Second-Order Differential Machine Learning

Master's thesis

by

NEIL KICHLER

Under the supervision of

Univ.-Prof. Dr. rer. nat. Uwe Naumann

A document submitted in partial fulfillment of the requirements for the degree of

M. Sc. in Computer Science

at

RWTH Aachen University

October 27, 2023

Abstract

High-dimensional stochastic models are indispensable in many real-world applications, such as biology, material science and quantitative finance. Stochastic differential equations have played a crucial role in advancing these domains, as noisy environments are integral to understanding complex phenomena beyond the scope of standard dynamical system theory. However, in practice, the simulation of stochastic models requires expensive Monte Carlo methods. The recent advancements in Machine Learning provide a promising avenue for creating much faster yet accurate surrogate models, as illustrated in Differential Machine Learning. It augments the typical supervised learning process with differential data labels obtained via automatic differentiation. This additional loss factor results in an effective, unbiased form of regularization.

We extend the learning process of neural network-based surrogate models with second-order derivative information. Using forward-over-reverse mode automatic differentiation and dimensionality reduction techniques, it is feasible to find relevant second-order hessian-vector products. If the three loss terms (payoff, differential payoff, and second-order differential payoff) are balanced correctly, the additional second-order information significantly enhances the final accuracy of the model. In a Bachelier model of a basket option with n correlated assets, we observe a doubling in the final model accuracy. An improvement that cannot be obtained through the prolonged execution of the original (differential) training setup. A further case study around the Heston model is considered. The ultimate goal is to create accurate surrogate models faster to bring effective pricing and online risk assessment of complex models closer to reality. Beyond finance, Second-Order Differential Machine Learning is a generally applicable tool for finding highly efficient, accurate surrogate models of existing numerical models.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 The need for surrogate models . . . 3
  - 1.1.2 Why Differential Machine Learning? . . . 4
- 1.2 Research Objective . . . 4
- 1.3 Contributions of this thesis . . . 5
  - 1.3.1 Key Idea . . . 5
  - 1.3.2 Steps towards the main results . . . 6

2 Background 7
- 2.1 Neural Networks . . . 7
  - 2.1.1 Multi-Layer Perceptron . . . 8
  - 2.1.2 Activation Functions . . . 10
  - 2.1.3 Universal Approximation Theorem . . . 10
- 2.2 Optimization Methods . . . 11
  - 2.2.1 Gradient Descent . . . 11
  - 2.2.2 Stochastic Gradient Descent . . . 12
  - 2.2.3 Adam . . . 13
- 2.3 Algorithmic Differentiation . . . 14
  - 2.3.1 Jacobian-Vector Product . . . 14
  - 2.3.2 Vector-Jacobian Product . . . 16
- 2.4 JAX . . . 17

3 Models for Option Pricing 21
- 3.1 Bachelier . . . 21
- 3.2 Heston . . . 24
- 3.3 Implementation . . . 24
  - 3.3.1 Euler-Maryuama . . . 25
  - 3.3.2 Monte Carlo . . . 27

4 Pathwise Sensitivities 29
- 4.1 Interchanging the derivative and expectation 29
- 4.2 Applicability 31
- 4.3 Smoothing payoff functions 32

5 Differential Machine Learning 35
- 5.1 Option prices from payoff samples 36
- 5.2 Learning with pathwise derivatives 38
- 5.3 Variance Reduction 42

6 Second-Order Differential Machine Learning 43
- 6.1 In which directions should we sample? 43
  - 6.1.1 Random directions 45
  - 6.1.2 Principal Component Analysis 45
- 6.2 Main algorithm 48
- 6.3 Loss Balancing 50
- 6.4 Results 51

7 Related Work 59

8 Conclusion 61
- 8.1 Discussion 61
- 8.2 Future Directions 62

A On the code developed 63

Acronyms 67

Notation 69

Bibliography 71

1 Introduction

"Truth ... is much too complicated to allow anything but approximations."
— John von Neumann, *The Works of the Mind*

1.1 Motivation

High-dimensional stochastic models are necessary for modeling complex phenomena of the real world. Fields like biology, material science, and quantitative finance, among many others, embrace using a stochastic process during simulation, as it is most often infeasible to rely on the fundamental governing physical equations. The techniques of stochastic modelling that were initially developed in the field of statistical mechanics led to many domains that can now reason about the macro perspective without getting stuck in an intractable simulation of a more fundamental process. By adopting such a stochastic perspective one, however, introduces various uncertainties into the computation and simulation of the process.

Therefore, quantifying uncertainty is quickly becoming an indispensable part of all scientific fields. The field of uncertainty quantification is vast and we will not attempt to provide a thorough introduction. But, consider buying a simple option. That is, imagine observing the current price of your favourite company listed on a stock exchange. Let us normalize this price to be \$100. You have a good feeling that the price will go up in the next year, so you buy a call option for \$5 (its premium) that gives you the right to buy the stock at \$100 (its strike price) exactly one year from now. This is a European Call Option [1]. The advantage over directly buying the stock is that in the event of, e.g. a bad earnings call resulting in the price falling to \$80, the option will expire but you only lost its premium (\$5). In the event of the price increasing, however, you gain asymmetrically. Namely, the difference of the price at the expiration date (called maturity) and the strike price.

In short, the payoff is given by: $(S_T - K)^+$, where S_T is the price at time of maturity T and K is the strike price. The $()^+$ is shorthand for the function: $\max(x, 0)$. Of course, we do not know in advance what will happen to the price. A lot of factors can influence it, including the market volatility, liquidity, sentiment, interest rates, political events, etc. So, how certain can we really be

[1] As opposed to, e.g., an American Option which gives you the right to buy the stock from time of purchase up to the expiration date. Many other exotic options exist. We will focus, for simplicity, on the European Call Option.

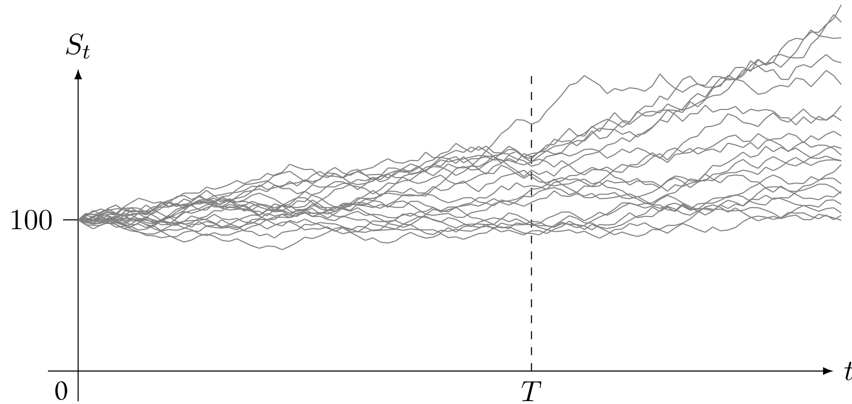
about the trade ending with a positive outcome? Or, in other words: What is the expectation of the payoff of this option? What we are interested in is the price or value V of the option[2]:

$$V = \mathbb{E}[\nu(S_T, K)],$$

where $\nu(S_T, K) = (S_T - K)^+$ is the payoff function defined as before. To get closer to answering this question we must build up a model. A model that can express the dynamics of the spot price S_t of the underlying asset at time t. In general, we consider a *Stochastic Differential Equation* (SDE) which for now can be described abstractly by:

$$\mathrm{d}S_t = a(S, t)\,\mathrm{d}t + b(S, t)\,\mathrm{d}W_t,$$

where $\mathrm{d}W_t$ denotes a Wiener process, also known as Brownian motion, and $a : \mathbb{R} \times [0, T] \to \mathbb{R}$, $b : \mathbb{R} \times [0, T] \to \mathbb{R}$ are functions describing the drift and volatility of the price. We can now sample from such a model which could lead to the following trajectories:



Given infinitely many paths of such an ergodic, stochastic process we could finally find a price by averaging the payoffs of all paths. This is of course computationally not feasible. Instead, we will describe implementation techniques to find approximate solutions alongside some popular models in chapter 3. For now, we want to finish this introduction with the heart of the problem domain: How sensitive is the computed price to the initial configuration? We are thus interested in:

$$\frac{\partial V}{\partial \theta},$$

where θ represents the input parameter we wish to take the partial derivative with respect to. Considering the sensitivity to the initial spot price S_0, we get $\frac{\partial V}{\partial S_0}$. These price sensitivities are known as the Greeks in finance. The naming will later become apparent. Sensitivity analysis provides a

[2]For now we do not consider any discounting factor based on, e.g., risk-free positive interest rates.

powerful toolset to quantify uncertainty in a model. In particular, we concentrate on derivative based methods. Through the advancements in Algorithmic Differentiation (AD) it is possible, with machine precision, to efficiently find (higher-order) derivative information for many computer programs. It is a fundamental building block of this work and will therefore be reviewed in detail in section 2.3 of the background chapter.

Financial institutions have great interest in understanding such uncertainties as they impose risk. Risk management has become an integral part of the financial industry. After the financial crisis of 2008 many countries have realized the huge impact this industry can have to the entire economy. As a result, the financial sector is now being put under more scrutiny. With regulations like BASEL 3 the financial institutions must ensure that their computations adjust the value of derivatives to consider various risks.

The above scenario will be the basis for the proceeding work as it captures the essence of the fundamental challenge posed in options trading. Instead of describing many models in varying domains, we thus focus exclusively on models from options trading in this thesis. Besides its direct real world applicability, it provides intuitive interpretations of the quantities we are interested in. But the presented methods can be used beyond the realm of quantitative finance.

1.1.1 The need for surrogate models

The techniques for finding values based on averaging over a large set of samples are known as Monte Carlo (MC) methods. The downside of such methods is that they are computationally very expensive. For very simple models, analytic solutions may exist. But, in general, using more interesting models or exotic payoff functions will lead to models without analytic solutions. The Heston model [19] is an example of a stochastic volatility model that, in general, has no analytical closed form solution. As a special case, it has an analytical solution for a European payoff with fixed parameters. This makes it an ideal candidate for a case study of the proposed methods. It is described in chapter 3 amongst more fundamental models. As an alternative, we must find surrogate models that execute much faster while staying sufficiently accurate.

The advancements in Machine Learning (ML) provide a promising avenue for the creation of much faster, yet accurate surrogate models. Especially in high-dimensional settings, neural network based methods excel because, in many occasions, they can overcome the curse of dimensionality. Such surrogate models can be used in production settings where there exists a need for near real time analysis and simulation. However, they are prone to overfitting and usually require some form of regularization which will introduce a potential bias. Furthermore, they require large training data sets and perform poorly in recovering the underlying risk variables.

1.1.2 Why Differential Machine Learning?

The methods of Differential Machine Learning that were introduced by Huge and Savine [21] provide a promising avenue to tackle the traditional challenges neural network based surrogates face. They demonstrate that incorporating differential data during training can significantly accelerate the convergence speed and the accuracy of the resulting model. This approach forms the foundation of the ideas explored in this thesis. Differential data refers to pathwise data samples $\frac{\partial y}{\partial x}$ which can be used on top of the usual values x and labels y used during training. First shown by Giles and Glasserman [15], such pathwise sensitivities can be efficiently found by using adjoint AD[3]. But, how do we recover the Greeks from pathwise sensitivities? In the seminal work of they show that one can recover the prices of an option by performing the typical Least-Squares method over the payoffs of the individual paths. The power of this method lies in the fact that many prices can be estimated with computed payoffs of a single MC pricing run. The samples, while noisy, remain unbiased. Furthermore, this method generalizes to first and higher-order sensitivities. As neural networks are prime candidates for performing least-squares regression the same principles will apply. The precise formulation and use of pathwise sensitivities will be covered in chapter 4.

So, to answer the question: We use Differential Machine Learning because it can efficiently learn surrogate models for predicting the prices and price sensitivities of arbitrary models given limited unbiased samples.

1.2 Research Objective

A natural question arises: If differential data can improve the accuracy of a surrogate model, can second-order differential data improve its accuracy even further? We want to address this question and further split out our research interest into the following research questions.

Research Questions:

- How does the accuracy and training speed of learned surrogate models change when trained with second-order differential data?
- What are effective strategies for finding meaningful tangent directions to be used in second-order differential ML?
- How should the training procedure balance the different loss terms?

A fundamental issue that immediately arises when we consider second-order differential data is that the computation of the Hessian will often turn out to be intractable. Therefore, we require

[3] Also known as reverse-mode AD or (more narrowly defined) as backpropagation in ML.

techniques to efficiently sample various directions via hessian-vector products. In many other domains, similar ideas have already been applied [34, 44].

Note that we do not consider the problem of calibrating the model we wish to find a surrogate for. Much work has already gone into this problem. Instead, we assume the parameters of the reference model to be calibrated already. Polala and Hientzsch [39] extend the ideas of Differential ML to learning from varying parameter sets. They further perform global optimization to find optimal parameter sets as needed in calibration. Moreover, Neural Differential Equations [24] provide a promising avenue for finding generating models that adhere to the dynamics of the training data. By injecting neural networks as parameters into the stochastic differential equation, Neural SDEs can be used for fine-tuning existing models or as an alternative to off-the-shelve models. Most important, Neural SDEs are fully differentiable making them ideal candidates to learn surrogate models from. This brief aside should provide surrounding context to of the general applicability.

1.3 Contributions of this thesis

The contributions of this thesis are as follows:

- We present a generalized formulation of Differential Machine Learning to arbitrary neural network based surrogate models.
- We replicate the key ideas presented by Huge and Savine [21] within this generalized framework.
- We extend the method to second-order differential data - what we refer to as Second-Order Differential ML.
- We implement efficient versions of the above in JAX [5] that can run on GPUs and TPUs.
- We experimentally compare the different methods in terms of training speed and accuracy of the learned surrogate model.
- We compare random directions for HVPs with directions from principal components of the PCA.

1.3.1 Key Idea

For the impatient, we provide a very brief overview of the key ideas presented in this thesis. Instead of just learning a neural network surrogate model via the output values generated by the reference model, Differential Machine Learning proposes to learn with derivative information that can be obtained via adjoint AD. We extend the idea to second-order derivative information. However,

this is often infeasible since the full Hessian is too expensive to compute at each iteration. Instead, we extract the directions of maximal variance in the derivative information using PCA. Those directions are then used to compute hessian-vector-products which further inform the learning.

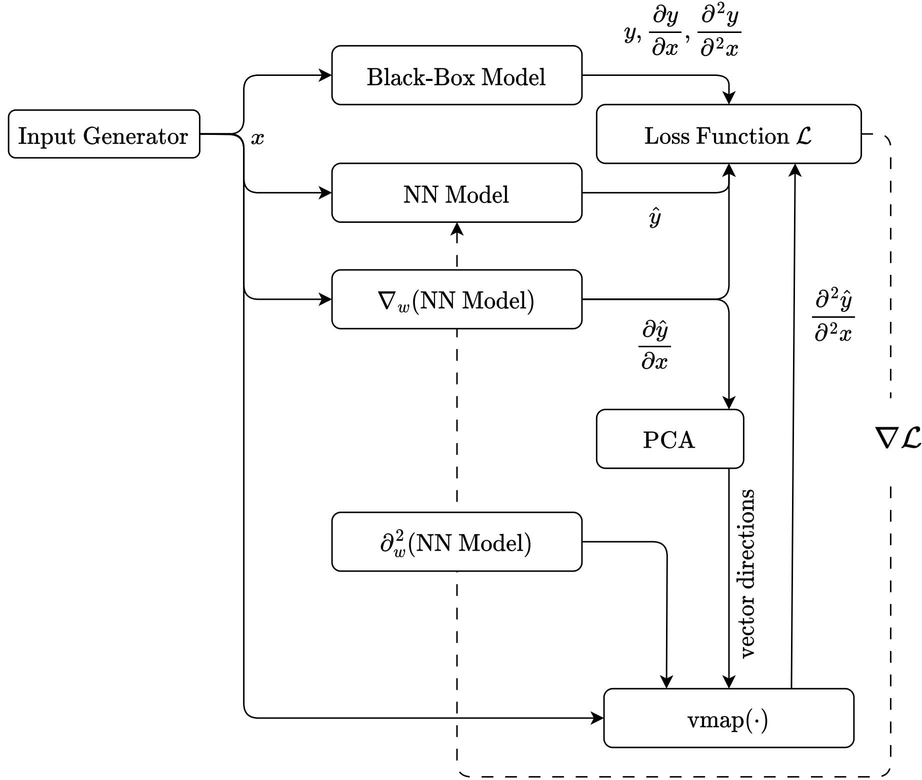


Figure 1.1: Visualization of Second-Order Differential Machine Learning.

1.3.2 Steps towards the main results

Before we describe the details of the outlined idea in chapter 6, we introduce the various concepts needed for understanding the method. The structure is such that we incrementally build up the methods needed for implementing Second-Order Differential ML. In chapter 2, the basics of neural networks, AD, and its implementation in JAX will be reviewed. While in chapter 3 we present common option price models that form the basis of learning surrogates. The mathematical justification of pathwise payoffs and sensitivities for pricing is given in chapter 4. We then introduce a generalized view on Differential ML in chapter 5, before ending up at Second-Order Differential ML. There we will present the final results. We compare our method to related work in chapter 7. A conclusion with directions for future research is given in chapter 8.

2 Background

"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin."

— Rich Sutton, *The Bitter Lesson*

In this chapter, we provide the necessary background knowledge that will be required to understand many of the implementation details discussed throughout the next chapters. A very brief introduction into neural networks and its learning process is given in section 2.1. After understanding back-propagation, this chapter dives deeper into the foundations of Algorithmic Differentiation (AD) which is required to implement it efficiently for arbitrary functions. AD will play a crucial role in this thesis and it is thus highly advised to understand the ideas presented in section 2.3. Finally, much of the presented code is implemented in JAX [5]. We provide an overview and reasons for this choice in section 2.4.

2.1 Neural Networks

A neural network is a very general concept which, at its core, interleaves linear operations (e.g., matrix multiplication, convolution) with nonlinear operations (activation functions) to approximate arbitrary functions. Let $f_{\boldsymbol{\vartheta}} : \mathcal{X} \to \mathcal{Y}$ denote the learned function of the neural network with trainable parameters $\boldsymbol{\vartheta}$[1]. These parameters are called the weights of the network. The function $f_{\boldsymbol{\vartheta}}$ of the neural network learns from samples of input domain $\mathcal{X}$ to predict an output in $\mathcal{Y}$ such that it closely models the target function. This learning procedure represents an optimization problem which can be stated as follows:

$$\boldsymbol{\vartheta}^* = \arg\min_{\boldsymbol{\vartheta}} \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2\right], \quad \hat{\boldsymbol{y}} = f_{\boldsymbol{\vartheta}}(\boldsymbol{x}), \tag{2.1}$$

where $\|\cdot\|_2$ is the L^2 norm and $\mathcal{D}$ generates data samples $(\boldsymbol{x}, \boldsymbol{y})$ based on the reference model. So, it minimizes the error by comparing the predicted output $\hat{\boldsymbol{y}}$ of the hypothesis function with the target output $\boldsymbol{y}$. Throughout this thesis, a $\hat{\cdot}$ will highlight that we deal with an approximation.

[1] We deviate from the typical notation and use $\boldsymbol{\vartheta}$ instead of $\boldsymbol{\theta}$ for the weights since we already use $\boldsymbol{\theta}$ to represent the parameters of the option pricing model. A list of notation can be found on page 69.

We thus want to minimize the expected generalization error, called *risk*. The learning problem can be stated more generally by:

Definition 2.1 (Risk)**.**

$$R_{\mathcal{D}}(\boldsymbol{\vartheta}) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}), \boldsymbol{y})\Big], \tag{2.2}$$

where $\mathcal{L}$ is the loss (e.g. L^2). However, we cannot consider the entire data distribution and must instead use a training set or sampler $\mathcal{S}$. The samples of $\mathcal{S}$ are independently drawn from $\mathcal{D}$. The observable risk thus differs from the risk and instead is called *empirical risk*.

Definition 2.2 (Empirical Risk)**.**

$$R_{\mathcal{S}}(\boldsymbol{\vartheta}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \boldsymbol{y}_i), \tag{2.3}$$

where $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{m} \sim \mathcal{S}$. Optimizing the empirical risk as a proxy for the true risk sets machine learning apart from traditional optimization where no such distinction exists. Since we will be dealing almost exclusively with the empirical risk, we will use the shorthand notation $\mathcal{J}$ and consider it the cost function.

Definition 2.3 (Cost)**.**

$$\mathcal{J}(\boldsymbol{\vartheta}) = R_{\mathcal{S}}(\boldsymbol{\vartheta}) \tag{2.4}$$

2.1.1 Multi-Layer Perceptron

A common neural network structure is the Multi-Layer Perceptron (MLP). It interleaves a linear mapping, i.e. a matrix multiplication of the weights with the input vector, with a nonlinear activation function σ. Since the target labels could be offset from the origin, the MLP adds a bias term $\boldsymbol{b}$. In total it has L layers resulting in:

$$f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) = \boldsymbol{W}^{[L-1]}\sigma \odot (\cdots(\boldsymbol{W}^{[1]}\sigma \odot (\boldsymbol{W}^{[0]}\boldsymbol{x} + \boldsymbol{b}^{[0]}) + \boldsymbol{b}^{[1]})\cdots) + \boldsymbol{b}^{[L-1]}, \tag{2.5}$$

where

$$\begin{aligned}
\boldsymbol{W}^{[l]} &\in \mathbb{R}^{n_{l+1}\times n_l} \text{are the weights of layer } l,\\
\boldsymbol{b}^{[l]} &\in \mathbb{R}^{n_{l+1}} \text{is the bias of layer } l,\\
\boldsymbol{\vartheta} &= (\boldsymbol{W}^{[0]}, \boldsymbol{W}^{[1]}, \ldots, \boldsymbol{W}^{[L-1]}, \boldsymbol{b}^{[0]}, \boldsymbol{b}^{[1]}, \ldots, \boldsymbol{b}^{[L-1]}).
\end{aligned}$$

The $\odot$ denotes an elementwise operation as is the case for the activation function. As a convenient shorthand notation, consider a linear layer.

