt{\tau}}] = \Phi(d_1) \tag{84}
$$

$$
\frac{\partial f}{\partial \tau} = \frac{f}{2\tau} + \sigma\sqrt{\tau}[\frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1^2}{2\sqrt{\tau}}\phi(d_1) - \frac{d_1}{2\sqrt{\tau}}\Phi(d_1)] = \frac{\sigma}{2\sqrt{\tau}}\phi(d_1) \tag{85}
$$

$$
\frac{\partial f}{\partial x^2} = \frac{1}{\sigma\sqrt{\tau}}\phi(d_1) \tag{86}
$$

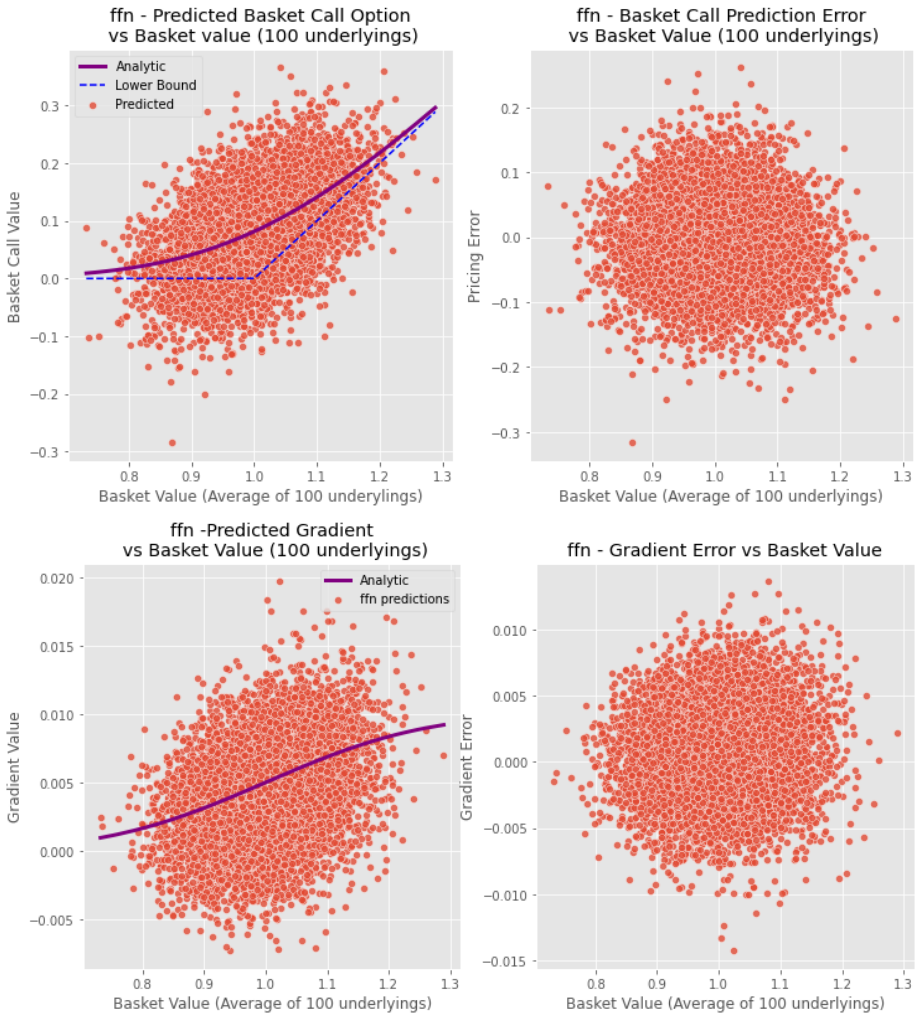


Figure 10: Feed Forward Network

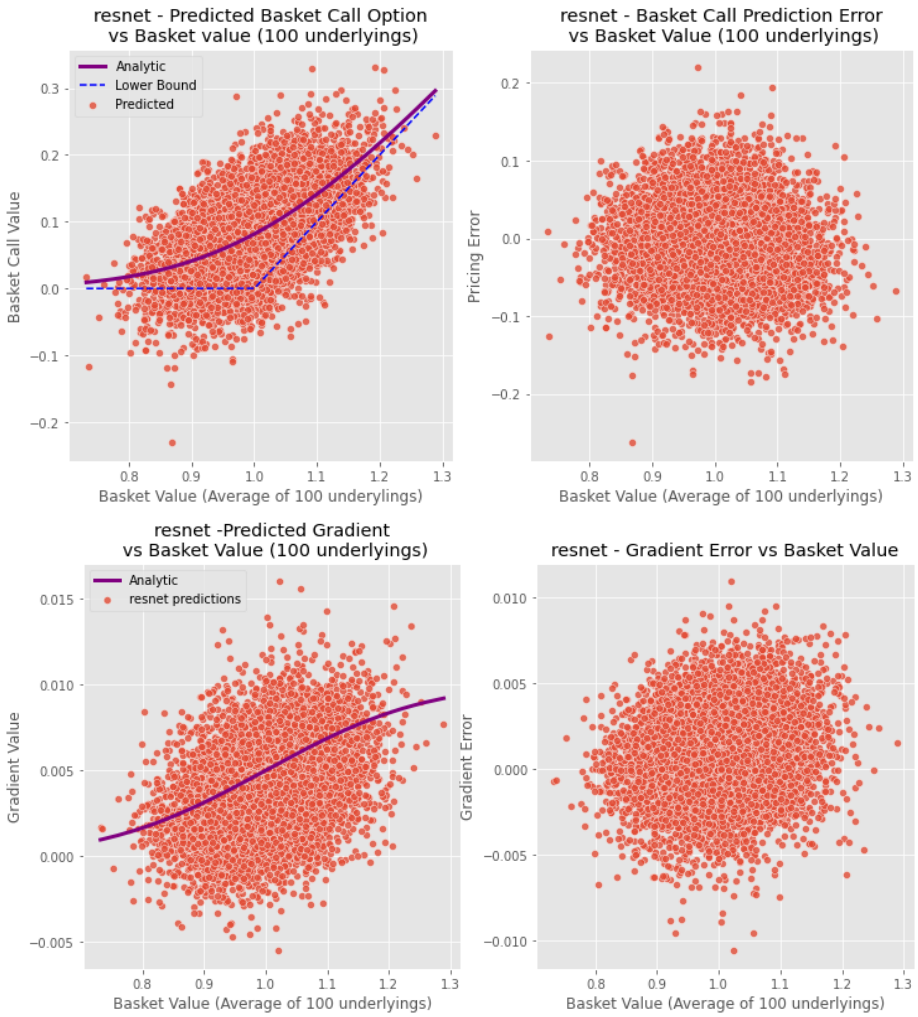


Figure 11: Residual Network

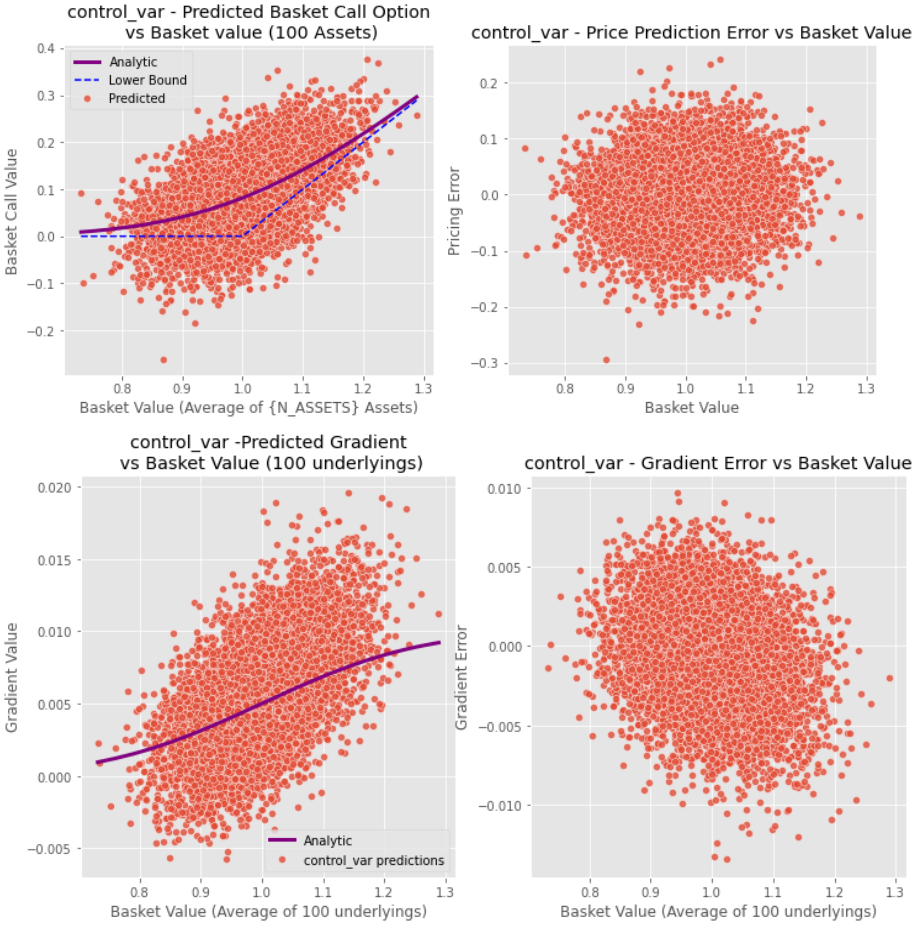


Figure 12: Control Variate

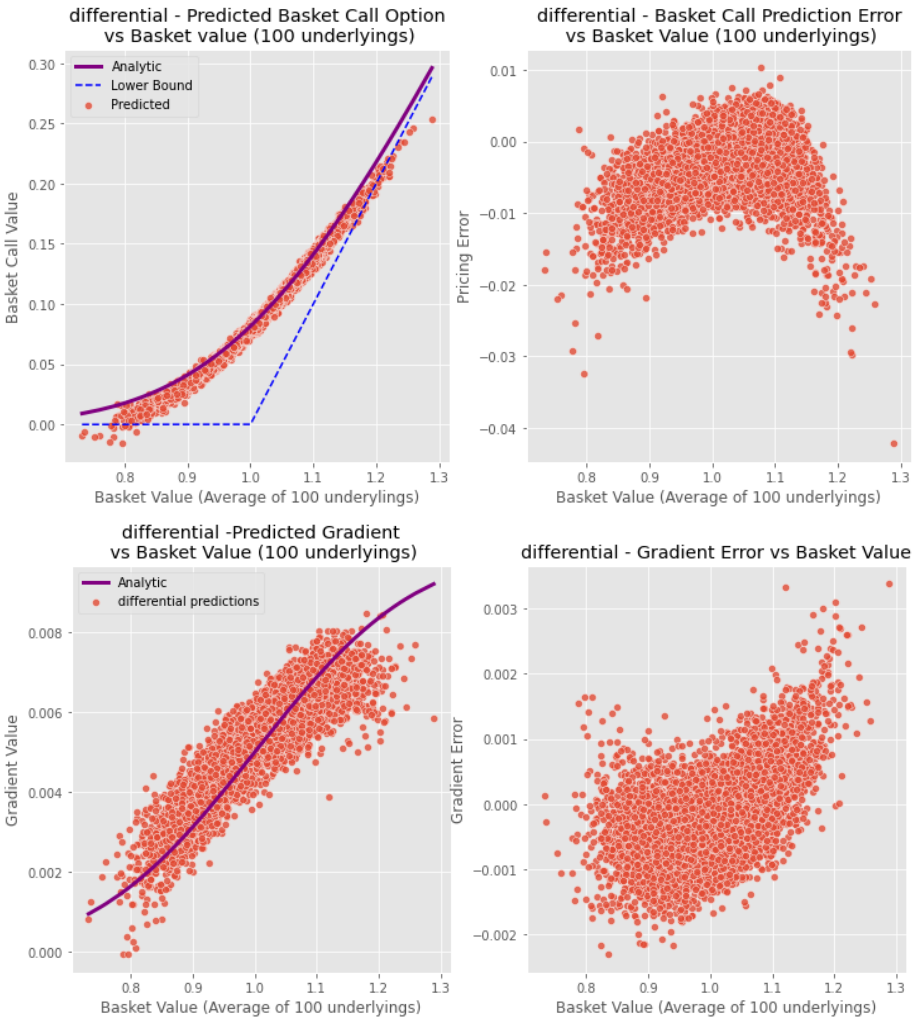


Figure 13: Differential Method

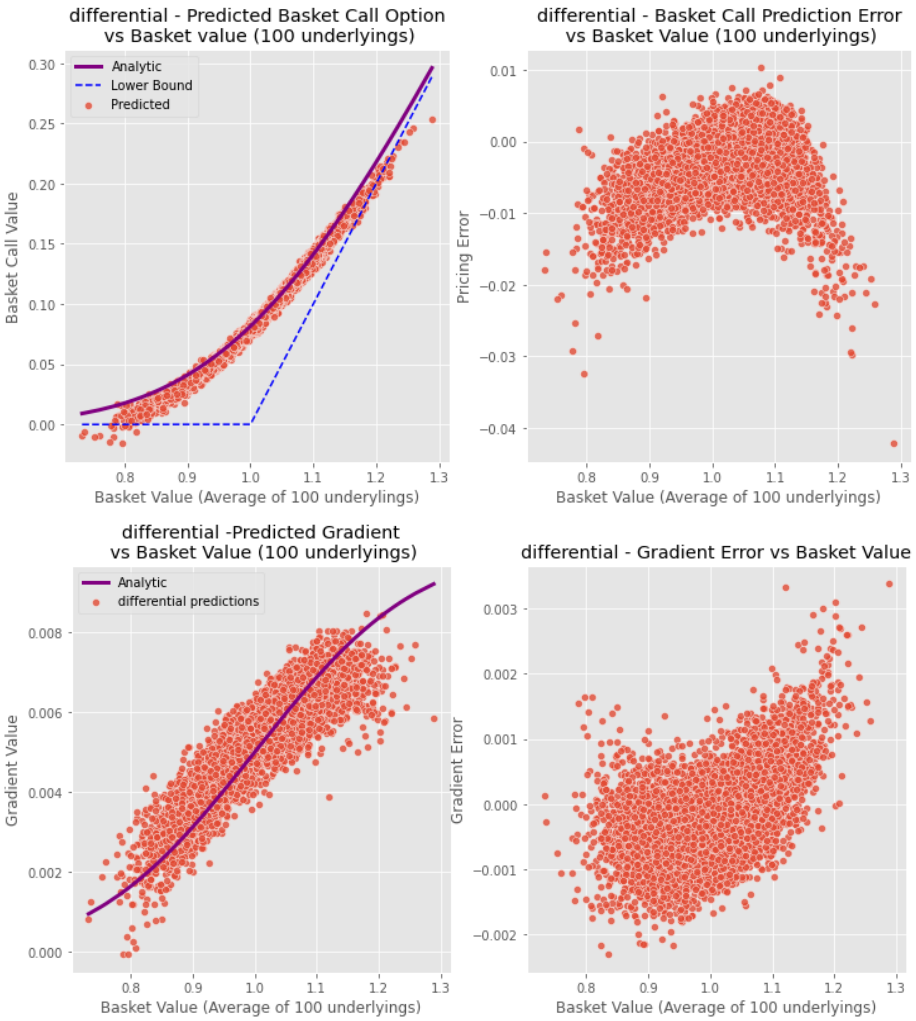


Figure 14: Control Variate

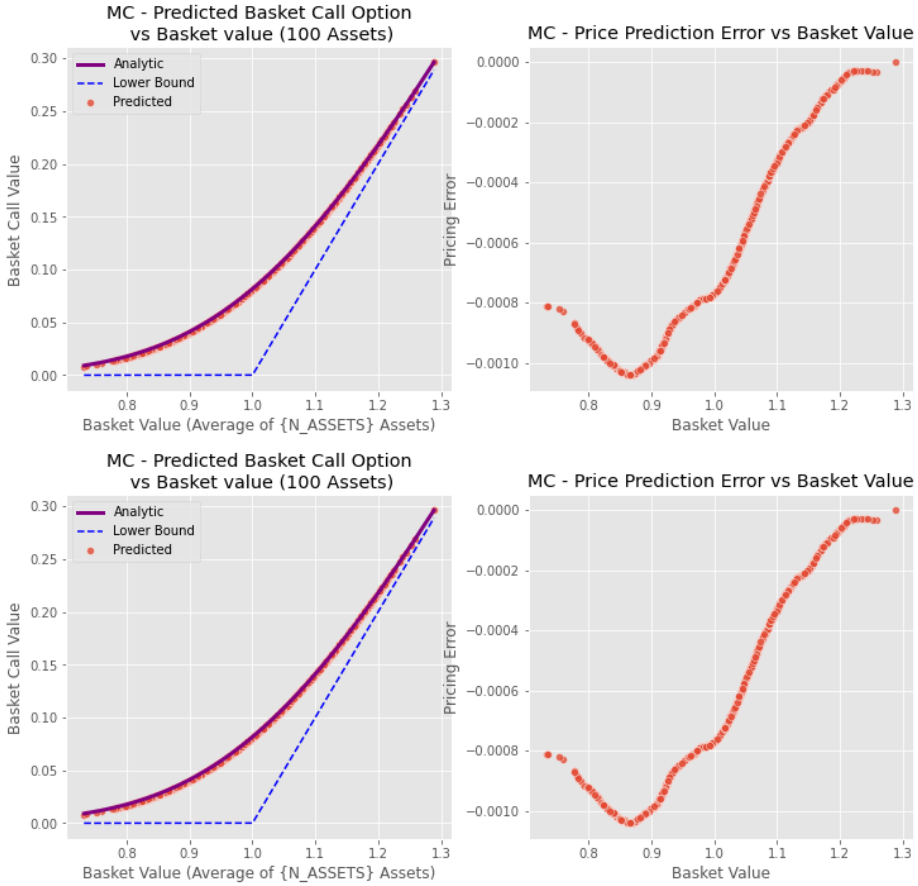


Figure 15: Monte Carlo

Rough Bergomi, European Calls

Context: As an extension to the 1 Asset European Call, we consider. Rough Volatility models may enable better fits of the implied voltility surface, but there is a lack of closed form / analytic expressions for prices, hence prices need to be approximated by Monte Carlo. This motivates the use of neural networks to approximate prices under rough volatility, and then use for inference as opposed to running MC simultions [34]. For brevity, we have included this work-in-progress experiment in Appendix A, as the results are similar to the Black-Scholes Call case.

NOTE: This experiment is still work-in progress

Dataset: In the Rough Bergomi experiment, we use the implementation for the Turbocharged Rough Bergomi scheme [2] from [48]. Our parameter space consists of:

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

$\mathbf{S_0}$	$\log(K)$	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 20: Rough Bergomi Parameter Space

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 [48], 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.

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.

Models: We consider two neural network architectures: a standard feed-forward,a gated neural network and also compare against polynomiaal 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.