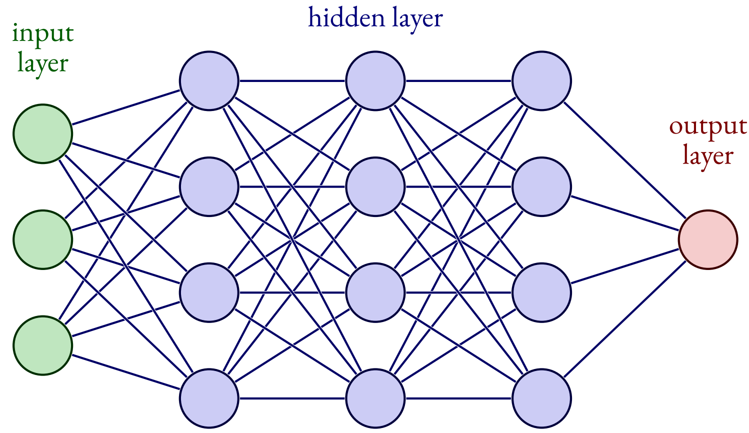


Figure 2.1: Visualization of a MLP with 3 hidden layers.

Definition 2.4. $\text{Linear}_{n_{l+1}\leftarrow n_l} = \boldsymbol{W}^{[l]}\boldsymbol{x} + \boldsymbol{b}^{[l]}$, with $\boldsymbol{W}^{[l]}$ and $\boldsymbol{b}^{[l]}$ as defined above.

Example 2.1.1 (MLP with 3 hidden layers)**.** Let $f_{\boldsymbol{\vartheta}} : \mathbb{R}^3 \to \mathbb{R}$ represent the learned function defined as:

$$f_{\boldsymbol{\vartheta}} = \text{Linear}_{n_4\leftarrow n_3} \circ \sigma \circ \text{Linear}_{n_3\leftarrow n_2} \circ \sigma \circ \text{Linear}_{n_2\leftarrow n_1} \circ \sigma \circ \text{Linear}_{n_1\leftarrow n_0},$$

where $n_0 = 3, n_1 = 4, n_2 = 4, n_3 = 4, n_4 = 1$, and $\sigma = \text{ReLU}$ is performed elementwise. The $\circ$ is function composition defined as usual. A visual representation of the network is shown in Figure 2.1.

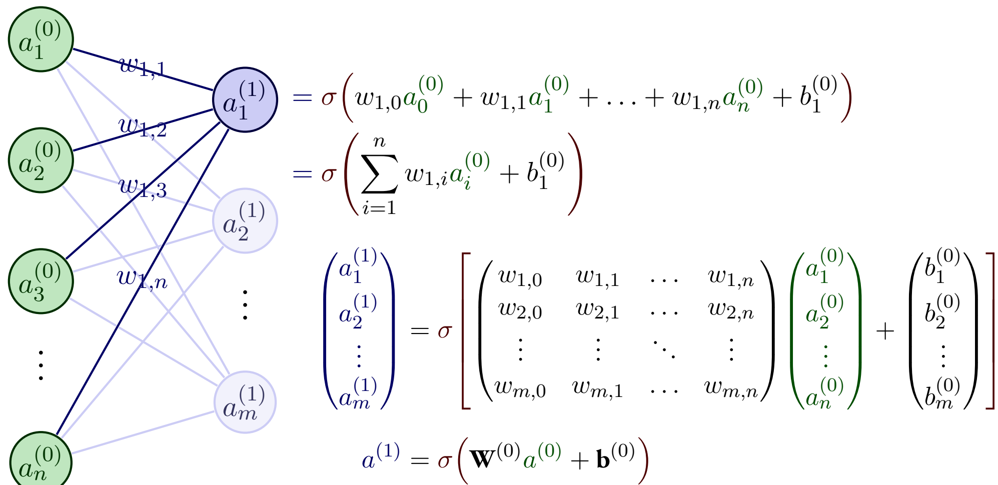
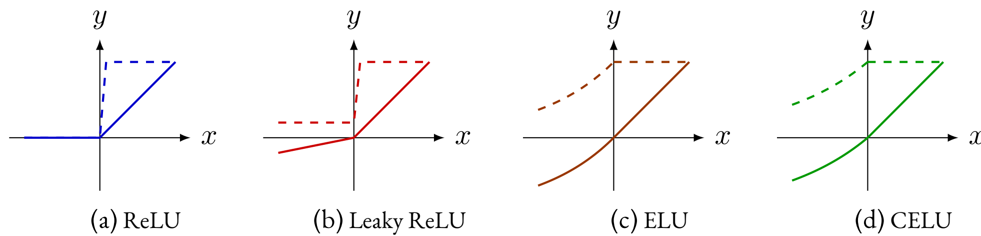


Figure 2.2: Visualization of a hidden layer.

The edges represent the weights of the linear layer. A hidden node is a non-linear activation implemented through σ. A detailed representation of a hidden layer is shown in Figure 2.2.

2.1.2 Activation Functions

The activation function σ is often chosen to be a Rectified Linear Unit (ReLU) which is just elementwise $\max(\boldsymbol{x}, \boldsymbol{0})$. This function is theoretically not differentiable at 0. In practice, it turns out that a piecewise linear function will produce correct derivative information for the individual linear pieces, producing a Heaviside step function. However, we want to find second-order derivative information of this model and thus need to consider a smoothed version of this function. Otherwise, the second derivative will collapse to 0 everywhere and will not follow the theoretical Dirac delta function[2]. Many modified ReLU activation functions exist, like Leaky ReLU, ELU, CELU etc.



(a) ReLU (b) Leaky ReLU (c) ELU (d) CELU

Figure 2.3: Visualization of various activation functions and their derivatives.

2.1.3 Universal Approximation Theorem

To understand the expressiveness of neural networks, we highlight that specific neural network configurations can provably approximate any continuous function on compact domain.

Theorem 2.1. Let $\mathcal{X} \subseteq \mathbb{R}^p$ be compact and $f : \mathcal{X} \to \mathbb{R}^q$ continuous. Then for every $\varepsilon > 0$, there is a MLP of depth 2 with continuous, nonpolynomial activation function, and learned function $f_{\boldsymbol{\vartheta}}$ such that:

$$\|f(\boldsymbol{x}) - f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\| \leq \varepsilon,$$

for all $\boldsymbol{x} \in \mathcal{X}$.

Proof. An overview of many variations of universal approximation (including Theorem 2.1) and their corresponding proofs is given by Pinkus [38]. □

Remark. The hidden layer of the neural network can be arbitrarily large.

[2] Which actually is a distribution or generalized function.

However, in practice many networks do not increase the width of the individual hidden layers by much. Instead, the rise of Deep Learning has shown that a deep neural network can be particularly effective. Kidger and Lyons [25] have recently proven universal approximation for a much wider set of narrow yet deep networks. They even consider almost nowhere differentiable activation functions.

Theorem 2.2. Let $\mathcal{X} \subseteq \mathbb{R}^p$ be compact and $f : \mathcal{X} \to \mathbb{R}^q$ continuous. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a nonaffine, continuous activation function which is continuously differentiable with nonzero derivative at at least one point. Then for every $\varepsilon > 0$, there is a MLP of arbitrary depth, with limited width hidden layers, and learned function $f_{\boldsymbol{\vartheta}}$ such that:

$$\|f(\boldsymbol{x}) - f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\| \leq \varepsilon,$$

for all $\boldsymbol{x} \in \mathcal{X}$.

Proof. See [25]. □

But how do neural networks efficiently learn in practice? This is what we discuss next.

2.2 Optimization Methods

To minimize the error between the neural network output and the target data we need an optimization algorithm that can minimize the Cost, i.e.:

$$\arg\min_{\boldsymbol{\vartheta}} \mathcal{J}(\boldsymbol{\vartheta}) \tag{2.6}$$

At the time of this writing, Adam [28] is the standard baseline optimizer in use for learning with neural networks. It is a variant of Stochastic Gradient Descent (SGD) and will be used in all experiments to be comparable with previous results. We therefore introduce the algorithm incrementally, starting with Gradient Descent and SGD, before explaining Adam in subsection 2.2.3.

2.2.1 Gradient Descent

Gradient descent minimizes the cost function using the following iterative scheme:

$$\boldsymbol{\vartheta} \leftarrow \boldsymbol{\vartheta} - \eta \nabla_{\boldsymbol{\vartheta}} \mathcal{J}(\boldsymbol{\vartheta}), \tag{2.7}$$

where η is the learning rate and $\nabla_{\boldsymbol{\vartheta}}$ is the gradient operator with respect to the parameters $\boldsymbol{\vartheta}$. The algorithm stops when $\nabla_{\boldsymbol{\vartheta}} \mathcal{J}(\boldsymbol{\vartheta}) < \epsilon$, for some tolerance ϵ, or when the training stops after a fixed iteration count. Various strategies exist for choosing and adjusting η over the course of training.

We will mostly stick to a small constant or learning rate scheduling. In learning rate scheduling the η is adapted based on a fixed scheduling function $s : \mathbb{N} \to \mathbb{R}$. It takes in the current training step index and returns a learning rate η.

Alternatively, the learning rate can be improved via line search. The theoretic optimum is found for η if $\mathcal{J}(\boldsymbol{\vartheta} - \eta \nabla_{\boldsymbol{\vartheta}} \mathcal{J}(\boldsymbol{\vartheta}))$ is minimized. But this is infeasible in practice and instead, one has to sample some possible η and pick the best candidate. Often a more effective strategy is to backtrack if the learning rate does not decrease the objective function and shrink η repeatedly until it does.

Note that for convex objective functions the number of steps the gradient descent algorithm is going to take is independent in the dimension of the search space. Its convergence rate is in this case $\mathcal{O}(\frac{1}{k})$, where k is the number of iterations [53, p. 14]. So, no matter how large the problem gets, the number of training iterations stays the same. However, in practice many problems are, of course, not convex. For example, the principal component analysis (PCA), something which we will discuss in subsection 6.1.2, is a nonconvex problem[3], but it provably has only a single global optimum! High-dimensional problems will often have many saddle points to overcome before reaching the global optimum. How fast an optimization algorithm can get out of such saddle points is thus vital. A natural extension is to consider stochastic optimization algorithms.

2.2.2 Stochastic Gradient Descent

A downside of gradient descent is that it considers the entire dataset. It can therefore be infeasible to use in practice as the training iteration time increases with the size of the dataset. Instead, consider only a batch of samples which get randomly chosen from the training sampler. Then, the gradient of the loss is only computed for this so called *minibatch*. We denote this approximate gradient by $\hat{\boldsymbol{g}}$. Again, similar comments on the learning rate apply as in the previous section.

Algorithm 2.1 Stochastic Gradient Descent (SGD)

Require: Initialized parameters $\boldsymbol{\vartheta}$, training sampler $\mathcal{S}$.
while $\boldsymbol{\vartheta}$ not converged **do**
  $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{m} \sim \mathcal{S}$ ▷ Sample training data
  $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\vartheta}} \sum_{i=1}^{m} \mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ ▷ Gradient of minibatch
  $\boldsymbol{\vartheta} \leftarrow \boldsymbol{\vartheta} - \eta \hat{\boldsymbol{g}}$ ▷ Update
end while

In practice, SGD benefits from being able to escape sharp local minima and invokes a form of implicit regularization of the parameters [46]. Many alternative formulations have been developed to accelerate the convergence of the learning procedure. Including adding Nesterov Momentum [51], using AdaGrad [12], RMSProp [50], or an approximate second-order method like L-BFGS

[3] Finding the smallest and largest eigenvalues of the PCA, however, is again a convex optimization problem.

[31]. The next algorithm combines some of the benefits of the aforementioned first-order optimization methods and has established itself as a baseline for many tasks. However, most optimization methods will work and find similar optima. Fine-tuning of the optimizer parameters and the learning rate will often play an equally important role in the performance of learning neural networks [48].

2.2.3 Adam

Momentum aids in faster convergence as it uses velocity information to guide the trajectory. Similar to how a ball rolling down a large bowl will accelerate, these methods use the gradient to compute an additional momentum term that is used instead for updating the parameters. However, the ball could roll beyond the minimum various times. Nesterov acceleration balances momentum by having the correction factor consider an optimistic parameter update [48]. Adam embeds the momentum in the computation of the first and second moments separately (here $\boldsymbol{m}, \boldsymbol{v}$). It then uses those moments to update the parameters. The bias correction is used to correct for potential bias in the initial few iterations [17, pp. 301–302].

Algorithm 2.2 Adam [28], with $\epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$ by default.

Require: Initialized parameters $\boldsymbol{\vartheta}$, training sampler $\mathcal{S}$.
- $\boldsymbol{m}, \boldsymbol{v} \leftarrow \mathbf{0}, \mathbf{0}$ ▷ Initialize first and second moment
- $t \leftarrow 0$ ▷ Time step
- **while** $\boldsymbol{\vartheta}$ not converged **do**
  - $t \leftarrow t + 1$
  - $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n} \sim \mathcal{S}$ ▷ Sample training data
  - $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\vartheta}} \sum_{i=1}^{m} \mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$ ▷ Gradient of minibatch
  - $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1)\hat{\boldsymbol{g}}$ ▷ Update first moment (biased)
  - $\boldsymbol{v} \leftarrow \beta_2 \boldsymbol{v} + (1 - \beta_2)(\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}})$ ▷ Update second moment (biased)
  - $\tilde{\boldsymbol{m}} \leftarrow \boldsymbol{m}/(1 - \beta_1^t)$ ▷ Bias corrected
  - $\tilde{\boldsymbol{v}} \leftarrow \boldsymbol{v}/(1 - \beta_2^t)$ ▷ Bias corrected
  - $\boldsymbol{\vartheta} \leftarrow \boldsymbol{\vartheta} - \eta \frac{\tilde{\boldsymbol{m}}}{\epsilon + \sqrt{\tilde{\boldsymbol{v}}}}$ ▷ Update
- **end while**

The initial proof of Adam's convergence rate given in [28] was shown to be incorrect [43]. There exist hyperparameter configurations and problems whereby Adam does not converge. However, in practice these scenarios rarely occur. Nonetheless, this led to an emergence of many proofs with stricter assumptions or modified algorithms with correct convergence bounds [11]. Modern alternatives include AdamW [33] and AdaBelief [57] but we do not consider them to better compare against existing results.

2.3 Algorithmic Differentiation

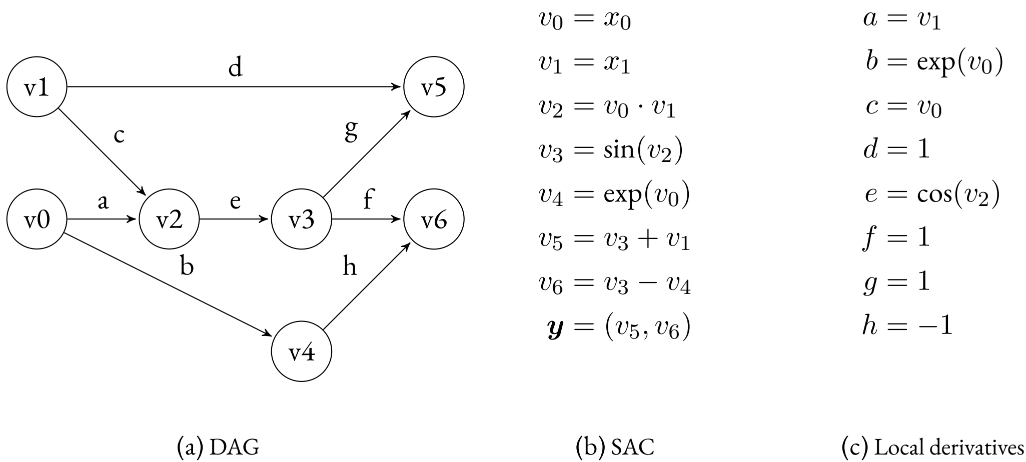
In this section, we describe the fundamental operations of Algorithmic Differentiation (AD). It has become the standard technique for obtaining derivative information as it is both efficient and accurate (up to machine precision). Compare this to finite-difference schemes which incur round-off and truncation error. In addition, the computational complexity scales with the number of inputs for finite-differences which for machine learning is infeasible to consider. Symbolic differentiation, on the other end of the spectrum, is accurate, however, suffers from an explosion of the expression length during the elaboration phase of e.g., the product rule. Instead, AD makes use of the chain rule to break up the computation of the derivative into elemental functions whose derivatives are known. It leverages the compiler and language features, like operator overloading, to effectively find derivatives automatically. We first discuss the Jacobian-Vector Product in subsection 2.3.1, where we apply tangent mode (or forward mode) AD, before presenting the Vector-Jacobian Product in subsection 2.3.2 which uses adjoint mode (or reverse mode) AD.

2.3.1 Jacobian-Vector Product

Any computer program representing a function can be split up into elementary functions that build up a computational *Directed Acyclic Graph* (DAG). This *Single Assignment Code* (SAC) is then used to obtain the local derivatives of the elementary functions.

Example 2.3.1. Let $f : \mathbb{R}^2 \to \mathbb{R}^2$,

$$f(\boldsymbol{x}) = f(x_0, x_1) = (\sin(x_0 \cdot x_1) + x_1, \sin(x_0 \cdot x_1) - \exp(x_0))$$



(a) DAG (b) SAC (c) Local derivatives

Figure 2.4: AD internals of Example 2.3.1.

The Jacobian $\boldsymbol{J}$ can be computed by applying the chain rule of differentiation. For the example we get:

$$\boldsymbol{J} = \begin{pmatrix} \frac{\partial v_5}{\partial v_0} & \frac{\partial v_5}{\partial v_1} \\ \frac{\partial v_6}{\partial v_0} & \frac{\partial v_6}{\partial v_1} \end{pmatrix} = \begin{pmatrix} a \cdot e \cdot g & d \\ b \cdot h + a \cdot c \cdot f & c \cdot e \cdot f \end{pmatrix} = \begin{pmatrix} v_1 \cos(v_2) & 1 \\ -\exp(v_0) + v_1 v_0 & v_0 \cos(v_2) \end{pmatrix}.$$

Implementations of tangent mode often extend objects to have a corresponding local derivative entry for every variable, commonly referred to as AD by dual numbers. This is, however, slightly misleading because the output value of an operation never depends on the input derivative [4].

Example 2.3.2. Consider the MLP described in Example 2.1.1.

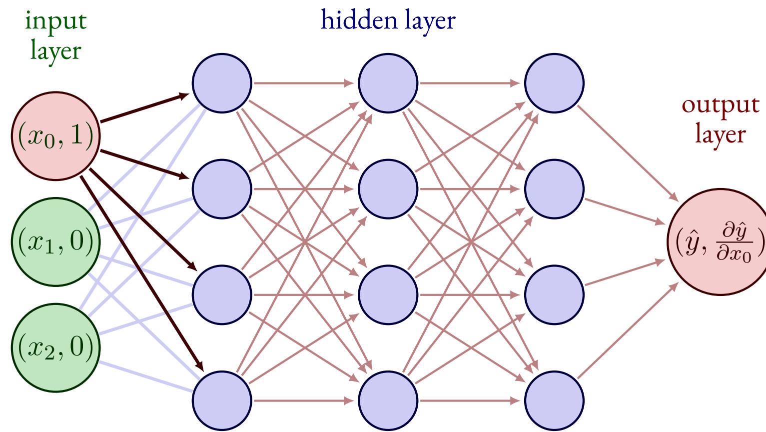


Figure 2.5: Computing the directional derivative with vector $\boldsymbol{x}^{(1)} = \boldsymbol{e}_0$ in tangent mode AD.

Tangent AD yields directional derivatives by *seeding* the local derivative entry of the input value and propagating this derivative value throughout the computation. Seeding refers to the process of setting the local derivative entry to 1 for the variable we wish to differentiate with respect to. Example 2.3.2 shows an MLP being seeded with $\boldsymbol{e}_0 = (1, 0, \ldots)^T$, i.e. the derivative entry of the first node of the input layer is activated. The derivative with respect to x_0 is thus found. The complete Jacobian is now found by n_0 executions of the Jacobian-Vector Product (JVP) seeded with the Cartesian basis vectors $\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_{n_0}$. Note that, despite its name, we do not have to materialize the Jacobian for a JVP. Its memory usage is thus only two times the original (for storing the derivative entries). The computational complexity of tangent AD, however, scales with the number of input variables, here n_0. For dense neural networks optimized with a loss function we usually have many input parameters but few output variables (often only one for the final

[4] The dual number system, of course, deals with this by taking $\varepsilon^2 = 0$, but when implementing the derivatives for elementary functions, the programming language will not guard you from using input derivative values in the computation of the output value.

loss value). Tangent AD is thus a very bad choice in this case and machine learning with neural networks only started to see practical results after considering adjoint AD, which we consider next.

2.3.2 Vector-Jacobian Product

The work-horse in optimizing neural networks is adjoint AD. This is also commonly known as *backpropagation*. In addition, finding the sensitivities of an option pricing model with respect to its many parameters can be achieved with a single execution of adjoint AD.

Tape based

In operator overloading based implementations the graph is stored in a contiguous data structure called the *tape*. Consider again example 2.3.1. During tangent AD the presented graph is never explicitly build up as the propagation of the dual numbers is sufficient. However, if we want to start taking the derivatives at the output we must store the intermediate computations. This is where the tape kicks in: At runtime, the overloaded functions call the adjoint AD tool to store its intermediate variables in the tape. We thus first run through the entire function of interest, called the *primal section*. Then, we seed the output value for which we wish to find the derivatives with respect to all inputs to. Finally, the program flow is reversed and the local adjoints are computed in the *reversed section*, i.e. the backpropagation phase. To get $\frac{\partial y_0}{\partial x_0}$ in example 2.3.1, we must seed y_0 to 1 and y_1 to 0 after the primal section. Then the adjoint $x_{0_{(1)}}$ will store the result. In addition, with this seed we get the result of $\frac{\partial y_0}{\partial x_1}$ stored in $x_{0_{(1)}}$ without any extra computation. To find the entire Jacobian it thus remains to seed with $\boldsymbol{y} = \boldsymbol{e}_1$ and perform the computation once more. The computational cost of adjoint AD therefore scales with the size of the output. Since many domains have problems with many more inputs than outputs this method is highly desirable.
However, this comes at a cost of significantly increased memory consumption. In C++, expression templates are used at compile time to collapse intermediate computations that would need to be stored, resulting in decreased memory usage. This is not an option in interpreted languages where code generation is more common.