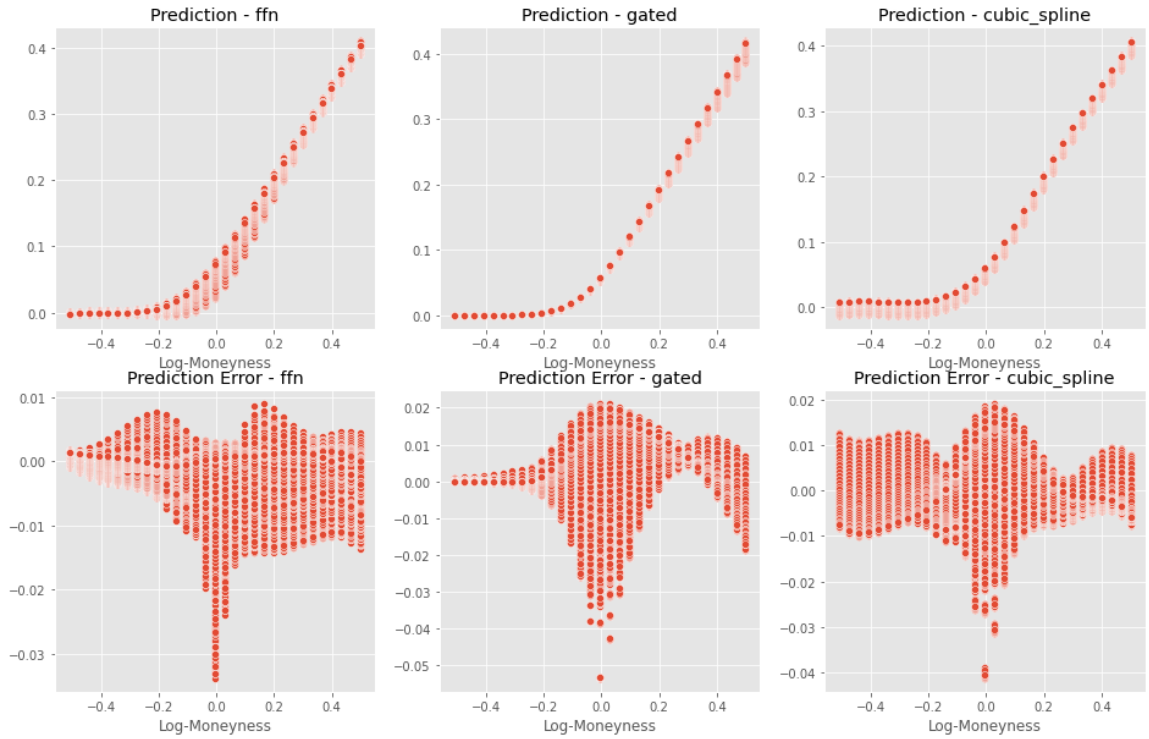


Figure 16: Errors for Rough Bergomi Approximations

Figure 17: Prices for Feed-Foward, Gated, Basis (Cubic Spline) Regression, respectively

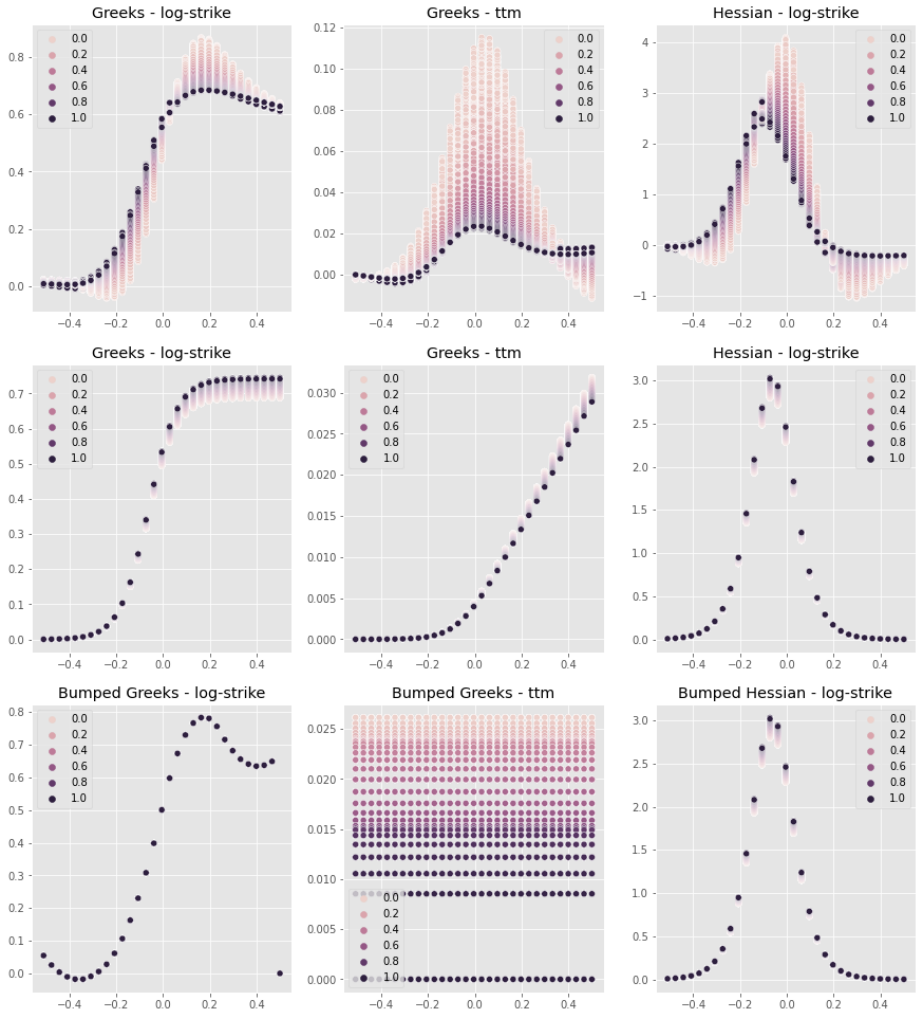


Figure 18: Greeks for Feed-Foward, Basis (Cubic Spline) Regression, Gated, respectively

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 [39] and [56]

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] Ales Cernỳ. *Mathematical techniques in finance: tools for incomplete markets*. Princeton University Press, 2009.

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

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

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

[15] 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.

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

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

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

[19] 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).

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

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

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

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

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

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

[26] Paul Glasserman. *Monte Carlo methods in financial engineering*. Vol. 53. Springer, 2004.

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

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

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

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

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

[32] Calypso Herrera et al. "Optimal stopping via randomized neural networks". In: *arXiv preprint arXiv:2104.13669* (2021).

[33] 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.

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

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

[36] 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.

[37] C. Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-Ready Applications*. O'Reilly Media, Incorporated, 2022. ISBN: 9781098107963. URL: https://books.google.co.uk/books?id=BAy%5C_zgEACAAJ.

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

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

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

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

[42] 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.

[43] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[44] Bernard Lapeyre and Jérôme Lelong. "Neural network regression for Bermudan option pricing". In: *Monte Carlo Methods and Applications* 27.3 (2021), pp. 227–247.

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

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

[47] 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.

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

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

[50] 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.

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

[52] 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.

[53] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational physics* 378 (2019), pp. 686–707.

[54] 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.

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

[56] 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).

[57] Takayuki Sakuma. “Application of deep quantum neural networks to finance”. In: *arXiv preprint arXiv:2011.07319* (2020).

[58] Steven E Shreve et al. *Stochastic calculus for finance II: Continuous-time models*. Vol. 11. Springer, 2004.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] Haojie Wang et al. “Deep learning-based BSDE solver for LIBOR market model with application to Bermudan swaption pricing and hedging”. In: *arXiv preprint arXiv:1807.06622* (2018).

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

[65] 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.