Code generation

AD by code generation takes the ideas presented in the tape based approach and performs them explicitly instead of relying on runtime data structure to capture the program execution. If code generation is to be performed at compile time, it requires a parser to capture the operations and control flow of the program. Once the structure is captured in an AST the corresponding derivative functions can be generated. It requires, of course, that the programmer specifies which functions need a corresponding derivative. Code generation can be done for tangent AD, but provides

little to no benefit over operator overloaded methods since no intermediate computations need to be stored. The code generator thus just has to insert the corresponding tangent code before the corresponding operation. The compiled code will therefore often be very similar for tangent AD. However, in adjoint AD intermediate computations that are required for the computation of the adjoint must be stored somehow. A common approach used in code generators like Enzyme or Tapenade [18] is to push the intermediate values onto a local stack.

$$\begin{aligned} v_0 &= x_0 \\ v_1 &= x_1 \\ v_2 &= v_0 \cdot v_1 \\ v_3 &= \sin(v_2) \\ v_4 &= \exp(v_0) \\ v_5 &= v_3 + v_1 \\ v_6 &= v_3 - v_4 \\ y_0 &= v_5 \\ y_1 &= v_6 \end{aligned}$$

(a) Primal section ↓

$$\begin{aligned} x_{0(1)} &+= v_{0(1)} \\ x_{1(1)} &+= v_{1(1)} \\ v_{0(1)} &+= a \cdot v_{2(1)} \\ v_{1(1)} &+= c \cdot v_{2(1)} \\ v_{2(1)} &+= e \cdot v_{3(1)} \\ v_{0(1)} &+= b \cdot v_{4(1)} \\ v_{3(1)} &+= g \cdot v_{5(1)} \\ v_{1(1)} &+= d \cdot v_{5(1)} \\ v_{3(1)} &+= f \cdot v_{6(1)} \\ v_{4(1)} &+= h \cdot v_{6(1)} \\ v_{5(1)} &= y_{0(1)} \\ v_{6(1)} &= y_{1(1)} \end{aligned}$$

(b) Reversed section ↑

$$a = \frac{\partial v_2}{\partial v_0} \quad b = \frac{\partial v_2}{\partial v_1} \quad c = \frac{\partial v_3}{\partial v_2} \quad d = \frac{\partial v_4}{\partial v_0} \quad e = \frac{\partial v_4}{\partial v_3} \quad f = \frac{\partial v_5}{\partial v_1} \quad g = \frac{\partial v_6}{\partial v_3} \quad h = \frac{\partial v_6}{\partial v_4}$$

(c) Local derivatives

Figure 2.6: Adjoint AD internals of Example 2.3.1. We assume all local adjoints $(\cdot)_{(1)}$ are initialized to 0.

The theory of efficient adjoint computation and the optimal balance between memory consumption and run time performance is a wide ranging subject. We will not consider it in more detail. The interested reader may look into [36] for a general discussion and [2] for a machine learning perspective on AD instead.

2.4 JAX

JAX [5] is an AD tool build in Python that uses both operator overloading and code generation techniques. Typical AD code generators, e.g., Tapenade [18], parse the code at compile time and generate the corresponding derivative code before executing the program. Instead, JAX builds at runtime the representation of the code via operator overloading facilities from Python. It then

uses a form of *Just-In-Time* (JIT) compilation to optimize the code during runtime and to generate the required derivative code. It allows JAX to target various accelerators like the GPU or TPU without any change in the user code. However, it requires the program to be written in a functional subset of Python[5], replacing many core control flow primitives with custom functions, e.g., `jax.scan`, `jax.while_loop`, `jax.cond`, `jax.vmap`. These primitives allow JAX to apply automatic vectorization throughout the code. Take for example `jax.vmap`: It replaces a typical for loop and is used to directly apply a function over a batch of data. Or `jax.scan`, which furthermore threads the intermediate values throughout the computation.

```
import numpy as np
import jax
import jax.numpy as jnp

arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
# NumPy
def sum_and_max(arr):
    total = 0
    max_values = []

    for subarr in arr:
        total += np.sum(subarr)
        max_values.append(np.max(subarr))

    return total, max_values

final, out = sum_and_max(arr)

# JAX
def scan_fn(carry, x):
    return carry + jnp.sum(x), jnp.max(x)

final, out = jax.scan(scan_fn, jnp.array(0), arr)
```

Figure 2.7: Examples of JAX specific control flow primitives.

Its efficient implementation of tensors based on a NumPy like interface, general purpose AD facilities, and its capabilities for auto-vectorization to various accelerators is the reason that we

[5] Regular Python code can still be used but results in significantly longer compile times since for loops are unrolled and individually optimized by XLA. On the other hand, `jax.scan` is compiled down to a single primitive call.

chose to implement the presented ideas in JAX. The JVP and VJP are fundamental primitives in JAX and can be called through `jax.jvp` and `jax.vjp`, respectively. We will encounter these operations throughout the remaining sections and will postpone the discussion to the point of first use (section 6.1).

However, JAX has its *sharp bits*. Besides the restriction to pure functional operations, the biggest limitation is the restricted use of dynamic arrays in JIT compiled contexts. This is in large part due to the limitations of XLA, the backend compiler used by JAX. For more details, we recommend the reader to look into the official documentation which includes many detailed explanations of its inner workings.

3 Models for Option Pricing

"The stock market is filled with individuals who know the price of everything, but the value of nothing."
— Philip Arthur Fisher, *Common Stocks and Uncommon Profits*

Practitioners in trading options have existed long before mathematical models of option prices had been invented. Already in the 17$^{\text{th}}$ century, during the first tulip markets in the Netherlands, it was common to deal with put and call options [22]. While the markets back then were ill-regulated, simple principles and heuristics like the put-call parity were already known to some [52]. It took, however, many decades to stop forms of *windhandel*[1] before the markets operated professionally. The uncertainty underlying the trades remained unknown for many centuries and is to this day a challenging problem, especially for more sophisticated pricing models. Using mathematical models allows one to gain more insight and to assess risk inherent in the trades.

In this chapter, we describe two common option pricing models that will act as the data generating distribution for learning the surrogate model. We start with one of the first models to ever be published on this subject by Bachelier in section 3.1, before presenting a stochastic volatility model of Heston [19] in section 3.2.

3.1 Bachelier

It was Louis Bachelier who in 1900 came up with a dynamical model that incorporated and could describe a stochastic process [1]. In short, it is described by:

$$\mathrm{d}S_t = \mu S_t dt + \sigma \, \mathrm{d}W_t, \tag{3.1}$$

where $t > 0$, μ is the constant drift for the interest rate, σ is the constant volatility, S_t the underlying asset price at time t, and $\mathrm{d}W_t$ describes a Wiener process, i.e. Brownian motion.

We can simplify this formula by considering the forward price.

Definition 3.1. (Forward Price) The forward price F_t is the discounted price, i.e. $F_t = S_t e^{r(T-t)}$,

[1] Trading on shares and options that one does not possess. It is dutch and literally translates to *wind trading*, used to describe the various forms of speculation that occurred during the tulip mania.

where r is the interest rate and time T the maturity.

Alternatively, when considering the T-forward measure $\mathcal{Q}_T$, the model simplifies to

$$\mathrm{d}F_t = \sigma\,\mathrm{d}W_t{}^{\mathcal{Q}_T}, \tag{3.2}$$

The forward measure results in the drift term to disappear from the SDE as it is incorporated in the probability measure of the modified Wiener process. From now on, we will implicitly assume that any equation dealing with a forward price is considering a forward measure as well.

Definition 3.2 (Wiener process)**.** A stochastic process $\mathbf{W} = \{W_t\}_{t\geq 0}$, on the interval $[0, T]$, is a Wiener process if it fulfills the following properties:

- $W_0 = 0$.
- $W_t - W_s$ and $W_v - W_u$ are independent for all $0 \leq s < t < u < v \leq T$.
- $W_{t+s} - W_s \sim N(0, t)$,
- With probability 1, W_t viewed as a function of t is continuous.

As in the introduction, we consider a European call option on the underlying asset with maturity at time T and strike price K.

Definition 3.3 (European call option payoff)**.** $\nu(S_T, K) = (S_T - K)^+$.

Definition 3.4 (Option price)**.** The option price is the expected value of the payoff:

$$V = \mathbb{E}[\nu(S_T, K)]$$

Let $V_C(F_t, K)$ denote the price of the European call option at time t for strike K. The call option price at time $t = 0$ can be calculated analytically and is:

$$V_C(F_0, K) = (F_0 - K)\Phi(z) + \sigma\sqrt{T}\varphi(z), \quad z = \frac{F_0 - K}{\sigma\sqrt{T}}, \tag{3.3}$$

where φ, Φ are the PDF and CDF of the standard normal distribution, respectively. We will use the analytic prices for comparison with the prices obtained by the surrogate model. For further derivations of the Bachelier model, including its characteristic function and PDE description see, e.g., [49].

The famous model by Black and Scholes is in fact very much related to the model of Bachelier, using a log-normal distribution instead of the normal distribution [45]. This change often makes

sense for options on stock prices since halving the price should be regarded with the same probability as doubling the price. Nonetheless, in many domains this property does not apply, e.g. interest rates move up or down 25 basis points, and the Bachelier model is suitable. Particularly in the regime of negative interest rates Bachelier can be applied without modification which is not the case for other models [8]. In addition, the volatility term of Bachelier can be considered in absolute change in F_t while Black and Scholes consider volatility with respect to relative changes in the price.

Definition 3.5 (Basket)**.** A Basket $\mathbf{S}_t \in \mathbb{R}^m$ of m securities $S_t^{[0]}, S_t^{[1]}, \ldots, S_t^{[m]}$ has at time t the price:

$$\mathbf{S}_t = \sum_{i=0}^{m} \omega_i S_t^{[i]}, \quad \sum_{i=0}^{m} \omega_i = 1,$$

where ω_i is the weight associated with the ith security.

Remark. *A Basket option can in many regards be treated like a single asset, making it applicable to all the defined option pricing models and payoffs. However, it requires care in realizing that the sum of the underlying (correlated) assets may not follow the distribution of the individual assets.*

Basket options are another domain where the Bachelier model is useful since even if the underlying asset prices are log-normal distributed, a weighted sum of such assets in general is not log-normal distributed. We will consider basket models with assets being jointly normal distributed. As a result, the price of the basket option will remain Gaussian and the Bachelier model computes suitable prices. In particular, consider a basket with correlated normally distributed assets that can be generated from a multivariate normal distribution. Then we can model the basket option with a correlated Bachelier model for m assets:

$$\mathrm{d}\mathbf{F}_t = \boldsymbol{\sigma}\, \mathrm{d}\mathbf{W}_t, \tag{3.4}$$

where $\mathbf{F}_t \in \mathbb{R}^m$ and $\mathrm{d}W_t^j\, \mathrm{d}W_t^k = \rho_{jk}$ with $j, k \in \{1, \ldots, m\}$. That is, the Wiener process $\mathrm{d}W_t^j$ is correlated to the process $\mathrm{d}W_t^k$ with constant ρ_{jk}. For $j = k$, the correlation is 1. Each asset j has a volatility σ_j. So, the constant volatilities $\boldsymbol{\sigma}$ are applied elementwise.

As pointed out by Huge and Savine [21], although we deal with m assets, the basket option price will turn out to be a nonlinear function of a single dimension. It requires a surrogate model to perform large dimensionality reduction to uncover and correctly represent this pricing function.

The Greeks can be found through differentiation via AD or analytically. As an example, we analytically derive the Delta of the call option price:

$$\begin{aligned}
\frac{\partial V_C(F_0, K)}{\partial F_0} &= \frac{\partial}{\partial F_0}\Big[(F_0 - K)\Phi(z) + \sigma\sqrt{T}\varphi(z)\Big] \\
&= \Phi(z)\frac{\partial}{\partial F_0}(F_0 - K) + (F_0 - K)\frac{\partial}{\partial F_0}\Phi(z) + \sigma\sqrt{T}\frac{\partial}{\partial F_0}\varphi(z) \\
&= \Phi(z) + \frac{(F_0 - K)\varphi(z)}{\sqrt{2\pi}\sigma\sqrt{T}} - \frac{1}{\sqrt{2\pi}}\varphi(z)\frac{F_0 - K}{\sigma\sqrt{T}} \\
&= \Phi(z), \quad z = \frac{F_0 - K}{\sigma\sqrt{T}}.
\end{aligned} \tag{3.5}$$

3.2 Heston

The Bachelier model considers only static volatility. However, during different market regimes, especially in exceptional scenarios, the volatility can vary drastically. Thus, Heston [19] suggests to extend the SDE with another stochastic process for the volatility. We get:

$$\begin{aligned}
\mathrm{d}S_t &= \mu S_t \,\mathrm{d}t + \sqrt{\nu_t} S_t \,\mathrm{d}W_t{}^S, \\
\mathrm{d}\nu_t &= \kappa(\theta - \nu_t)\,\mathrm{d}t + \xi\sqrt{\nu_t}\,\mathrm{d}W_t{}^\nu,
\end{aligned} \tag{3.6}$$

where μ is the drift of the underlying asset, ν the instantaneous variance, θ the long run average variance of the price, κ the rate of mean reversion of ν_t to θ, and ξ the volatility of the volatility. The Wiener process W_t^S is correlated to Wiener process W_t^ν with ρ, i.e. $\mathbb{E}[\mathrm{d}W_t{}^S \,\mathrm{d}W_t{}^\nu] = \rho\,\mathrm{d}t$. While the Heston model has in general no analytical solution, for the special case of the European (call) option with fixed parameters it can be obtained via its characteristic function. The Greeks can again be computed through AD.

To avoid that the price becomes negative during the simulation, we can consider logarithmic asset prices. By the application of Itô's lemma, the adapted price dynamics are:

$$\mathrm{d}(\ln S_t) = (\mu - \frac{1}{2}\nu_t)\,\mathrm{d}t + \sqrt{\nu_t}\,\mathrm{d}W_t{}^S. \tag{3.7}$$

A derivation of Itô's lemma is beyond the scope of this thesis but can be found in any mathematical finance textbook, e.g [16].

3.3 Implementation

To implement the model, a discretization method must be applied. Common methods include the Euler-Maryuama or the Milstein scheme. We will focus on the former in subsection 3.3.1,

including discretizations for Bachelier and Heston. Moreover, as we cannot exhaustively sample the payoffs to obtain the option price, we introduce the Monte Carlo method for estimating the price in subsection 3.3.2.

3.3.1 Euler-Maryuama

Consider again the general SDE:

$$\mathrm{d}S_t = a(S,t)\,\mathrm{d}t + b(S,t)\,\mathrm{d}W_t,$$

where W_t is a Wiener process and we want to solve it for some time interval $[0,T]$. Taking the integral on both sides from time t to some small time increment Δt, we have:

$$\int_t^{t+\Delta t} \mathrm{d}S_\tau = \int_t^{t+\Delta t} a(S_\tau,\tau)\ \mathrm{d}\tau + \int_t^{t+\Delta t} b(S_\tau,\tau)\ \mathrm{d}W_\tau.$$

Now we can approximate the integrals on the right hand side with a basic left endpoint rectangle approximation. For the first term:

$$\int_t^{t+\Delta t} a(S_\tau,\tau)\ \mathrm{d}\tau \approx a(S_t,t)\int_t^{t+\Delta t} \mathrm{d}\tau = a(S_t,t)\Delta t.$$

And for the second term:

$$\int_t^{t+\Delta t} b(S_\tau,\tau)\ \mathrm{d}W_\tau \approx b(S_t,t)\int_t^{t+\Delta t} \mathrm{d}W_\tau = b(S_t,t)\Delta W_t,$$

where $\Delta W_t = W_{t+\Delta t} - W_t$.

As a result, we obtain the following approximation to the above SDE:

$$\hat{S}_{t+\Delta t} - \hat{S}_t = a(\hat{S}_t,t)\Delta t + b(\hat{S}_t,t)\Delta W_t$$

were $\hat{S}_t$ denotes an approximate time-discretized price. If, instead of the actual time from $[0,T]$, we consider an indexing scheme with $\Delta t = \frac{T}{n}$ and $i \in \mathbb{N}$ such that $0 \leq i < n$, we get the common definition of Euler-Maryuama.

Definition 3.6. The Euler-Maryuama discretization for the above SDE on time interval $[0,T]$ is given by:

$$\hat{S}_{i+1} = \hat{S}_i + a(\hat{S}_i,t_i)\Delta t + b(\hat{S}_i,t_i)\Delta W_i, \tag{3.8}$$

where $\Delta W_i = W_{t_{i+1}} - W_{t_i}$, $\Delta t = \frac{T}{n}$, $i \in \mathbb{N}$ s.t. $0 \leq i < n$, and n the number of time steps.

Remark. *By Definition 3.2, ΔW_i is normally distributed and independent of i. Therefore,*

$$\Delta W_i \sim N(0, t_{i+1} - t_i) = N(0, \Delta t) = \sqrt{\Delta t} N(0, 1).$$

This approach is thus reminiscent of the Euler method for discretizing ODEs, except that it also takes a random value every time step to increment the dynamics.

Bachelier

The Euler-Maryuama discretization of the Bachelier model is is found through direct application of Definition 3.6 and given by:

$$\hat{S}_{i+1} = \hat{S}_i + \sigma \Delta W_i, \tag{3.9}$$

where σ is the constant volatility and $\Delta W_i \sim N(0, 1)\sqrt{\Delta t}$ as before.

For a basket option, we can simply apply this discretization for each asset. Note, however, that it is required to keep the Wiener process samples correlated. This is achieved in the same manner as for the Heston model.

Heston

The Euler-Maryuama discretization of the Heston model in Equation 3.6 is given by:

$$\hat{\nu}_{i+1} = \hat{\nu}_i + \kappa(\theta - \hat{\nu}_i^+)\Delta t + \xi \sqrt{\hat{\nu}_i^+} \Delta W_i^\nu \tag{3.10}$$

$$\hat{S}_{i+1} = \hat{S}_i + \mu \hat{S}_t \Delta t + \sqrt{\hat{\nu}_i^+} \hat{S}_i \Delta W_i^S, \tag{3.11}$$

Considering log-based prices, we get:

$$\hat{\nu}_{i+1} = \hat{\nu}_i + \kappa(\theta - \hat{\nu}_i^+)\Delta t + \xi \sqrt{\hat{\nu}_i^+} \Delta W_i^\nu \tag{3.12}$$

$$\hat{S}_{i+1} = \hat{S}_i \exp((\mu - \frac{1}{2}\hat{\nu_i}^+)\Delta t + \sqrt{\hat{\nu_i}^+} \Delta W_i^S), \tag{3.13}$$

where $\hat{\nu}_i^+ = \max(0, \hat{\nu}_i)$ is using full truncation of $\hat{\nu}_i$ to avoid that the volatility becomes negative. Even if in theory of the continuous case the Feller condition $2\kappa\theta > \xi^2$ guarantees positive volatilises, this is not ensured once we discretize. Note, however, that this introduces a discontinuity that must be considered when differentiating since $\lim_{\hat{\nu}_i \to 0^+} \frac{\partial}{\partial \hat{\nu}_i} \sqrt{\hat{\nu}_i} = \infty$. We discuss solutions to this problem in chapter 4. To generate the random samples of the Wiener processes

with correlation ρ we can draw samples from a multivariate normal distribution with covariance matrix $\boldsymbol{\Sigma} = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$ and mean $\boldsymbol{\mu} = \mathbf{0}$.

A multivariate normal distribution can be implemented by sampling from the uncorrelated standard normal distributions and using the Cholesky decomposition.

Algorithm 3.1 Multivariate Normal Distribution

Require: Covariance matrix $\boldsymbol{\Sigma}$.
$(\boldsymbol{L}, \boldsymbol{L}^T) \leftarrow \text{Chol}(\boldsymbol{\Sigma})$ ▷ By Cholesky factorization
$\mathbf{Z} \sim N(0, 1)$ ▷ Draw i.i.d. standard normal samples
$\mathbf{X} \leftarrow \boldsymbol{L}\boldsymbol{Z}$ ▷ Correlated samples
return X

This approach is always well defined as the covariance matrix is symmetric and positive definite. Furthermore, the samples in $\mathbf{X}$ follow the covariance matrix $\boldsymbol{\Sigma}$, as:

$$\begin{aligned}
\mathbb{E}[\mathbf{X}\mathbf{X}^T] &= \mathbb{E}[(\boldsymbol{L}\boldsymbol{Z})(\boldsymbol{L}\boldsymbol{Z})^T] \\
&= \mathbb{E}[\boldsymbol{L}\boldsymbol{Z}\boldsymbol{Z}^T\boldsymbol{L}^T] \\
&= \boldsymbol{L}\mathbb{E}[\mathbf{Z}\mathbf{Z}^T]\boldsymbol{L}^T \quad \text{(by linearity of Expectation)} \\
&= \boldsymbol{L}\mathbf{I}\boldsymbol{L}^T = \boldsymbol{L}\boldsymbol{L}^T = \boldsymbol{\Sigma}.
\end{aligned}$$

3.3.2 Monte Carlo

For more exotic options or custom models the analytic solution will not be available and we must resort to simulation using Monte Carlo (MC) methods. For example, the Heston model does not have an analytical solution for any path-dependent payoff. The option price is found through simulating many paths of the discretized SDE and averaging over the (discounted) payoffs.

Definition 3.7 (Unbiased Estimate)**.** An unbiased estimate will converge in expectation to the true value. That is, given estimator $\hat{F}$ for true value $F = \mathbb{E}_{\boldsymbol{x}\sim\mathcal{X}}[f(\boldsymbol{x})]$, it is unbiased if:

$$\mathbb{E}[\hat{F}] = F.$$

The MC estimator for expectation F is unbiased:

$$\mathbb{E}[\hat{F}] = \mathbb{E}_{\hat{\boldsymbol{x}}_i \sim \mathcal{X}}\Big[\frac{1}{m}\sum_{i=1}^{m} f(\hat{\boldsymbol{x}}_i)\Big] \tag{3.14}$$

$$= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}_{\hat{\boldsymbol{x}}_i \sim \mathcal{X}}[f(\hat{\boldsymbol{x}}_i)] \quad \text{(by linearity of Expectation)} \tag{3.15}$$

$$= \mathbb{E}[f(\boldsymbol{x})] = F. \tag{3.16}$$

In options pricing, we have an equivalent problem of estimating, e.g., the call option price V_C through a MC estimator:

$$\mathbb{E}[\hat{V}_C] = \mathbb{E}\Big[\frac{1}{m}\sum_{i=1}^{m} \nu(\hat{S}_T^{(i)}, K)\Big] = \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}[\nu(\hat{S}_T^{(i)}, K)] = \mathbb{E}[\nu(S_T, K)] = V_C. \tag{3.17}$$

We can now compute unbiased estimates of option prices. It remains to show how to efficiently compute the Greeks, i.e. the gradient of the price with respect to the parameters.

4 Pathwise Sensitivities

In this section, we present the way we will sample sensitivity information from the option pricing models presented in chapter 3. Traditionally, computing the Greeks through tangent AD or finite differences requires, for each parameter, sampling many paths, taking the average over them with some discounting applied to come up with an approximation of the option price and the Greeks. Multiple invocations of the entire Monte Carlo simulation is often rather expensive. The adjoint AD method can be used to find the gradient with respect to all the input parameters at once. However, it requires considering all paths at once, leading to severe memory requirements. The memory requirements scale with the number of paths and the number of time steps (to store the intermediate results that are required for the adjoint computation). Instead, what if we could consider computing the gradient for every path separately and approximate the Greeks by averaging over these *pathwise sensitivities*? This concept is known by many names, depending on the context. The pathwise gradient estimator, process derivative [37] in optimization, and in finance, the pathwise derivative [6], [16, pp. 386–388] or smoking adjoints [15]. In ML, the technique was popularized through the work of Kingma and Welling [27] on Variational Autoencoders and is referred to by the *reparameterisation trick*. Besides its potential computational savings, it gives an unbiased estimate with overall lower variance compared to other methods. However, it is not always applicable.

We first describe the concept in general in section 4.1, before stating when it is applicable in section 4.2. In particular, we will see that one big disadvantage of the pathwise derivative method is that it does not support discontinuous payoff functions by default. Since these payoffs are a cornerstone of the option pricing models, we discuss smoothing techniques for discontinuous payoffs in section 4.3.

4.1 Interchanging the derivative and expectation

In order to take the derivative of the payoff function ν, we first consider the realization of the random path as an explicit parameter $z \sim \mathcal{Z}$, where the set $\mathcal{Z}$ represents the possible random

vectors. The payoff can be decomposed into a function on f, where $f : \Theta_{\text{in}} \times \mathcal{Z} \to \Theta_{\text{out}}$ represents the underlying path:

$$\nu(g(\boldsymbol{\theta}, \boldsymbol{z})),$$

where the input parameters $\boldsymbol{\theta} \in \Theta_{\text{in}}$. In the case of Bachelier, $\boldsymbol{\theta} = (S_0, K)$, $f(\boldsymbol{\theta}) = S_T - K$ and ν is $(\cdot)^+$. E.g., $S_0 \sim U(90, 110)$ and $K = 100$. Alternatively, we can also think of f returning multiple values that correspond to the parameters of the payoff function in Definition 3.3.

We have unbiased estimates of pathwise derivatives of the payoff, if:

$$\mathbb{E}_{\boldsymbol{z}\sim\mathcal{Z}}\left[\frac{\partial}{\partial S_0}\nu(g(\boldsymbol{\theta}, \boldsymbol{z}))\right] = \frac{\partial}{\partial S_0}\mathbb{E}_{\boldsymbol{z}\sim\mathcal{Z}}\left[\nu(g(\boldsymbol{\theta}, \boldsymbol{z}))\right] \tag{4.1}$$

If we can interchange the derivative with the expectation as above, it is possible to compute the derivative for each path individually. We will discuss the applicability of the method in section 4.2. The derivative of a path can further be broken down using the chain rule:

$$\frac{\partial}{\partial S_0}\nu(g(\boldsymbol{\theta}, \boldsymbol{z})) = \frac{\partial \nu(g(\boldsymbol{\theta}, \boldsymbol{z}))}{\partial S_T}\frac{\partial S_T}{\partial S_0} \tag{4.2}$$

We will be using adjoint AD to compute Equation 4.2 automatically. As an example, we provide the analytic pathwise derivative for the Bachelier model of European call options.

Bachelier

We consider again the Bachelier model. We get for a fixed $\boldsymbol{z} \sim \mathcal{Z}$ and $\boldsymbol{\theta} \in \Theta_{\text{in}}$:

$$\frac{\partial \nu(g(\boldsymbol{\theta}, \boldsymbol{z}))}{\partial F_T} = \frac{\partial}{\partial F_T}(F_T - K)^+ = \mathbb{1}_{F_T > K}. \tag{4.3}$$

Note that at $F_T = K$ the derivative does not exist, but the event $F_T = K$ occurs with probability 0. As a result, the payoff function is almost surely differentiable with respect to F_T.

Furthermore,

$$F_T = F_0 + \int_0^t \sigma \, \mathrm{d}W_t, \quad 0 \le t \le T, \quad \text{(by definition of SDE)}$$

So,

$$\frac{\partial F_T}{\partial F_0} = 1.$$

Overall the pathwise derivative of the payoff under the Bachelier model is just $\mathbb{1}_{S_T > K}$.

For the basket option, the pathwise derivative payoff for the individual dimensions remains the same since $\frac{\partial F_t^{(i)}}{\partial F_0^{(j)}} = 0$, for all $i \neq j$ and $\frac{\partial F_t^{(i)}}{\partial F_0^{(i)}} = 1$.

4.2 Applicability

The method of pathwise derivatives is only applicable if certain conditions can be fulfilled. Glasserman [16, pp. 393–395] discusses practical sufficient conditions to verify the validity of the pathwise method. In practice, the deciding criterion is whether the (discontinuous) payoff function ν is Lipschitz continuous with respect to the parameters, and differentiable almost everywhere.

Definition 4.1. The payoff function ν is Lipschitz continuous, if there exists a real constant $k_\nu \geq 0$ such that for all $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \Theta_{\text{in}}$ and $\boldsymbol{z} \sim \mathcal{Z}$

$$\|\nu(g(\boldsymbol{\theta}_2, \boldsymbol{z})) - \nu(g(\boldsymbol{\theta}_1, \boldsymbol{z}))\| \leq k_\nu \|\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1\|, \tag{4.4}$$

i.e. it adheres to the Lipschitz continuity condition.

If ν is a smooth function of f it is sufficient to consider whether f is Lipschitz continuous. However, we almost always deal with non-smooth payoff functions like in the next example.

Example 4.2.1. Consider the European call option payoff. We already discussed at Equation 4.3 that the payoff has only one non-differentiable point at $S_T = K$ and is thus almost everywhere differentiable. Furthermore, the payoff is Lipschitz continuous because the $(\cdot)^+$ function is Lipschitz continuous:

Let $y_1, y_2 \in \Theta_{\text{out}}$ represent the output of two different paths and assume w.l.o.g. that $y_1 \geq y_2$,

$$\|\nu(y_1) - \nu(y_2)\| = \|(y_1)^+ - (y_2)^+\| = \left\{ \begin{array}{ll} \|y_1 - y_2\|, & \text{if } y_1 > 0, y_2 > 0 \\ \|y_1\|, & \text{if } y_1 > 0, y_2 < 0 \\ 0, & \text{if } y_1 < 0, y_2 < 0 \end{array} \right\} \leq \|y_1 - y_2\|.$$

Since we are, in addition, interested in second-order pathwise derivative information, we further require ν to be twice differentiable almost everywhere and that Definition 4.1 holds for $\nu^{(1)}$. To be precise, there exists a real constant $k_{\nu^{(1)}}$ such that for all $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \Theta_{\text{in}}$ and $z \sim \mathcal{Z}$

$$\|\nu^{(1)}(f(\boldsymbol{\theta}_2, \boldsymbol{z})) - \nu^{(1)}(f(\boldsymbol{\theta}_1, \boldsymbol{z}))\| \leq k_{\nu^{(1)}} \|\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1\|. \tag{4.5}$$

These conditions can be generalized to higher-order pathwise derivatives by requiring ν to be n times differentiable almost everywhere and taking the $(n-1)^{\text{th}}$ derivative of f in Equation 4.5.

Example 4.2.2. Again, the European call option payoff, but now considering applicability for second-order pathwise derivatives. We know that $\nu^{(1)}$ is a Heaviside step function which is clearly not Lipschitz continuous as it is not even continuous. Therefore, Equation 4.1 does not hold. Without modification, second-order pathwise derivatives are thus not applicable. This turns out

to be almost always the case in options pricing since the European payoff is amongst the simplest payoffs to be considered. Note that without the Lipschitz condition, we have that $\nu^{(1)}$ is differentiable everywhere except at the strike and thus almost everywhere differentiable. However, the value of the derivative is always 0 whenever it does exist. Almost everywhere differentiable payoff functions are thus not sufficient. The nature of the Dirac delta is not captured when considering the pathwise method.

The last example is motivation to consider techniques to make the second-order pathwise derivatives applicable by modifying the payoff functions to be well-behaved. A technique that is often used in this context is smoothing.

4.3 Smoothing payoff functions

To alleviate the problem of discontinuous payoff functions we consider smoothing the payoff function. Smoothing functions is frequently done in quantitative finance, however, one must take care to not introduce large bias into the computation. Many smoothing functions have been proposed but we consider only sigmoidal smoothing and cubic spline functions. In addition, for the $(\cdot)^+$ function we could consider one of the differentiable ReLU alternatives described in subsection 2.1.2. Here we briefly describe and visualize the other two methods.

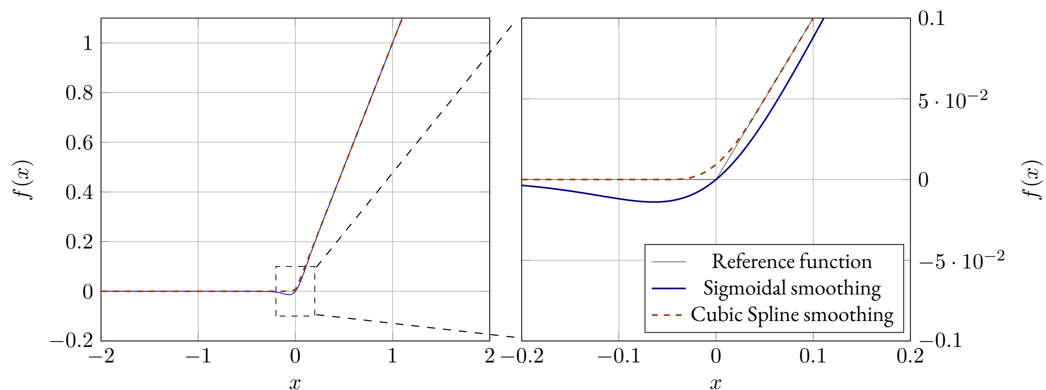


Figure 4.1: Smoothing functions for payoff $(\cdot)^+$, where smoothing width $w = 0.05$.

Sigmoidal Smoothing

A general approach for smoothing the discontinuous transition between two functions at position p can be achieved through the use of sigmoidal smoothing. We perform smoothing between function $f_1 : \mathbb{R} \to \mathbb{R}$ and $f_2 : \mathbb{R} \to \mathbb{R}$ via $\tilde{f} : \mathbb{R} \times \mathbb{R}^2 \to \mathbb{R}$ defined as

$$\tilde{f}(x,p,w) = (1 - \sigma(x,p,w))f_1(x) + \sigma(x,p,w)f_2(x), \quad \sigma(x,p,w) = \frac{1}{1 + e^{-\frac{x-p}{w}}}$$

where p is the position to change between the two functions and w the width of the smoothing. For $\nu = (\cdot)^+$, we can first split up the function into $\begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$. By sigmoidal smoothing we thus get:

$$\tilde{\nu}(x,w) = \frac{x}{1 + e^{-\frac{x}{w}}}.$$

Cubic Spline Smoothing

We can consider a cubic spline approximation of the $(\cdot)^+$ function as suggested in [23] for use in the Heston model.

$$\tilde{\nu}(x,w) = \begin{cases} 0, & x < -w \\ -\frac{1}{16w^3}x^4 + \frac{3}{8w}x^2 + \frac{1}{2}x + \frac{3w}{16}, & -w \leq x \leq w\,, \\ x, & x > w \end{cases} \tag{4.6}$$

where w is the width of the smoothing.

A visual comparison of the two methods is given in Figure 4.1. The cubic spline method more closely follows the function, however, the sigmoidal smoothing has the benefit that at $x = 0$ the smoothed function is 0, just like the true payoff function. The cubic spline method should be preferred when smoothing the truncated volatility in the Heston model as the volatility should never be negative.

5 Differential Machine Learning

"Data beats algorithms, but better data beats more data."

— Peter Norvig

Traditional neural network based learning of surrogates considers a dataset of values generated from the function representing the model we wish to approximate. However, the reference model often encodes more information than just output values given some input values. In particular, derivative information can, in many cases, easily be obtained but is often completely neglected.

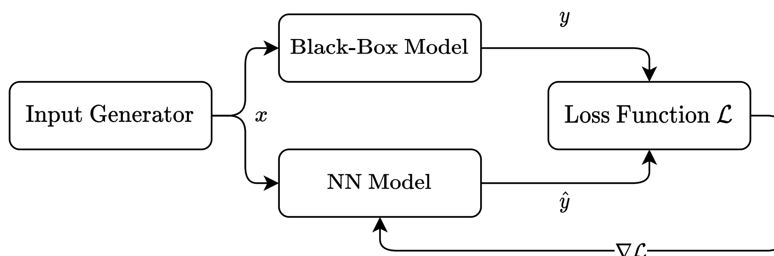


Figure 5.1: Learning neural network based surrogate models.

One method that does not neglect derivatives is Differential Machine Learning (DML). In the field of quantitative finance, the concept of DML was first introduced by Huge and Savine [21]. The core idea boils down to using derivative information during training of a neural network. In options pricing, this added data corresponds to pathwise derivatives of the payoff as discussed in chapter 4. It thus provides information of the Greeks during the learning process. Considering derivative information is, however, not new and has been applied in various other fields.

Model distillation of neural networks represents another domain in which derivatives are easily found, yet often not used. After all, the training process of the initial neural network had to use backpropagation to perform learning. This information could also be used when learning a smaller neural network that should distill the knowledge of the larger one. In this domain, the idea is referred to by Sobolev Training [10]. The name is motivated by the mathematical foundation of Sobolev spaces, which incorporate derivatives into the norm of a vector space. Since the ideas are more commonly known by Differential Machine Learning in options pricing, we will stick to it throughout this thesis.

In this chapter, we present DML for general neural-network based surrogate models. We explain how many option prices can be computed through random payoff samples in section 5.1. In section 5.2 we, moreover, consider pathwise derivatives during learning and describe the algorithm. Finally, the need for reducing the variance of the sampling process becomes apparent when training in this pathwise regime. We briefly describe applicable methods for variance reduction in section 5.3.

5.1 Option prices from payoff samples

Computing an option price for each configuration of the initial parameters by sampling many payoffs and averaging over them via MC, as described in subsection 3.3.2, is often too costly to perform in production environments. Instead of throwing away the sampled payoffs after finding the option price for some initial spot price S_0, could we instead reuse the prior generated information for predicting the price given other S_0 and hence save a large fraction of the computational cost?

Consider a scenario in which the option prices for the initial spot prices $S_0 \in \{100, 101, 103\}$ have been computed. The price for $S_0 = 102$ might already be sufficiently constrained that it can be found through pure curve fitting of the already computed payoff samples of the other option price computations. Instead of creating many paths to find the option price given a single initial state, split up the paths to start with randomly sampled initial states given some input parameter range. Then regression is applied to fit a curve that best describes the option prices given the entire parameter range. Figure 5.2 visualizes this method for the toy example. Through the regression we can price an option given $S_0 = 102$ even if no samples explicitly were generated for this input. The major benefit of this approach is thus that learning to predict many prices is achieved at the cost of just a single large MC sampling across the entire domain of the input parameters. One simple approach for regression first used in the more difficult setting of American option pricing and outlined by Longstaff and Schwartz [32], is to apply least-squares regression to fit the curve. As a result, the method is known as Least-Squares Monte Carlo and can generically be described by:

$$\boldsymbol{\vartheta}^* = \arg\min_{\boldsymbol{\vartheta}} \mathbb{E}_{(\boldsymbol{\theta},\boldsymbol{z})\sim\Theta_{\text{in}}\times\mathcal{Z}}\left[\|\nu(f(\boldsymbol{\theta},\boldsymbol{z})) - f_{\boldsymbol{\vartheta}}(\boldsymbol{\theta})\|_2^2\right], \tag{5.1}$$

where $f_{\boldsymbol{\vartheta}}$ is the fitted curve with coefficients $\boldsymbol{\vartheta}$ for random input parameters $\boldsymbol{\theta} \sim \Theta_{\text{in}}$ and random path noise samples $\boldsymbol{z} \sim \mathcal{Z}$.

Example 5.1.1 (Least squares of quadratic candidate function)**.** Consider fitting a quadratic function $\alpha x^2 + \beta x + \gamma$ to the grey points plotted in Figure 5.2. Thus, $\boldsymbol{\vartheta} = (\alpha, \beta, \gamma)$ and

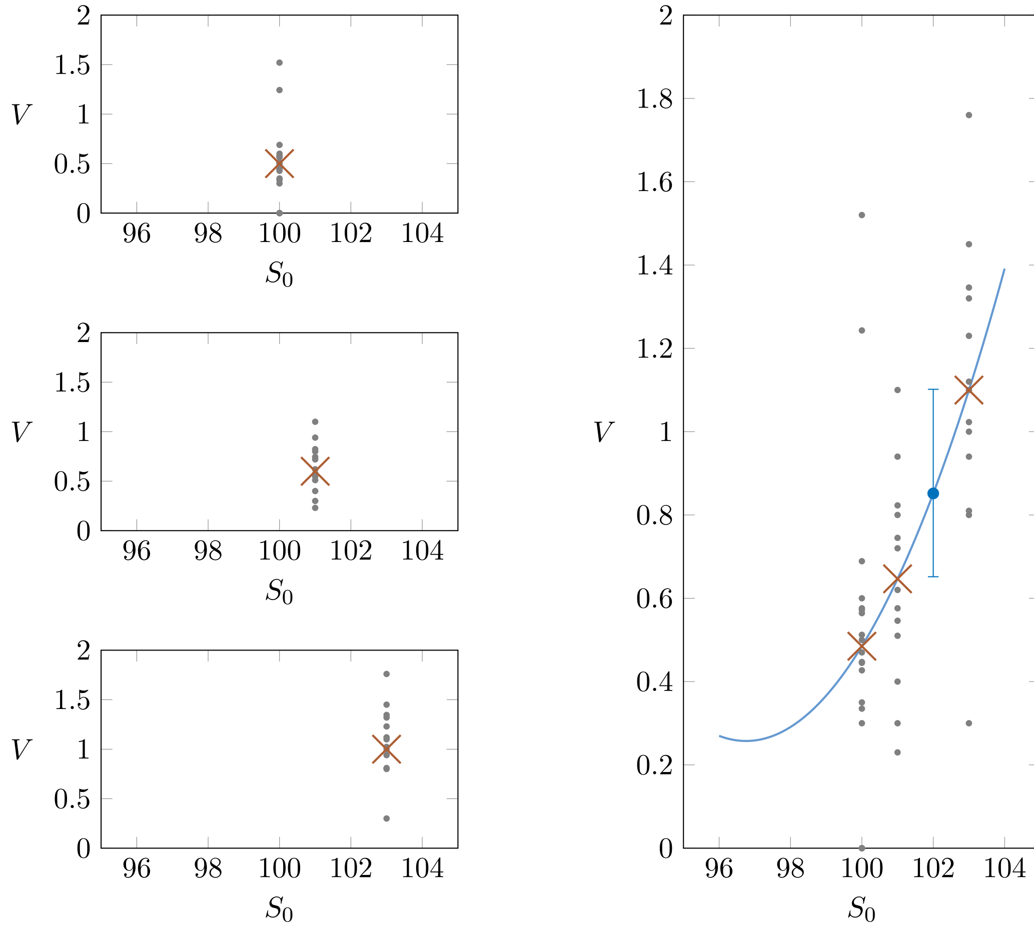


Figure 5.2: Individual Monte Carlo pricing (left) vs Least-Squares Monte Carlo (right).

we want to find $\boldsymbol{\vartheta}^*$ which best fits those points. Performing the optimization results in the blue regression line of the figure.

Equation 5.1 is very reminiscent of Equation 2.1 when we described optimization using neural networks. In fact, the method is not limited to least-squares regression and we could instead consider a neural network to perform the regression. By the universal approximation theorem of neural networks, we can converge to any function if the neural network grows infinitely wide or deep, as described in subsection 2.1.3. As a result, we can fit the neural network through gradient based optimization. Furthermore, the following derivation shows that the learned function $f_{\boldsymbol{\vartheta}}$ indeed converges to the expectation $\mathbb{E}[y|\boldsymbol{x}]$ we seek to approximate. Here $y = f(\boldsymbol{x}) = \nu(g(\boldsymbol{x}))$ and $\boldsymbol{x} = (\boldsymbol{\theta}, \boldsymbol{z})$ are used for convenience.

$$\mathcal{R}_{\mathcal{D}}(\boldsymbol{\vartheta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\Big[\mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}), y)\Big] \tag{5.2}$$

$$= \iint \mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}), y) p(\boldsymbol{x}, y)\, d\boldsymbol{x}\, dy \tag{5.3}$$

$$= \iint (f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) - y)^2 p(\boldsymbol{x}, y)\, d\boldsymbol{x}\, dy \tag{5.4}$$

With the last equation explicitly setting the loss to the squared-error loss. We minimize the risk by setting its gradient to 0.

$$2 \int (f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) - y) p(\boldsymbol{x}, y)\, dy = 0 \tag{5.5}$$

So,

$$\int f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) p(\boldsymbol{x}, y)\, dy = \int y p(\boldsymbol{x}, y)\, dy \tag{5.6}$$

$$f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) \int p(\boldsymbol{x}, y)\, dy = \int y p(\boldsymbol{x}, y)\, dy \tag{5.7}$$

By the law of total probability,

$$f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) p(\boldsymbol{x}) = \int y p(\boldsymbol{x}, y)\, dy \tag{5.8}$$

$$f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) = \int y \frac{p(\boldsymbol{x}, y)}{p(\boldsymbol{x})}\, dy \tag{5.9}$$

By Bayes' rule,

$$f_{\boldsymbol{\vartheta}}(\boldsymbol{x}) = \int y p(y|\boldsymbol{x})\, dy = \mathbb{E}[y|\boldsymbol{x}]. \tag{5.10}$$

Under the squared error loss, the optimal regression function converges to the mean prediction. This is a well-known result in ML [4]. However, since we only consider $\mathcal{S}$ and not $\mathcal{D}$ during training, the squared error loss of the payoff values can result in overfitting to the training samples as it heavily weighs outliers.

5.2 Learning with pathwise derivatives

Typically in ML, the problem of overfitting is addressed through some form of regularization. The Bayesian perspective highlights that we can view least-squares regression as Maximum-Likelihood estimation under the assumption of Gaussian noise being present in the output. We are interested in the posterior:

$$p(\boldsymbol{\vartheta}|\boldsymbol{x},\boldsymbol{y},\sigma^2) \propto p(\boldsymbol{\vartheta},\boldsymbol{x},\boldsymbol{y},\sigma^2) \tag{5.11}$$

$$\propto p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\vartheta},\sigma^2)p(\boldsymbol{\vartheta}) \tag{5.12}$$

If we do not consider any prior information for the parameters $\boldsymbol{\vartheta}$, i.e. sampling from the uniform distribution, the optimal $\boldsymbol{\vartheta}$ can thus be found through maximizing the likelihood. An equivalent optimization scheme is to minimize the negative log-likelihood [35, Ch. 7.3]:

$$-\log p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\vartheta},\sigma^2) = -\log\Big[\prod_{i=1}^{m} N(y_i|f_{\boldsymbol{\vartheta}}(x_i),\sigma^2)\Big] \tag{5.13}$$

$$= -\sum_{i=1}^{m}\log\Big[\frac{1}{\sqrt{2\pi\sigma^2}}\exp\Big(-\frac{1}{2\sigma^2}(y_i - f_{\boldsymbol{\vartheta}}(x_i))^2\Big)\Big] \tag{5.14}$$

$$= -\sum_{i=1}^{m}\log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{1}{2\sigma^2}(y_i - f_{\boldsymbol{\vartheta}}(x_i))^2 \tag{5.15}$$

$$= \sum_{i=1}^{m}\log(\sqrt{2\pi\sigma^2}) + \frac{1}{2\sigma^2}\sum_{i=1}^{m}(y_i - f_{\boldsymbol{\vartheta}}(x_i))^2 \tag{5.16}$$

$$\propto \|\boldsymbol{y} - f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\|_2^2 \tag{5.17}$$

Which indeed results in the sum of squared errors. If we furthermore consider a Gaussian prior on the weights and perform Maximum-a-Posteriori estimation we find that this corresponds to regression using L_2 regularization [4, Ch. 3.3]. However, such regularization adds additional hyperparameters that introduce further bias-variance considerations leading to the well known bias-variance tradeoff. Instead, consider a modified posterior which depends on the gradient information $\nabla_{\boldsymbol{x}}\boldsymbol{y}$.

$$p(\boldsymbol{\vartheta}|\boldsymbol{x},\boldsymbol{y},\nabla_{\boldsymbol{x}}\boldsymbol{y},\sigma^2) \propto p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\vartheta},\sigma^2)p(\nabla_{\boldsymbol{x}}\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\vartheta},\sigma^2) \tag{5.18}$$

Then the negative log-likelihood results in the differential loss.

Definition 5.1 (Differential Loss)**.** Given input $\boldsymbol{x}$, target $\boldsymbol{y}$, predicted output $f_{\boldsymbol{\vartheta}}(\boldsymbol{x})$, differential target $\nabla_{\boldsymbol{x}}\boldsymbol{y}$, and predicted differential $\nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}}(\boldsymbol{x})$, the differential loss is defined by:

$$\|\boldsymbol{y} - f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\|_2^2 + \lambda\|\nabla_{\boldsymbol{x}}\boldsymbol{y} - \nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\|_2^2, \tag{5.19}$$

where $\lambda \in \mathbb{R}_{\geq 0}$ is an added balancing factor.

The differential loss thus naturally arises when considering that the parameters $\boldsymbol{\vartheta}$ depend on the target outputs $\boldsymbol{y}$ and differential target outputs $\nabla \boldsymbol{y}$ with the assumption of Gaussian noise on both. This assumption indeed holds for the noise generation used in the option pricing models of Bachelier and Heston (see chapter 3).

In Sobolev Training the Equation 5.19 is also known as the Sobolev Loss. In this context, Srinivas and Fleuret [47] come to the same conclusion by considering Gaussian noise perturbations $\boldsymbol{\epsilon}$ to the input $\boldsymbol{x}$ and prove the following via the use of a Taylor expansion.

$$\begin{aligned} \mathbb{E}_{\boldsymbol{\epsilon} \sim N(0,\sigma^2)}\left[\sum_{i=1}^{m}(f(\boldsymbol{x}_i+\boldsymbol{\epsilon}) - f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i+\boldsymbol{\epsilon}))^2\right] &= \sum_{i=1}^{m}(f(\boldsymbol{x}_i) - f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i))^2 \\ &+ \sigma^2 \sum_{i=1}^{m} \|\nabla_{\boldsymbol{x}} f(\boldsymbol{x}_i) - \nabla_{\boldsymbol{x}} f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i)\|_2^2 \\ &+ \mathcal{O}(\sigma^4), \end{aligned}$$

where $f(\boldsymbol{x}) = \boldsymbol{y}$ is the target output function. The model thus becomes more robust to Gaussian noise when considering differential data. In addition, it has been shown that the universal approximation characteristic still holds (with the usual caveats) under this adapted loss [10].

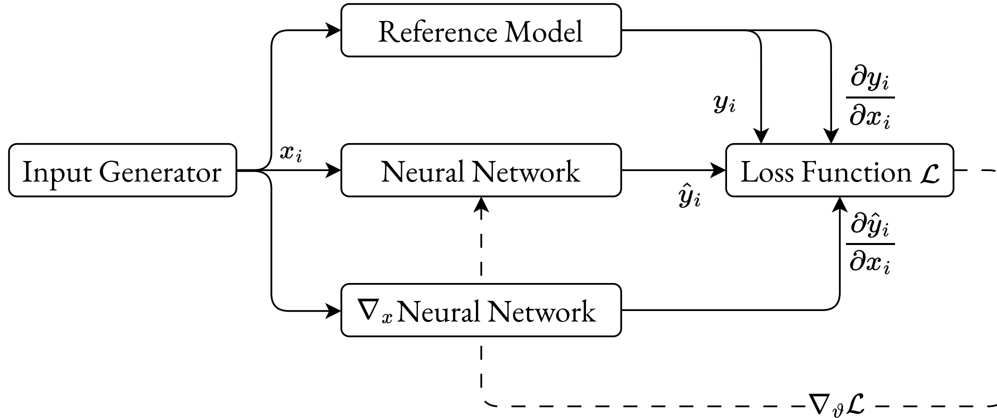


Figure 5.3: Visualization of Differential Machine Learning for a single sample.

Having established the theoretical justification of the differential loss, we now describe the full optimization algorithm in the context of option pricing. Consider an options pricing model like the Bachelier or Heston model as the reference model to be approximated. To generate a target label $\boldsymbol{y}_i$ for $\boldsymbol{x}_i$, we consider the payoff of the path: $\boldsymbol{y}_i = \nu(f(\boldsymbol{x}_i))$. The pathwise gradient of the payoff $\nabla_{\boldsymbol{x}} \boldsymbol{y}_i = \nabla_{\boldsymbol{x}} \nu(f(\boldsymbol{x}_i))$ can be obtained as described in chapter 4. To be precise, $\boldsymbol{x}_i = (\boldsymbol{\theta}_i, \boldsymbol{z}_i)$ where we only find the gradient with respect to $\boldsymbol{\theta}_i$ since $\boldsymbol{z}_i$ is some random initialization

of the noise path. For Bachelier, $\boldsymbol{\theta}_i = (\hat{S}_0,)_i$ while Heston has a path for the volatility and thus $\boldsymbol{\theta}_i = (\hat{S}_0, \hat{\nu}_0)_i$. For learning the neural network surrogate, we use a stochastic gradient-based optimizer like SGD or Adam (see subsection 2.2.2). It considers a minibatch of samples $\{(\boldsymbol{x}_i, \boldsymbol{y}_i, \nabla_{\boldsymbol{x}} \boldsymbol{y}_i)\}_{i=1}^m \sim \mathcal{S}$ from the reference model in each training iteration. The samples can either be precomputed or generated on-the-fly during a training iteration. General best practices for preprocessing the data also applies in this context. We normalize the data samples elementwise using $\tilde{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \boldsymbol{\mu}_x}{\boldsymbol{\sigma}_x}$ and $\tilde{\boldsymbol{y}}_i = \frac{\boldsymbol{y}_i - \boldsymbol{\mu}_y}{\boldsymbol{\sigma}_x}$. For normalizing the differential targets, consider some element x of $\boldsymbol{x}_i$ and y of $\boldsymbol{y}_i$. Then, by the chain rule of differentiation

$$\frac{\partial \tilde{y}}{\partial \tilde{x}} = \frac{\partial \tilde{y}}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial \tilde{x}} = \frac{\sigma_x}{\sigma_y} \frac{\partial y}{\partial x}, \tag{5.20}$$

where by linearity of the derivative $\frac{\partial \tilde{y}}{\partial y} = \frac{\partial}{\partial y} \frac{(y - \mu_y)}{\sigma_y} = \frac{1}{\sigma_y}$ and $\frac{\partial x}{\partial \tilde{x}} = \frac{\partial}{\partial \tilde{x}} (\sigma_x \tilde{x} + \mu_x) = \sigma_x$. Henceforth, we assume that the data is normalized either upfront or through a normalization layer in the surrogate model architecture.

For the drawn input samples of the batch, the neural network surrogate model generates predictions $\{\hat{\boldsymbol{y}}_i\}_{i=1}^m$. Furthermore, by using adjoint AD the pathwise derivatives $\{\nabla_{\boldsymbol{x}} \hat{\boldsymbol{y}}_i\}_{i=1}^m$ can be computed efficiently. We indicate the gradient with respect to $\boldsymbol{x}$ by $\nabla_{\boldsymbol{x}}$ which is internally implemented through VJPs. The loss $\mathcal{L}$ is $\|\cdot\|_2^2$. Taking the mean of the minibatch, we get the mean squared error as our cost function. Figure 5.3 illustrates the entire process described above on a per sample basis. In addition, Algorithm 5.1 covers all the steps of the method from the perspective of a minibatch.

Algorithm 5.1 Differential Machine Learning.

Require: The following inputs must all be initialized.

- Surrogate model $\mathcal{N}(\boldsymbol{\vartheta})$ with function $f_{\boldsymbol{\vartheta}}$ and initial parameters $\boldsymbol{\vartheta}$
- Reference model $\mathcal{S}$
- Optimizer G

while $\boldsymbol{\vartheta}$ not converged **do**
  $\{(\boldsymbol{x}_i, \boldsymbol{y}_i, \nabla_{\boldsymbol{x}} \boldsymbol{y}_i)\}_{i=1}^m \sim \mathcal{S}$ ▷ Sample training data (incl. pathwise derivatives)
  $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\vartheta}} \sum_{i=1}^m \mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \boldsymbol{y}_i) + \lambda \mathcal{L}(\nabla_{\boldsymbol{x}} f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \nabla_{\boldsymbol{x}} \boldsymbol{y}_i)$ ▷ Gradient of minibatch
  $\boldsymbol{\vartheta} \leftarrow G(\boldsymbol{\vartheta}, \hat{\boldsymbol{g}})$ ▷ Update
end while
return $\mathcal{N}$

For option pricing models the reference model $\mathcal{S}(\Theta_{\text{in}}, \mathcal{Z})$ will first sample initial $\boldsymbol{\theta}_i \sim \Theta_{\text{in}}$ and $\boldsymbol{z}_i \sim \mathcal{Z}$ for each path. The pathwise (derivative) payoff is then computed as described before. The optimizer G differs from those presented in section 2.2 in that it externalizes the sampling

and gradient calculation. It only is responsible for the surrogate parameter state updating. As a result, it more closely reflects the actual implementation and allows applying the well known optimizers in the regime of (Second-Order) Differential Machine Learning.

5.3 Variance Reduction

For an individual training iteration to be insightful, it is vital that the training batch is representative of at least some portion of the target distribution. Variance reduction techniques for pathwise derivatives that are commonly applied in options pricing can aid the training process. Many variance reduction techniques already exist with a good starting point being [16, Ch. 4]. Note, however, that not all of those methods, are applicable in the pathwise regime, e.g., Importance Sampling. One simple, yet effective method is that of antithetic paths. We, therefore, briefly describe the concept of antithetic paths. It is a technique that with little effort can reduce the variance of the payoff sampling process by making the following observation. Often the Wiener process is simulated using samples $Z_i \sim N(0,1)$ from a standard normal distribution. Then, we can further consider using the same negative samples for a second path, i.e. add $\tilde{Z}_i = -Z_i$ at every time step. If we then average between the resulting payoffs of the two paths we have an antithetic path.

To understand this approach, consider a simpler example of a uniformly distributed $U_0 \sim U(0,1)$ in the range $[0,1]$. Then, also $\tilde{U}_0 = 1 - U_0$ is uniformly distributed. Intuitively, if U_0 happens to be a large sample it can be counter-balanced by U_1. Furthermore, the mean of the average of the two remains the same: $\frac{1}{2}(U_0 + \tilde{U}_0) = \frac{1}{2}$. Now using an inverse transformation to a standard normal the same idea remains applicable, whereby $\frac{1}{2}(Z_i + \tilde{Z}_i) = 0$. Glasserman [16] shows that a sufficient condition for a reduction in the variance to occur is given by two random variables $Y_i, \tilde{Y}_i$ having negative covariance, i.e. $\mathrm{Cov}\left[Y_i, \tilde{Y}_i\right] < 0$. This is the case for the standard normal and thus variance is reduced for the antithetic paths. Note, however, that this also means that we require twice the number of paths as before which could have reduced as well. In practice, antithetic paths are usually still worthwhile.

Many other approaches for reducing the variance exist that are also applicable in the pathwise regime. For instance, by considering quasi-MC methods that use non-random sequences instead of random samples to more efficiently cover the sampling space. Moreover, multi-level MC reduces the accuracy of most samples and only requires some samples to have high accuracy. As a result, this multi-grid scheme can be much more computationally efficient [14]. Alternatively, improved discretization schemes can also improve the variance. A treatment of these methods in the context of DML is beyond the scope of this thesis.

6 Second-Order Differential Machine Learning

If first-order derivative information improves the accuracy of the surrogate model, can second-order information improve it even further? This is the question we want to address in this chapter. However, the full Hessian is often infeasible to obtain and use in each training iteration. Therefore, we will consider techniques to approximate it using the Hessian-Vector Product (HVP) which can compute second-order directional derivatives. But, in which directions should we sample HVPs? We explore answering the question using random directions and directions from a PCA in section 6.1. Furthermore, loss balancing strategies for balancing the loss, differential loss and second-order differential loss are covered in section 6.3. Finally, we present results comparing vanilla ML, DML and Second-Order DML for finding surrogate models of the Bachelier and Heston model in section 6.4.

6.1 In which directions should we sample?

Computing the full Hessian for during each training iteration is computationally infeasible as it scales quadratically in the number of training inputs times the number of training outputs. Moreover, the Hessian would have to be computed for both the neural network surrogate model and the reference model. Instead, we consider sampling second-order directional derivatives through the use of the Hessian-Vector Product (HVP). Despite its name, we do not have to materialize the full Hessian and can thus significantly reduce the computational cost. The HVP can be implemented in four distinct ways: JVP-of-JVP, JVP-of-VJP, VJP-of-JVP, and VJP-of-VJP. General AD knowledge (see section 2.3) tells us that we should prefer the VJP for first-order derivatives since it scales with the number of output values. Once we compute the second-order directional derivatives over the Jacobian, the JVP is to be preferred as it requires less memory and has favorable overhead compared to the VJP which needs to keep track of the intermediate values. As a result, the JVP-of-VJP is in most cases the preferred method to implement the HVP.

From now on, we must be more careful with what we mean by $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Do we use the JVP or VJP? In which directions do we apply the individual derivatives?

Definition 6.1 (Gradient). The gradient is the transpose of the total derivative. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, with input argument $\boldsymbol{x}$, the gradient at position $\boldsymbol{p} = (p_1, \ldots, p_n)$ is given by:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{p}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{p}) \\ \frac{\partial f}{\partial x_2}(\boldsymbol{p}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\boldsymbol{p}) \end{bmatrix}. \tag{6.1}$$

Remark. *We often use $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. However, the two $\boldsymbol{x}$ incarnations refer to different things. The bottom one refers to the fact that we take the gradient with respect to the input argument $\boldsymbol{x}$ of f. The second $\boldsymbol{x}$ is one example input denoting the position we consider to apply the gradient to. This notation is useful for functions that also depend on other arguments.*

We thus compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ in coordinates using VJPs at position $\boldsymbol{x}$ applied to the Cartesian basis vectors $\boldsymbol{e}_0, \boldsymbol{e}_1, \ldots, \boldsymbol{e}_n$. This use of the gradient is in agreement with the previous sections. In this work, we never explicitly use the VJP for directions other than the Cartesian basis vectors and thus do not give it a unique symbol. On the other hand, we do want to find directional derivatives through JVPs with directions other than the Cartesian basis vectors. We adopt the mathematical notation used by JAX [5] since it closely matches the implementation in code.

Definition 6.2 (JVP). Let $f : \mathbb{R}^n \to \mathbb{R}^m$. The Jacobian-Vector Product (JVP) is a mapping from input position $\boldsymbol{x} \in \mathbb{R}^n$ and tangent vector $\boldsymbol{v} \in \mathbb{R}^n$ to the directional derivative at $\boldsymbol{x}$ in direction $\boldsymbol{v}$:

$$\partial(f)(\boldsymbol{x}, \boldsymbol{v}) = \partial f(\boldsymbol{x})\boldsymbol{v} = \boldsymbol{J}\boldsymbol{v}, \tag{6.2}$$

where $\partial f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ is the Jacobian linear map and $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ the Jacobian.

Remark. *We make a distinction between $\boldsymbol{J}$, which is a matrix, and its linear map $\partial f(\boldsymbol{x})$ to highlight that the Jacobian does not have to be materialized to compute a JVP.*

Again, the Jacobian $\boldsymbol{J}$ could be found through application of JVPs to the standard basis vectors. For comparison, the VJP would apply the transposed vector on the left side instead, i.e.: $\boldsymbol{v}^\top \partial f(\boldsymbol{x})$.

Definition 6.3 (HVP). Let $f : \mathbb{R}^n \to \mathbb{R}$. The Hessian-Vector Product (HVP) is a mapping from position $\boldsymbol{x} \in \mathbb{R}^n$ and tangent vector $\boldsymbol{v} \in \mathbb{R}^n$ to the second-order directional derivative at $\boldsymbol{x}$ in direction $\boldsymbol{v}$. In particular, when considering an implementation of the HVP using tangent-over-adjoint mode, i.e. JVP-of-VJP, we have:

$$\partial(\nabla_{\boldsymbol{x}} f)(\boldsymbol{x}, \boldsymbol{v}) = \partial \nabla_{\boldsymbol{x}} f(\boldsymbol{x})\boldsymbol{v} = \boldsymbol{H}\boldsymbol{v}, \tag{6.3}$$

where $\boldsymbol{H} \in \mathbb{R}^{n \times n}$ is the Hessian.

Remark. *For HVPs, it is thus sufficient to consider the JVP of the gradient when f has one output.*

For the HVP, we only consider functions with one output value since the output of the payoff function in options pricing is always one-dimensional. As a result, we do not need to consider seeding the VJP since it is always $\boldsymbol{e}_0 = (1,)$.

6.1.1 Random directions

A first naive answer to the question in which directions we should sample the HVPs is to consider random ones. After all, given enough training iterations the entire space will be covered eventually. In particular, the many HVPs of random vectors should in expectation ideally converge to the full Hessian. This idea has already been explored by Martens, Sutskever, and Swersky [34] in the context of finding an estimate of the Hessian given a neural network. They consider random HVPs for optimizing a cost function during learning of a neural network and in the context of score-matching. We, on the other hand, want to use the generated information as augmented training data during learning.

They propose to draw random vectors $\boldsymbol{v}$, satisfying the constraint $\mathbb{E}[\boldsymbol{v}\boldsymbol{v}^\mathsf{T}] = \boldsymbol{I}$. Then by linearity we get an unbiased estimator: $\mathbb{E}\left[\partial \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \boldsymbol{v}\boldsymbol{v}^\mathsf{T}\right] = \partial \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \mathbb{E}\left[\boldsymbol{v}\boldsymbol{v}^\mathsf{T}\right] = \boldsymbol{H}$. The outer product can be applied after computation of the HVP. Moreover, they introduce a variance reduction technique and propose the Curvature-Propagation algorithm. While they show improved results we will not consider it in the comparative study as it is beyond the scope of this thesis. Instead, we stick to the former approach.

6.1.2 Principal Component Analysis

Our end goal is, however, not to approximate the Hessian as accurately as possible. We want to improve the learning convergence rate and final accuracy of the surrogate model. Of course, it is quite likely that a method that better approximates the Hessian for the second-order derivative targets will end up in better surrogate model training. But there might be more useful directions that could be sampled frequently in the initial phases of learning to rapidly find good-enough surrogates. We thus consider the principal component analysis (PCA) as one potential method for finding informative directions already early on.

PCA is a dimensionality reduction technique that can capture the directions of largest variance through its principal components. A simple illustration is provided in Figure 6.1. It applies PCA to a dataset of Gaussian noise samples that have been shifted and rotated. If we apply PCA to the differential data, we can find directions with large uncertainty. In particular, we thus want to find the most important principal components that capture the largest amount of variance in the data.

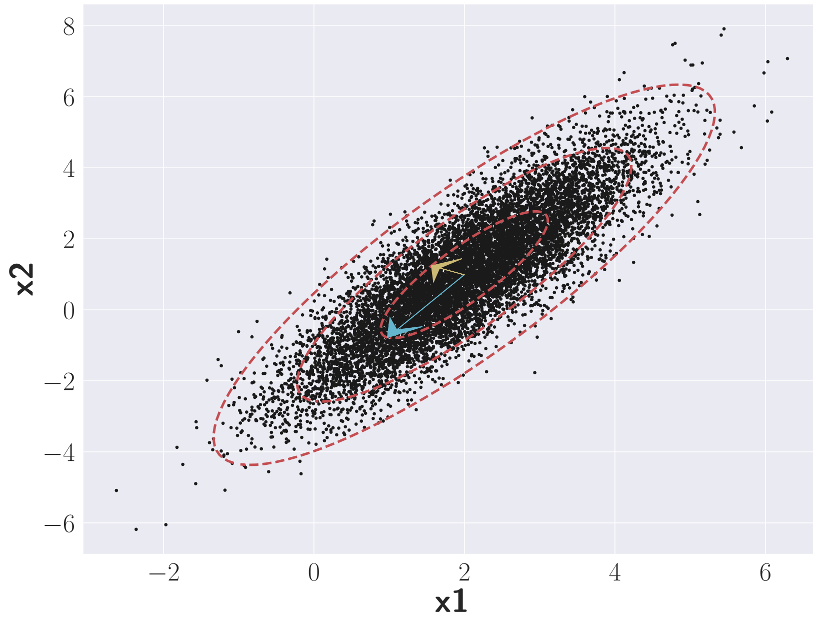


Figure 6.1: PCA capturing variance on rotated Gaussian data. In red: first three standard deviations. The scaled principal components encode directions of maximal variance (Modified from [7]).

The PCA can be implemented using an Eigendecompostion or through the Singular value decomposition (SVD). The SVD has a direct relationship to the PCA which is considered next.

Definition 6.4 (SVD)**.** The (thin) SVD

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^{\mathsf{T}}$$

decomposes matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ into the matrix $\boldsymbol{U} \in \mathbb{R}^{m\times n}$, $\boldsymbol{S} \in \mathbb{R}^{n\times n}$, and $\boldsymbol{V}^{\mathsf{T}} \in \mathbb{R}^{n\times n}$. $\boldsymbol{U}$ and $\boldsymbol{V}$ are both unitary matrices representing the left and right singular vectors, each forming an orthonormal basis. $\boldsymbol{S}$ is a diagonal matrix representing the singular values from largest to smallest.

Remark. *We only consider the thin SVD which is equivalent to the full SVD if $m > n$, but avoids computing with the irrelevant columns in $\boldsymbol{U}$ and rows in $\boldsymbol{S}$ (See [35, Ch. 12.2.3]).*

Much can be said about the SVD and the useful properties of the decomposition [35, Ch. 12.2]. We only consider it for finding the principal components. Consider the data matrix $\boldsymbol{A}$ to be mean-centered, i.e. for each data vector, subtract the mean of the data vectors. Then we find the eigenvectors of $\boldsymbol{A}^{\mathsf{T}}\boldsymbol{A}$, through a diagonalization, to be $\boldsymbol{V}$.

$$
\begin{aligned}
\boldsymbol{A}^\top \boldsymbol{A} &= (\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top)^\top(\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top) \\
&= \boldsymbol{V}\boldsymbol{S}^\top\boldsymbol{U}^\top\boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top \\
&= \boldsymbol{V}\boldsymbol{S}^\top\boldsymbol{S}\boldsymbol{V}^\top \quad \text{(since } \boldsymbol{U} \text{ is unitary)} \\
&= \boldsymbol{V}\boldsymbol{S}^2\boldsymbol{V}^\top.
\end{aligned}
$$

Since

$$\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}\boldsymbol{S}^2.$$

As a result, the right singular vectors in $\boldsymbol{V}$ correspond to directions for the principal components. Furthermore, the eigenvalues $\boldsymbol{S}^2 = \texttt{diag}(\boldsymbol{s}^2)$ of $\boldsymbol{A}^\top\boldsymbol{A}$ are proportional to the variance of the principal components

$$\boldsymbol{s}_{\sigma^2} = \boldsymbol{s}^2/\mathsf{sum}(\boldsymbol{s}^2) = \boldsymbol{s}^2/\sum_i^n s_i^2.$$

We can thus use $\boldsymbol{s}_{\sigma^2}$ to select principal components that describe a certain percentage of the variance. The percentage of variance explained by the $k_{\boldsymbol{v}}$ most important principal components is found by

$$\boldsymbol{k}_{\boldsymbol{v}} = \arg\max(\mathsf{cumsum}(\boldsymbol{s}_{\sigma^2}) > \kappa),$$

where cumsum is the cumulative sum defined as usual and arg max finds the index of the first occurrence where the condition κ is true. For example, if $\kappa = 0.95$ we select the principal components describing 95% of the variance.

Finding all the principal components at each iteration of training can be computationally expensive even with the thin SVD taking $\mathcal{O}(mn^2)$ time. Alternative implementations that only need to find the first $k_{\boldsymbol{v}}$ singular values could be much faster. In particular, Krylov subspace iteration methods can be much faster in this context. In addition, the initial starting eigenvectors for the iteration could be informed by previous training steps for even faster convergence. Otherwise, PCA could be applied to only every nth iteration. Also, the PCA could be applied on quantized data since the precise values are not needed to find generally important directions. The use of iterative methods and quantization is beyond the scope of this thesis but highlights that considering principal components in each iteration can be feasible for larger problems.

6.2 Main algorithm

We now have all the necessary components to present the main algorithm for second-order DML. A brief outline of the approach was already given in subsection 1.3.1. Figure 1.1 visualizes the general building blocks. How those pieces fit together will be made precise in this section.

Starting from the DML Algorithm 5.1, we extend the cost function with an additional loss term for the second-order differential information. As in the justification of the differential loss, a probabilistic argument can be made to consider the posterior $p(\boldsymbol{\vartheta}|\boldsymbol{x}, \boldsymbol{y}, \nabla_{\boldsymbol{x}}\boldsymbol{y}, \partial\nabla_{\boldsymbol{x}}\boldsymbol{y}, \sigma^2)$ depending on the second-order differential information.

$$p(\boldsymbol{\vartheta}|\boldsymbol{x}, \boldsymbol{y}, \nabla_{\boldsymbol{x}}\boldsymbol{y}, \partial\nabla_{\boldsymbol{x}}\boldsymbol{y}, \sigma^2) \propto p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\vartheta}, \sigma^2)p(\nabla_{\boldsymbol{x}}\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\vartheta}, \sigma^2)p(\partial\nabla_{\boldsymbol{x}}\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\vartheta}, \sigma^2) \tag{6.4}$$

Then the negative log-likelihood results in the full second-order differential loss, considering the entire Hessian. As noted before, this formulation is infeasible in most cases and instead we define the second-order differential loss to depend on HVPs of $k_{\boldsymbol{v}}$ tangent vectors.

Definition 6.5 (Second-Order Differential Loss)**.** Given input $\boldsymbol{x}$, target $\boldsymbol{y}$, predicted output $f_{\boldsymbol{\vartheta}}(\boldsymbol{x})$, differential target $\nabla_{\boldsymbol{x}}\boldsymbol{y}$, predicted differential $\nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}}(\boldsymbol{x})$, second-order differential target $\partial(\nabla_{\boldsymbol{x}}(f_{\boldsymbol{\vartheta}}))(\boldsymbol{x}, \boldsymbol{v}_k)$, and predicted second-order differential $\partial(\nabla_{\boldsymbol{x}}f)(\boldsymbol{x}, \boldsymbol{v}_k)$ given tangent vector $\boldsymbol{v}_k$, the second-order differential loss is defined by:

$$\lambda_0\|\boldsymbol{y} - f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\|_2^2 + \lambda_1\|\nabla_{\boldsymbol{x}}\boldsymbol{y} - \nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}}(\boldsymbol{x})\|_2^2 \tag{6.5}$$

$$+ \lambda_2 \sum_{k=1}^{k_{\boldsymbol{v}}} \|\partial(\nabla_{\boldsymbol{x}}(f_{\boldsymbol{\vartheta}}))(\boldsymbol{x}, \boldsymbol{v}_k) - \partial(\nabla_{\boldsymbol{x}}f)(\boldsymbol{x}, \boldsymbol{v}_k)\|_2^2, \tag{6.6}$$

where $\lambda_0, \lambda_1, \lambda_2 \in \mathbb{R}_{\geq 0}$ are parameters for balancing the loss terms, and $k_{\boldsymbol{v}}$ the number of tangent vectors to consider.

The final missing piece is to find the tangent vectors for the minibatch. We apply PCA via SVD to the mean subtracted derivative information $\{\nabla_{\boldsymbol{x}_i}\tilde{\boldsymbol{y}}_i\}_{i=1}^m$ and find the principal components through the right singular vectors of the SVD. In addition, we scale the principal components using the singular values to give the directions appropriate weight. Then we mean adjust the vectors to be applicable on the original data. If the training data is already normalized to mean 0 and variance 1, as is commonly suggested, we do not require the steps of subtracting the mean from the data and mean adjusting the principal components. Normalization for second-order DML is akin to that of DML except that we also need to normalize the second-order targets. Consider, as before, some element x of $\boldsymbol{x}_i$ and y of $\boldsymbol{y}_i$. Then

$$\frac{\partial^2 \tilde{y}}{\partial \tilde{x}^2} = \frac{\partial}{\partial \tilde{x}}\left(\frac{\partial \tilde{y}}{\partial \tilde{x}}\right) \tag{6.7}$$

$$= \frac{\partial}{\partial x}\left(\frac{\sigma_x}{\sigma_y}\frac{\partial y}{\partial x}\right)\frac{\partial x}{\partial \tilde{x}} \quad \text{(by the chain rule and Equation 5.20)} \tag{6.8}$$

$$= \frac{\sigma_x}{\sigma_y}\frac{\partial^2 y}{\partial x^2}\sigma_x \qquad \text{(by taking } x = \tilde{x}\sigma_x + \mu_x\text{)} \tag{6.9}$$

We find the $k_{\boldsymbol{v}}$ most important principal components through the cumulative explained variance. So, we sum up the singular value scores until we reach, e.g., 95% of variance explained. Finally, we can apply the HVPs on the reference model and compute the predictions, differential

Algorithm 6.1 Second-Order Differential Machine Learning.

Require: The following inputs must all be initialized.

- Surrogate model $\mathcal{N}(\boldsymbol{\vartheta})$ with function $f_{\boldsymbol{\vartheta}}$ and initial parameters $\boldsymbol{\vartheta}$.
- Reference model $\mathcal{S}$.
- Optimizer G.
- hyperparameter κ for selecting principal components. By default, $\kappa = 0.95$, i.e. select the principal components explaining 95% of the variance in the pathwise gradients.
- loss balancing parameters $\lambda_0, \lambda_1, \lambda_2$.

while $\boldsymbol{\vartheta}$ not converged **do**

$\{(\boldsymbol{x}_i, \boldsymbol{y}_i, \nabla_{\boldsymbol{x}}\boldsymbol{y}_i)\}_{i=1}^m \sim \mathcal{S}$ ▷ Sample training data (incl. pathwise derivatives)

$\boldsymbol{\mu} \leftarrow \{\frac{1}{m}\sum_{i=1}^m \nabla_{\boldsymbol{x}}\boldsymbol{y}_i\}$ ▷ Mean of pathwise gradients

$\{\nabla_{\boldsymbol{x}_i}\tilde{\boldsymbol{y}}_i\}_{i=1}^m \leftarrow \{\nabla_{\boldsymbol{x}}\boldsymbol{y}_i - \boldsymbol{\mu}\}_{i=1}^m$ ▷ Mean subtracted data

$(\boldsymbol{U}, \boldsymbol{s}, \boldsymbol{V}^\top) \leftarrow \text{SVD}(\{\nabla_{\boldsymbol{x}_i}\tilde{\boldsymbol{y}}_i\}_{i=1}^m)$ ▷ Singular Value Decomposition

$\{\tilde{\boldsymbol{v}}_k\}_{k=1}^{n_0} \leftarrow \texttt{diag}(\boldsymbol{s})\boldsymbol{V}$ ▷ Principal components

$\{\boldsymbol{v}_k\}_{k=1}^{n_0} \leftarrow \{\tilde{\boldsymbol{v}}_k + \boldsymbol{\mu}\}_{k=1}^{n_0}$ ▷ Principal components (mean adjusted)

$\boldsymbol{s}_{\sigma^2} \leftarrow \boldsymbol{s}^2/\texttt{sum}(\boldsymbol{s}^2)$ ▷ Scaled $\boldsymbol{s}$ to represent % of variance

$k_{\boldsymbol{v}} \leftarrow \arg\max(\texttt{cumsum}(\boldsymbol{s}_{\sigma^2}) > \kappa)$ ▷ Take $k_{\boldsymbol{v}}$ most significant principal components

Gradient $\hat{\boldsymbol{g}}$ of minibatch:

$$\hat{\boldsymbol{g}} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\vartheta}}\sum_{i=1}^m\Bigg[\lambda_0\mathcal{L}(f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \boldsymbol{y}_i) + \lambda_1\mathcal{L}(\nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}}(\boldsymbol{x}_i), \nabla_{\boldsymbol{x}_i}\boldsymbol{y}_i) + \lambda_2\sum_{k=1}^{k_v}\mathcal{L}(\partial(\nabla_{\boldsymbol{x}}f_{\boldsymbol{\vartheta}})(\boldsymbol{x}_i, \boldsymbol{v}_k), \partial(\nabla_{\boldsymbol{x}}f)(\boldsymbol{x}_i, \boldsymbol{v}_k))\Bigg]$$

$\boldsymbol{\vartheta} \leftarrow G(\boldsymbol{\vartheta}, \hat{\boldsymbol{g}})$ ▷ Update surrogate parameters

end while

return $\mathcal{N}$

predictions and second-order differential predictions using the principal components as tangent vectors. It remains to compute and backpropagate the gradient of the cost function through the surrogate model for learning to occur. The entire method is described in Algorithm 6.1.

6.3 Loss Balancing

Ideally, the loss balancing parameters are defined implicitly by the algorithm. In DML, Huge and Savine [21] suggest to treat each sample of the targets and differential targets with equal weight. Since each training target $\boldsymbol{y}_i \in \mathbb{R}$ has accompanying differential target $\nabla_{\boldsymbol{x}} \boldsymbol{y}_i \in \mathbb{R}^n$ for $\boldsymbol{x}_i \in \mathbb{R}^n$, we get

$$\lambda_0 = \frac{1}{1+\alpha n}, \quad \lambda_1 = \frac{\alpha n}{1+\alpha n}, \tag{6.10}$$

where α is a hyperparameter weighing the importance of differential targets for training. By default, it is just $\alpha = 1$.

However, this static approach is insufficient for second-order learning. Since we do not compute the entire hessian, it is unclear upfront how many second-order differential targets will be considered in any particular training iteration. Depending on how many principal components are needed to account for the desired percentage of variance, the number of samples differs. Therefore, we suggest to use the information provided by PCA and perform adaptive loss balancing. We start with a generalization of Equation 6.10.

$$c = 1 + \alpha n + \beta n^2 \tag{6.11}$$

$$\lambda_0 = \frac{1}{c}, \quad \lambda_1 = \frac{\alpha n}{c}, \quad \lambda_2 = \frac{\beta n^2}{c}, \tag{6.12}$$

where β is a hyperparameter weighing the importance of second-order differential targets. By default, $\beta = 2k_{\boldsymbol{v}}/n^2$ and the loss will thus be informed by the number of principal components used in the iteration. We end up with $\lambda_2 = 2k_{\boldsymbol{v}}/c$ and a factor of 2, because the hessian would be symmetric. If random directions are to be used, we do not know $k_{\boldsymbol{v}}$. In this case, the algorithm can be adapted in such a way that it takes κ as the hyperparameter for the percentage of n^2 vectors to draw randomly. Note that, so far, we have only considered the number of generated samples irrespective of their magnitude. This is usually fine, as long as the data has been normalized for training. However, we implicitly assume that the data is, therefore, close to normally distributed since we use z-score normalization. If the variance of a minibatch turns out to be much larger, this could be problematic.

6.4 Results

In this section, we evaluate the proposed methods considering two case studies. We first study a Bachelier model of a multidimensional basket option, before we consider the stochastic volatility model by Heston.

Surrogate Model

In both scenarios, we use a MLP as the surrogate model (see section 2.1). The MLP has 4 hidden layers with width 20 and, uses the CELU activation function, a continuously differentiable alternative to ReLU (see subsection 2.1.2). The optimizer G is chosen to be Adam with default settings as in Algorithm 2.2. In addition, we use a cosine one-cycle learning rate schedule with peak $\eta = 0.1$, raising η for 30% of the cycle, starting at $\eta = 4\text{e}{-3}$, and finishing with $\eta = 1\text{e}{-5}$.

Bachelier

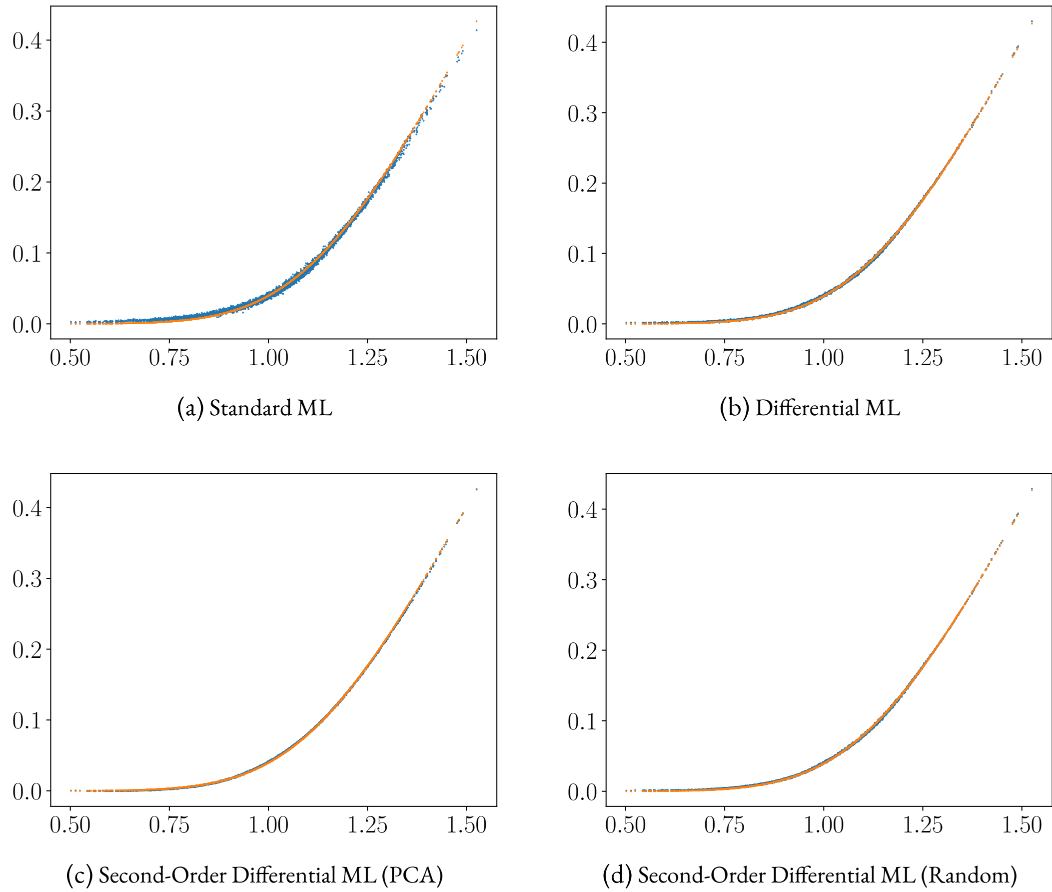
The Bachelier model is described in section 3.1. To be comparable with previous results [21], we also use a basket of 7 underlying assets. Therefore, we have the model

$$\mathrm{d}\mathbf{F}_t = \sigma \, \mathrm{d}\mathbf{W}_t, \tag{6.13}$$

where $\mathbf{F}_t \in \mathbb{R}^7$ and $\mathrm{d}W_t^j \, \mathrm{d}W_t^k = \rho_{jk}$. The correlation matrix is generated from a random covariance matrix. The parameters for the simulation are as follows: The option has maturity $T = 1$ (year), initial spot prices $\mathbf{F}_0$ are Gaussian distributed and centered at $\mathbf{100}$, strike price $K = 110$, and basket volatility $\sigma = 0.2$. In the implementation, all prices are scaled down by a factor of 100. Furthermore, antithetic paths are optionally used as described in section 5.3. The weights of the basket assets are sampled by a uniform distribution and scaled to sum up to 1. For the European payoff function, we use sigmoidal smoothing with $w = 0.005$. European payoff for the weighted sum is applicable for the basket since the basket price is Gaussian due to the individual underlying assets being jointly Gaussian distributed. The sampler $\mathcal{S}$ uses 8192 samples which get repeatedly reshuffled during the training iterations. We thus want to observe the performance also given limited available data.

A comparison of the different methods for learning to predict the price of a basket call option that is modelled by a Bachelier model is given in Figure 6.2. We observe a significant decrease in the root mean squared error (RMSE) of price predictions when considering differential ML (0.345 vs 0.123). Second-Order DML further decreases the error (0.089). While standard ML produces reasonable results for the prices, the Greeks (Delta and Gamma) have significantly worse predictions (0.550 and 97.547). It is to be expected given that the training did not inform the

learning about the Greeks. Again, DML halves the RMSE (0.275 and 88.753) whereas second-order DML improves the RMSE by more than 3.5 times compared to ML (0.152 and 74.737).



(a) Standard ML

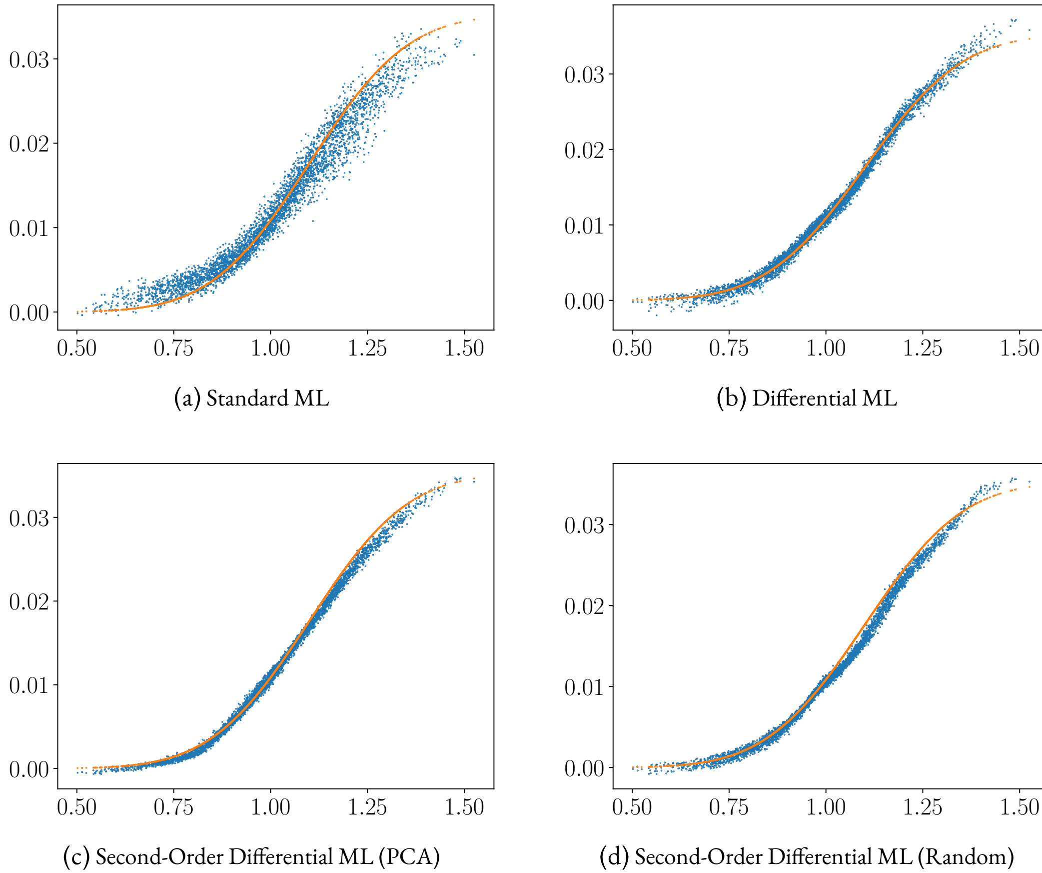
(b) Differential ML

(c) Second-Order Differential ML (PCA)

(d) Second-Order Differential ML (Random)

Figure 6.2: Price V given initial spot price S_0 of Bachelier call option basket model of 7 dimensions learned with different surrogate models. Orange represents the analytic solution, blue the predictions.

Moreover, second-order DML using PCA is the only method that qualitatively captures the Gamma curve (Figure 6.4c). Note, however, that there seems to exist a staircase pattern on the wings which is reproducible on many runs. The origin of this behaviour is not fully understood. One hypothesis is that the limited directions seen through the (k_v reduced) PCA results in clustering values closer together than they should. Further investigation in this area is warranted. To ensure that these results were not caused by random fluctuations, we ran every test 30 times to compute a mean and standard deviation (see Table 6.1–6.3). Second-Order DML consistently outperformed the other methods. A 50 dimensional case further manifests the improved accuracy of these models.



(a) Standard ML (b) Differential ML

(c) Second-Order Differential ML (PCA) (d) Second-Order Differential ML (Random)

Figure 6.3: Delta $\frac{\partial V}{\partial S_0}$ given initial spot price S_0 of Bachelier call option basket model of 7 dimensions learned with different surrogates. Orange represents the analytic solution, blue the predictions.

A variation of second-order DML using random tangent vectors is also considered. Trying to apply the technique directly led to no improvements over regular ML. After investigating the problem, it became clear that the balance between the different loss terms is crucial for efficient learning to occur.

Table 6.1: RMSE of surrogate models for predicting the **price** of a Bachelier modelled basket option.

# Assets	# Samples	Method			
		Standard ML	DML	2nd-Order PCA	2nd-Order RNG
7	8192	0.320 ±0.022	0.123 ±0.009	**0.101±0.016**	**0.099±0.004**
50	16384	0.0131	0.0062	0.0020	–

After many unsatisfactory attempts in trying to incorporate existing loss balancing strategies including Soft Adept [20], and ReLoBraLo [3], we found that balancing the mean losses using softmax similar to ReLoBraLo after each iteration provides significant benefits. The extend to

Table 6.2: RMSE of surrogate models for predicting the **Delta** of a Bachelier modelled basket option.

# Assets	# Samples	Method			
		Standard ML	DML	2nd-Order PCA	2nd-Order RNG
7	8192	0.532 ±0.009	0.261 ±0.025	**0.155±0.018**	0.231±0.010
50	16384	0.0027	0.0011	0.0007	–

Table 6.3: RMSE of surrogate models for predicting the **Gamma** of a Bachelier modelled basket option.

# Assets	# Samples	Method			
		Standard ML	DML	2nd-Order PCA	2nd-Order RNG
7	8192	97.625 ±0.33	87.390 ±2.20	**75.54±0.51**	87.392 ±1.31
50	16384	1.9224	1.6090	0.8736	–

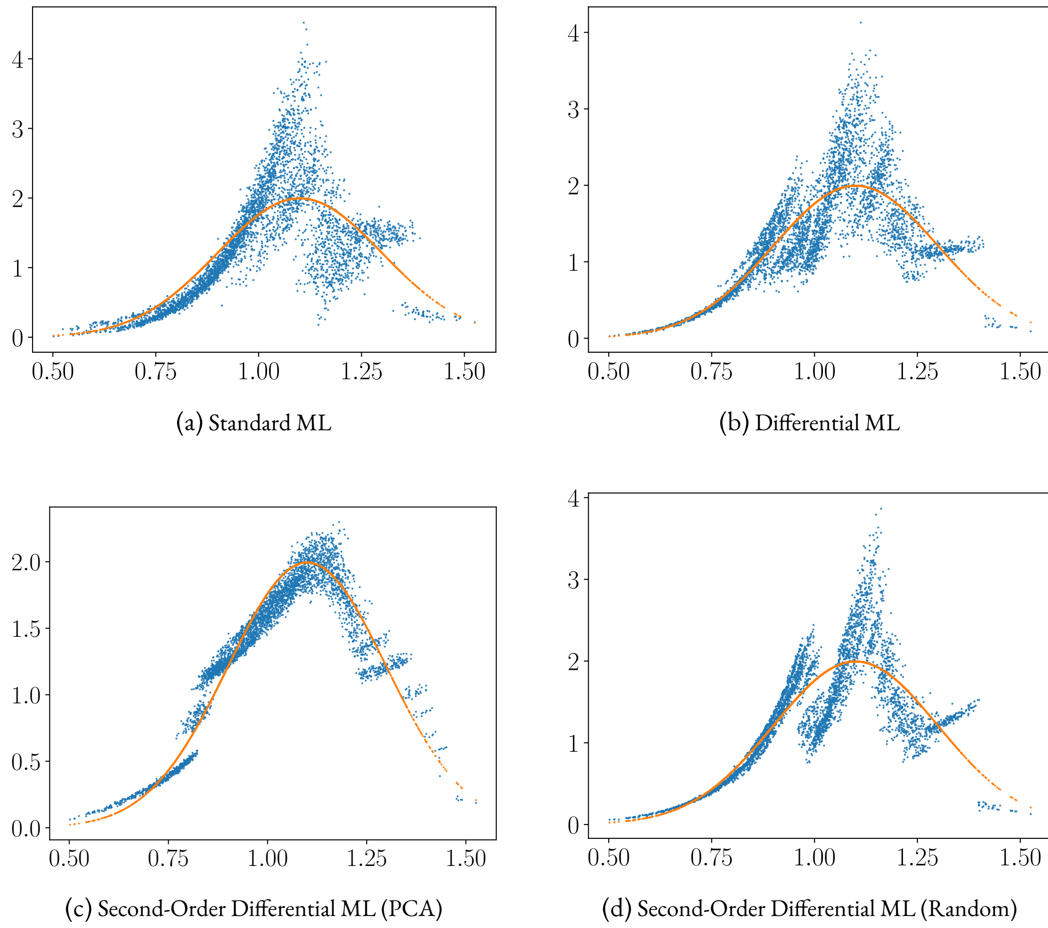
which this property is generalizable or is merely a coincidence of the specific problem domain is still an open problem that we did not solve. Nonetheless, with such an adaptive loss balancing the price predictions are on par with the PCA approach. The Greeks, on the other hand, remain worse than that of the PCA version, yet still better or on par with DML. For the above reasons, we did not consider random directions for the 50 dimensional case.

Heston

The Heston model is described in section 3.2. The option has maturity $T = 1$ (year), initial spot prices $S_0 \sim U(50, 150)$ and initial volatility $\nu_0 \sim U(0.01, 0.1)$, strike price $K = 100$, correlation $\rho = -0.3$, interest rate $r = 0$, mean reversion rate $\kappa = 1$, $\xi = 1$, and $\theta = 0.09$. Normalization as described in section 5.2 is applied during training.

We compare the results of standard ML with DML in Figure 6.6. If we compare the results to the true targets shown in the test data section of Figure 6.7, we can observe a significant improvement in the prediction of the Greeks. Concretely, the RMSE of price predictions drop from 0.0309 to 0.0162, the predicted Deltas from 0.0581 to 0.0359 and the predicted Vegas from 0.0191 all the way down to 0.0071. However, so far we did not consider second-order DML. It turns out that we had a lot of trouble to get the approach working using the regular Euler-Maryuama discretization scheme. After further investigation it became clear that the variance and kurtosis of the pathwise (derivative) payoff samples are getting very large. While we are aware of techniques to combat this problem, it would go much beyond the scope of this thesis (see section 8.1 for a discussion and potential future directions). Instead, we highlight and visualize the key problems that need to be addressed for the second-order DML approach to be applicable.

The training data shown in Figure 6.7e already hints at the problem. If we for example generate 2^{13} payoff samples the variance was computed to be around 5.26 with a skewness of 342 and kur-



(a) Standard ML

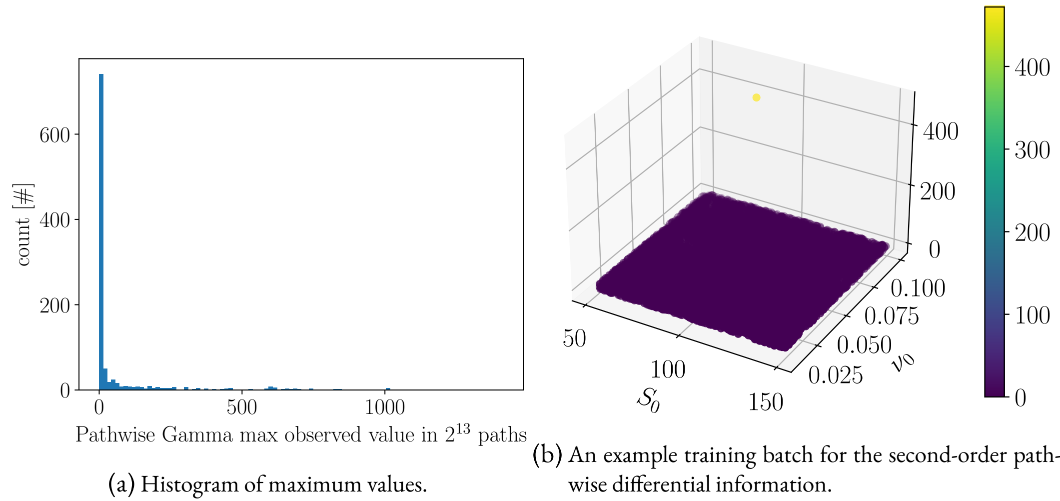
(b) Differential ML

(c) Second-Order Differential ML (PCA)

(d) Second-Order Differential ML (Random)

Figure 6.4: Gamma $\frac{\partial^2 V}{\partial S_0^2}$ given initial spot price S_0 of Bachelier call option basket model of 7 dimensions learned with different surrogates. Orange represents the analytic solution, blue the predictions.

tosis at 137745. These astronomical numbers are further demonstrated by mean and max values 1024 simulations of 2^{13} paths. If we look at a single training batch, it becomes clear that almost all information will be uninformative, i.e. 0, and only very rarely we obtain useful information (see Figure 6.5b). If the samples are informative, then the values usually blow up into the other direction (like the yellow dot), thereby resulting in a tail heavy distribution (see Figure 6.5a).



(a) Histogram of maximum values.

(b) An example training batch for the second-order pathwise differential information.

Figure 6.5: Most pathwise second-order derivative payoff samples are uninformative for small volatility.

Under these circumstances, learning will be almost impossible since most of the training batches will be 0, thereby pushing the second-order samples to 0 until the gradient becomes 0. With a gradient of 0, no parameters will be changed and thus no learning occurs. It highlights the importance of variance reduction in the context of pathwise derivative computation.

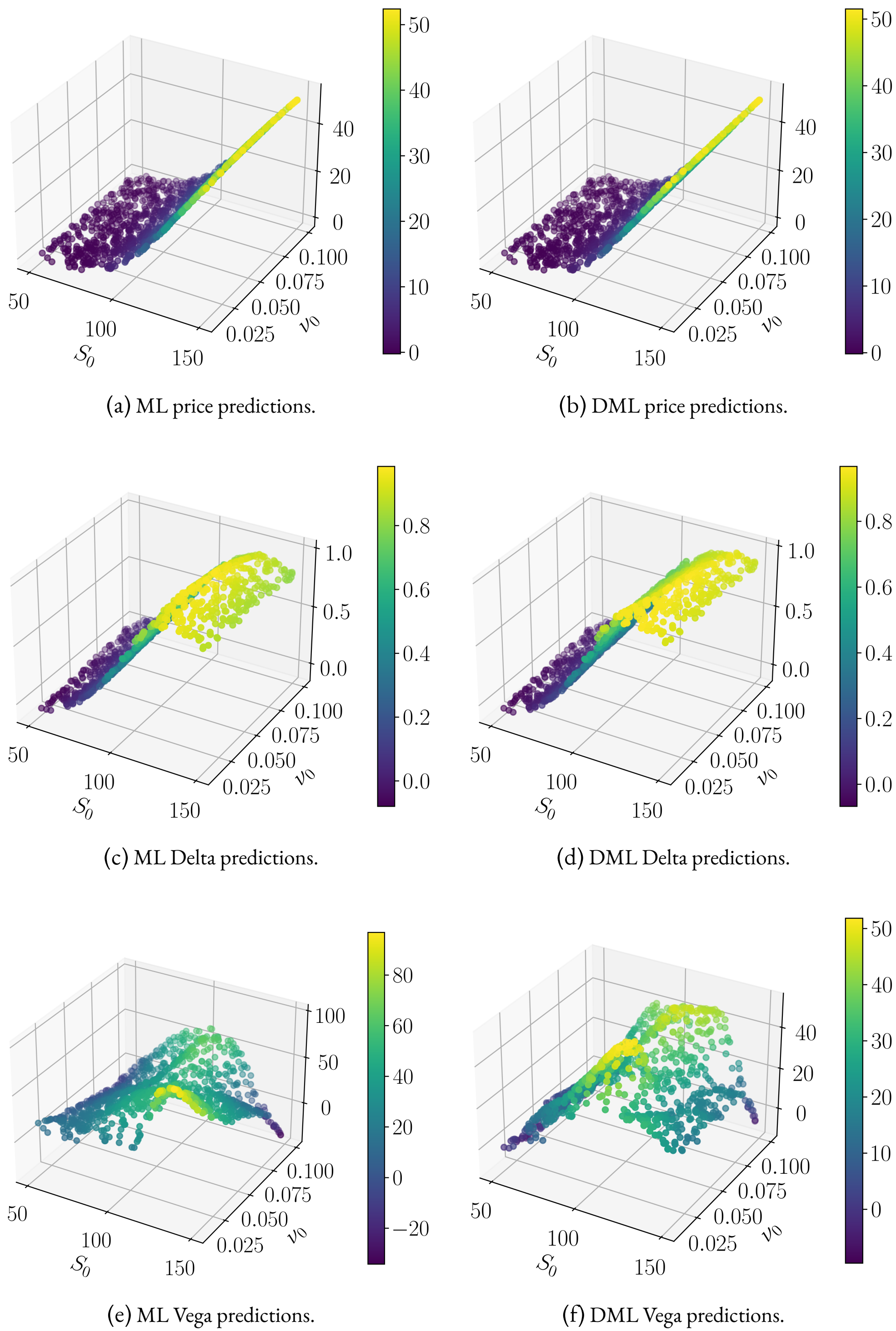
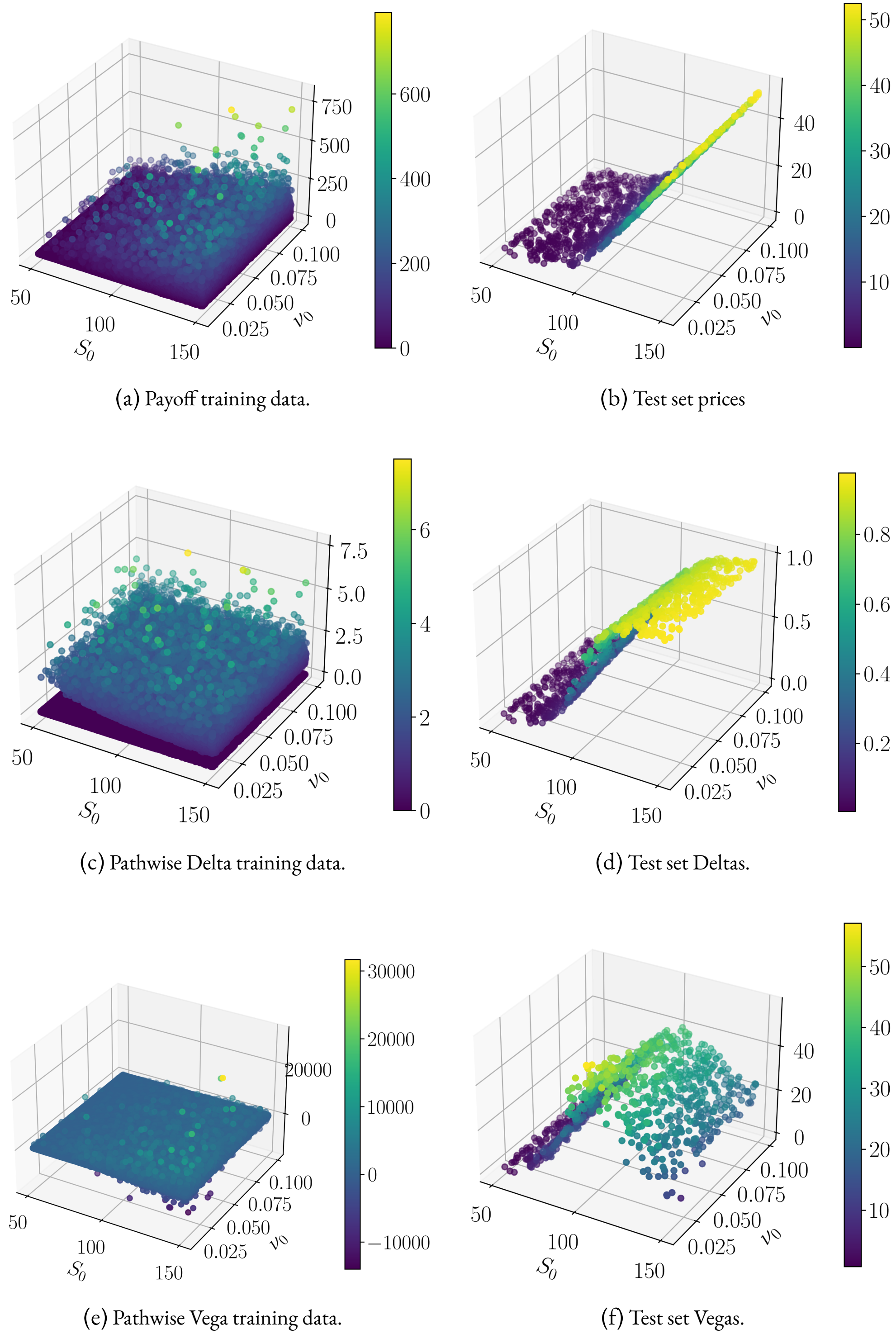


Figure 6.6: Comparison of ML and DML surrogate models for predicting the price and Greeks of the Heston model. Bottom axis correspond to initial spot price S_0 and initial volatility ν_0.



(a) Payoff training data.

(b) Test set prices

(c) Pathwise Delta training data.

(d) Test set Deltas.

(e) Pathwise Vega training data.

(f) Test set Vegas.

Figure 6.7: Heston training data (left) and test set with prices and Greeks (right).

7 Related Work

Much work has gone into using neural networks to enhance, augment, or replace many of the techniques in traditional numerical methods. This section highlights many notable efforts with no aim to be exhaustive. We point out the key differences to what is proposed in this thesis. It should give a broad overview of the landscape of neural network based techniques for approximating (stochastic) differential equations.

Sobolev Training

The concept of Differential ML, as introduced by Huge and Savine [21], has already been explained. In addition, Czarnecki et al. [10] already introduced a similar methods, referred to as Sobolev Training, and extended it to other domains such as model distillation and classification tasks. However, second-order derivative information was so not fully considered by the authors. Christodoulou [9] briefly explored the use of Hessian information but it did not result in better training convergence or accuracy in their experiments. The reason for this behaviour remained largely unexplored. With our proposed method, we show that it is possible to successfully embed second-order derivative data into the learning process.

SDEs as PDEs

Many connections have been established between SDEs and PDEs. As such, the extensive theory of learning neural networks for solving PDEs could be applied in this setting. However, PDEs loose the independence property of Monte Carlo paths and instead compute values based on a particular stencil pattern. Parallelism in PDE solvers is thus notoriously challenging, making such approaches often infeasible for training in high dimensions. The proposed context is thus usually preferred, but interesting results can nonetheless be found in the area of applying ML surrogates to PDEs, e.g. PINNs.

PINNs

Physics-Informed Neural Networks (PINNs) aim to aid the learning process of a neural network by incorporating physically inspired invariants into the loss function [40, 41, 42]. It is a form of

soft regularization that, in its most basic form, adds the PDE residual and boundary conditions to the loss function. Note that gradient information can also aid training in this context, as the gradient of the PDE residual must be 0 as well [55]. While this method can result in improved model accuracy, it remains an open problem area as the soft regularization can make optimization challenging [29]. Furthermore, the original concept is limited to ODEs/PDEs, but it can be extended to modeling stochastic processes, as demonstrated in NN-aPC [56].

PI-GANs

PI-GANs take the idea of PINNs and apply it to generative adverserial networks (GANs) to model stochastic processes [54]. It is a data driven training method considering physical constraints. They focus on generating surrogate sample paths instead of mean prediction. Of course, once the sample paths are accurately modelled, mean prediction is possible through standard techniques. However, GANs are notoriously difficult to train and require significantly more compute compared to regular neural networks. Moreover, the training behaviour in high dimensional settings has not been thoroughly investigated yet. Initial experiments by Yang, Zhang, and Karniadakis [54] suggest that training in high dimensions is possible. We view this class of models as a promising candidate for learning from sample data in domains where a accurate reference model is hard to come. Since PI-GANs are differentiable, they could be used as a reference model in DML.

Neural Differential Equations

Neural ODEs solve the problem of parameter specification in differential equations by learning the parameters through an embedded neural network [24]. This idea can be extended to the domain of stochastic differential equations (Neural SDEs) [26, 30], allowing for calibration of SDEs to real world data. While this is also required for the Heston model, this thesis will not consider the problem of SDE calibration. Instead, we focus on efficiently obtaining solutions to pre-specified SDEs. However, as in the case of PI-GANs, Neural SDEs are fully differentiable and could be used as reference models for training in DML. The combination could establish itself as a powerful technique for learning surrogate models for price predictions given a calibrated and fitted Neural SDE model.

Gaussian Processes

Going beyond neural networks, we might want to consider Gaussian Processes [35, Ch. 15]. They have the advantage of directly encoding uncertainty into the estimation. However, the training and inference time of Gaussian processes can be slow, especially when dealing with high-dimensional data.

8 Conclusion

In this thesis, we investigated the use of learning with differential data. We have seen how to use pathwise data to compute a mean prediction. In particular, for option pricing we justified the use of Least-Squares Monte Carlo, allowing to train prices from pathwise payoff samples. It can be extended to learning the Greeks through pathwise derivatives given potentially smoothed payoff functions. If we, furthermore, consider the input data to be perturbed by Gaussian noise, the need for optimization using derivative information arises naturally. We introduced Differential Machine Learning as a general method to embed differential targets into the training of neural network surrogate models. It extends typical learning by predicting pathwise derivative targets through adjoint AD over the reference and surrogate model. We, moreover, hint at theoretical justifications for this approach and demonstrate its improved accuracy over standard ML.

The central questions underlying this research revolved around getting insight into the effect second-order differential data could have to the learning process. The infeasible nature of comparing the Hessian at each training iteration opens up a large landscape of approximate techniques that try to capture the essence of the curvature information through as little HVPs as possible. Beyond random directions, we propose to use PCA on differential data to find principal components of maximal variance. We have experimentally shown the superior accuracy of second-order DML in pricing a Bachelier modelled basket option. However, second-order DML opens up many challenges, from dealing with discontinuities in the models, adaptive loss balancing, to the need for variance reduction. We, therefore, discuss some of the remaining challenges in making this method applicable to larger applications and look at potential future directions.

8.1 Discussion

The proposed second-order DML method delivers promising results, yet at the same time raises many new questions. We would like to understand the scaling behaviour of this method in more detail. The existing models that use a MLP still train in a matter of seconds on the GPU. On this scale, the added training cost over DML is just a small multiple. If instead more sophisticated neural network architectures are to be used, it becomes vital to understand the impact on training iteration speed the proposed methods have. As with any hyperparameter, finding generally

appropriate settings can be challenging. Currently, the default is to consider the principal components describing 95% of the variance, i.e. $\kappa = 0.95$. It is an arbitrary choice pointing towards the need for a better understanding of the compression properties of the differential data space in the context of second-order DML.

On a technical aside, the proposed method raises an interesting challenge to AD. The optimal choice for the HVPs in the context of ML is almost always taking a forward-over-reverse approach. However, when using PCA on the differential data, we only find the principal components, i.e. the directions for the JVPs, after we computed the VJPs. As a result, we first have to materialize the Jacobian and only then apply the JVP-of-VJP. We thereby perform some computations twice. Using a reverse-over-reverse method would circumvent this problem as you can seed the adjoint output vectors after the primal computation has been performed. Nonetheless, the added overhead and the much higher memory consumption will probably make the forward-over-reverse method the only option for large neural network surrogate models. We are looking forward to clever optimizations in this context.

The experimental studies of the Heston model shine a light on the importance of variance reduction. Without meaningful training batches, learning will not occur. Considering better discretization schemes or antithetic paths would be obvious next steps. Moreover, Vibrato Monte Carlo is an interesting generalization of the pathwise derivative method that could be worth considering. Giles [13] proposes to model the final time step of the discretized SDE path via a Gaussian distribution. We could then take the gradient with respect to the expectation of this Vibrato path. It is always differentiable and would, in addition, allow the use of arbitrary payoff functions without the need for smoothing.

8.2 Future Directions

Besides working on the above mentioned open problems, there exists many promising future avenues. The techniques could be used in the context of classification. Also, why should we stop at second-order derivative information? What about third-order, fourth-order, ..., n^{th}-order derivative information? Does there exist a context in which such information would provide meaningful benefits? Is there a generalizable cost-benefit analysis to be found for incorporating the nth-order derivative information into learning? Furthermore, sampling from the stochastic reference model could be made more efficient with well known quasi-random sequences. But, what if we considered the already generated derivative information as uncertainties to consider for future sampling of the reference model? Finally, PCA only considers linear transformations. Would Kernel PCA, or a non-linear method like the autoencoder help in requiring even fewer directions to sample? Despite these uncertainties, one thing is clear: The gradient will lead us to the optimal direction!

A On the code developed

The proposed method are implemented using JAX [5] and the corresponding repository can be found under: https://github.com/neilkichler/diff-ml.

The specific results presented in this thesis are located in the `notebooks` directory. In particular, `notebooks/bachelier.ipynb` and `notebooks/heston.ipynb`. In this section, we give a high level, simplified overview of some of the most important implementation aspects. We assume a basic understanding of JAX as outlined in section 2.4. Further information can be found in their official documentation. In addition, we use the Equinox and Optax library for neural network training. Equinox is a thin wrapper on top of JAX. It transforms custom classes using `eqx.Module` into `PyTrees` (which JAX knows how to deal with). Optax is an optimization library that, just like the Optimizer G in our algorithms, can take in a current state of the parameters and some gradients to produce a new state for the parameters. It is also worth noting that JAX is a functional subset of Python, i.e. it requires explicit state. That also implies that the random state is explicitly provided for each function. As an example, consider implementing a linear layer as described in Definition 2.4.

```
import equinox as eqx
import jax

class Linear(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __init__(self, in_size, out_size, key):
        wkey, bkey = jax.random.split(key)
        self.weight = jax.random.normal(wkey, (out_size, in_size))
        self.bias = jax.random.normal(bkey, (out_size,))

    def __call__(self, x):
        return self.weight @ x + self.bias
```

Figure A.1: Linear layer in JAX, requiring explicit random state.

```
class MLP(eqx.Module):
    layers: list

    def __init__(self, key, activation=jax.nn.relu):
        k1, k2, k3, k4 = jax.random.split(key, 4)
        self.activation = activation
        self.layers = [Linear(3, 4, key=k1),
                       Linear(4, 4, key=k2),
                       Linear(4, 4, key=k3),
                       Linear(4, 1, key=k4)]

    def __call__(self, x):
        for layer in self.layers:
            x = self.activation(layer(x))
        return x
```

Figure A.2: A MLP using Equinox.

The MLP described in Example 2.1.1 could then be implemented as in Figure A.2. Jax only applies operations to functions with a single output. To operate on a batch of data, which usually requires a for loop, we can use the `vmap` function. Consider, for example, the loss function. The model predicts one output y given $\boldsymbol{x}$. To apply it over a batch of $\{\boldsymbol{x}_i\}_{i=1}^m$, we then use `vmap` and compute the loss elementwise with, e.g., `mean_squared_error`.

```
import jax.numpy as jnp

def mean_squared_error(y, pred_y):
    return jnp.mean((y - pred_y) ** 2)

@eqx.filter_jit
def loss_fn(model, x, y):
    pred_y = jax.vmap(model)(x)
    return mean_squared_error(y, pred_y)
```

Figure A.3: Loss function.

The DML training loop can now be transcribed almost literally from the algorithm Algorithm 5.1. We first initialize the optimizer by selecting the parameters $\boldsymbol{\vartheta}$ to consider. A filter operation in Equinox simply extracts all the arrays by default as this almost always corresponds to the learning parameters. Then, `n_epochs` are performed using the training sampler $\mathcal{S}$, here

`train_ds`. We then perform a training step with `train_step` using the differential loss function `dml_loss_fn` to compute the minibatch gradients `grads` and use the optimizer `optim` to compute the parameter updates.

```
def dml_train(model, train_ds, test_ds, optim, n_epochs):

  @eqx.filter_jit
  def train_step(model, opt_state, x, y, dydx):
    loss, grads = eqx.filter_value_and_grad(dml_loss_fn)(model, x, y, dydx)
    updates, opt_state = optim.update(grads, opt_state)
    model = eqx.apply_updates(model, updates)
    return model, opt_state, loss

  opt_state = optim.init(eqx.filter(model, eqx.is_array))
  for epoch in range(n_epochs):
    for (x, y, dydx) in train_ds:
      model, opt_state, loss = train_step(model, opt_state, x, y, dydx)
    test_loss = evaluate(model, test_ds)

  return model
```

Figure A.4: DML training loop.

The differential loss function uses loss balancing as specified in Equation 6.10. The `model_fn` function can be viewed as $\boldsymbol{x} \mapsto (f_\vartheta(\boldsymbol{x}), \nabla_{\boldsymbol{x}} f_\vartheta(\boldsymbol{x}))$, efficiently computing the value and gradient using reverse-mode AD.

```
def dml_loss_fn(model, x, y, dydx, alpha=1):
    n = x.shape[1]
    lambda_0 = 1.0/(1.0 + alpha * n)
    beta = (alpha * n)/(1.0 + alpha * n)
    model_fn = eqx.filter_value_and_grad(model)
    y_pred, dydx_pred = vmap(model_fn)(x)

    value_loss = lambda_0 * mean_squared_error(y_pred, y)
    grad_loss  = lambda_1  * mean_squared_error(dydx_pred, dydx)
    loss = value_loss + grad_loss
    return loss
```

Figure A.5: The differential loss function.

By specifying a training and test dataset generator (for evaluation) we can already learn using the following example setup.

```
key = jax.random.PRNGKey(seed=42)
model = eqx.MLP(key, in_size=x_train.shape[1], out_size="scalar")
optim = optax.adam(learning_rate=0.01)
model = dml_train(model, train_ds, test_ds, optim, n_epochs=100)
```

Figure A.6: DML training setup.

This already covers almost everything there is to know for DML. We will not outline second-order DML as it is more involved. Instead, we refer to the online code repository. Finally, we highlight how the HVP operator $\partial \nabla_{\boldsymbol{x}}(\cdot)$ maps naturally to concepts in JAX. Consider the following implementation of the HVP.

```
def hvp(f, primals, tangents):
    return jax.jvp(lambda x: jax.grad(f)(x), primals, tangents)[1]
```

Figure A.7: Implementation of the HVP in JAX given function $f : \mathbb{R}^n \to \mathbb{R}$.

As we can see, the ∇ operator for the gradient directly maps to the `jax.grad` function which is internally implemented using VJPs. The ∂ function corresponds to the JVP and can directly be implemented using `jax.jvp`. To apply multiple tangent vectors at once via vectorization, as we would like to do for computing the HVPs given the principal component vectors, consider the Hessian Matrix Product (HMP). It can be implemented in JAX by the following.

```
def hmp(f, primals):

    def hvp_(tangents):
        return hvp(f, (primals,), (tangents, ))

    return jax.vmap(hvp_)
```

Figure A.8: Implementation of the HMP in JAX given function $f : \mathbb{R}^n \to \mathbb{R}$.

So, we return a new function that could now be applied to a matrix of principal component vectors given the same input position. With that, we have all the required foundations covered.

Acronyms

AD	Algorithmic Differentiation
AST	Abstract Syntax Tree
CDF	Cumulative Distribution Function
DAG	Directed Acyclic Graph
DML	Differential Machine Learning
GPU	Graphics Processing Unit
HVP	Hessian-Vector Product
JIT	Just-In-Time (compilation)
JVP	Jacobian-Vector Product
MC	Monte Carlo
ML	Machine Learning
MLP	Multi-Layer Perceptron
PCA	Principal component analysis
PDE	Partial Differential Equation
PDF	Probability Density Function
PINNs	Physics-Informed Neural Networks
ReLU	Rectified Linear Unit
SAC	Single Assignment Code
SDE	Stochastic Differential Equation
SGD	Stochastic Gradient Descent
SVD	Singular value decomposition
TPU	Tensor Processing Unit
VJP	Vector-Jacobian Product

Notation

m	The batch size or number of samples.
$\boldsymbol{e}_i$	The i$^{\text{th}}$ Cartesian basis vector.
$\mathcal{J}$	The cost function.
$\mathcal{D}$	The data distribution.
$\odot$	Elementwise multiplication.
$\mathbb{E}[x]$	The expectation of x.
$\circ$	The function composition operator.
$\mathbb{1}_p$	The indicator function with value 1 if condition p is true, else 0.
n_i	The width of the i$^{\text{th}}$ layer.
$\mathcal{L}$	The loss function.
$\lVert\cdot\rVert_p$	The L^p norm.
$N(\mu, \sigma^2)$	The normal distribution with mean μ and variance σ^2.
$U(a, b)$	The uniform distribution in the support range [a, b].
V	The true value of the option price.
ν	The payoff function.
$\mathbb{R}$	The set of real numbers.
$\boldsymbol{\theta}$	The reference model parameters.
Φ	The standard normal cumulative distribution function.
φ	The standard normal probability density function.
$\boldsymbol{\vartheta}$	The surrogate model parameters.
$\mathcal{S}$	The training distribution.
$\cdot^{\intercal}$	The transpose operator.
W_t	A sample of the Wiener process at t.

Bibliography

1. L. Bachelier. "Théorie de la spéculation". *Annales scientifiques de l'École Normale Supérieure, Serie 3, Volume 17*, 1900, pp. 21–86. DOI: 10.24033/asens.476.
2. A. Baydin et al. "Automatic differentiation in machine learning: A survey". *Journal of Machine Learning Research*, 2018. arXiv: 1502.05767.
3. R. Bischof and M. Kraus. "Multi-objective loss balancing for physics-informed deep learning", 2021. arXiv: 2110.09813.
4. C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN: 0387310738. URL: https://www.microsoft.com/en-us/research/people/cmbishop/prml-book/.
5. J. Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. 2018. URL: https://github.com/google/jax.
6. M. Broadie and P. Glasserman. "Estimating security price derivatives using simulation". *Management science* 2, 1996, pp. 269–285. DOI: 10.1287/mnsc.42.2.269.
7. S. L. Brunton and J. N. Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019, pp. 21–23. DOI: 10.1017/9781108380690.002.
8. J. Choi et al. "A Black–Scholes user's guide to the Bachelier model". *Journal of Futures Markets* 5, 2022, pp. 959–980. DOI: 10.1002/fut.22315.
9. S. F. Christodoulou. "Approximation of Expensive Functions through Neural Networks". Presented at the Institute of Software and Tools for Computational Engineering. Bachelor Thesis. RWTH Aachen, 2020.
10. W. M. Czarnecki et al. "Sobolev training for neural networks". *Advances in neural information processing systems*, 2017. arXiv: 1706.04859.
11. A. Défossez et al. "A Simple Convergence Proof of Adam and Adagrad". *Transactions on Machine Learning Research*, 2022. arXiv: 2003.02395.

12. J. Duchi, E. Hazan, and Y. Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". *Journal of Machine Learning Research* 61, 2011, pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html.

13. M. B. Giles. "Vibrato monte carlo sensitivities". *Monte Carlo and Quasi-Monte Carlo Methods 2008*. 2009, pp. 369–382. DOI: 10.1007/978-3-642-04107-5_23.

14. M. B. Giles. "Multilevel Monte Carlo Path Simulation". *Operations Research* 3, 2008, pp. 607–617. DOI: 10.1287/opre.1070.0496.

15. M. Giles and P. Glasserman. "Smoking adjoints: Fast monte carlo greeks". *Risk* 1, 2006, pp. 88–92. URL: https://ora.ox.ac.uk/objects/uuid:f536b1a8-c988-4bda-9c5f-a322652139fd.

16. P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer New York, NY, 2003. DOI: 10.1007/978-0-387-21617-1.

17. I. Goodfellow et al. *Deep learning*. Vol. 1. MIT Press, 2016.

18. L. Hascoet and V. Pascual. "The Tapenade automatic differentiation tool: principles, model, and specification". *ACM Transactions on Mathematical Software (TOMS)* 3, 2013, pp. 1–43. DOI: 10.1145/2450153.2450158.

19. S. L. Heston. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". *The Review of Financial Studies* 2, 2015, pp. 327–343. DOI: 10.1093/rfs/6.2.327.

20. A. A. Heydari, C. A. Thompson, and A. Mehmood. "SoftAdapt: Techniques for Adaptive Loss Weighting of Neural Networks with Multi-Part Loss Functions", 2019. arXiv: 1912.12355.

21. B. N. Huge and A. Savine. "Differential Machine Learning", 2020. arXiv: 2005.02347.

22. J. Jonker and O. Gelderblom. "Amsterdam as the Cradle of Modern Futures Trading and Options Trading, 1550-1650". 2005, pp. 189–206.

23. C. Käbe, J. H. Maruhn, and E. W. Sachs. "Adjoint-based Monte Carlo calibration of financial market models". *Finance and Stochastics*, 2009, pp. 351–379. DOI: 10.1007/s00780-009-0097-9.

24. P. Kidger. "On Neural Differential Equations". PhD thesis. University of Oxford, 2021. arXiv: 2202.02435.

25. P. Kidger and T. Lyons. "Universal Approximation with Deep Narrow Networks". *Proceedings of Thirty Third Conference on Learning Theory*. 2020, pp. 2306–2327. URL: https://proceedings.mlr.press/v125/kidger20a.html.

26. P. Kidger et al. "Neural SDEs as Infinite-Dimensional GANs". *Proceedings of the 38th International Conference on Machine Learning*. 2021, pp. 5453–5463. arXiv: 2102.03657.
27. D. P. Kingma and M. Welling. "Auto-Encoding Variational Bayes". *International Conference on Learning Representations*, 2014. arXiv: 1312.6114.
28. D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *3rd International Conference on Learning Representations*. 2015. arXiv: 1412.6980.
29. A. Krishnapriyan et al. "Characterizing possible failure modes in physics-informed neural networks". *Advances in Neural Information Processing Systems*. 2021, pp. 26548–26560. arXiv: 2109.01050.
30. X. Li et al. "Scalable Gradients and Variational Inference for Stochastic Differential Equations". *Proceedings of The 2nd Symposium on Advances in Approximate Bayesian Inference*. 2020, pp. 1–28. URL: https://proceedings.mlr.press/v118/li20a.html.
31. D. C. Liu and J. Nocedal. "On the limited memory BFGS method for large scale optimization". *Mathematical programming* 1-3, 1989, pp. 503–528. DOI: 10.1007/BF01589116.
32. F. Longstaff and E. Schwartz. "Valuing American Options by Simulation: A Simple Least-Squares Approach". *Review of Financial Studies*, 2001, pp. 113–47. DOI: 10.1093/rfs/14.1.113.
33. I. Loshchilov and F. Hutter. "Decoupled Weight Decay Regularization". *International Conference on Learning Representations*. 2019. arXiv: 1711.05101.
34. J. Martens, I. Sutskever, and K. Swersky. "Estimating the Hessian by Back-propagating Curvature". *Proceedings of the 29th International Conference on Machine Learning*, 2012. arXiv: 1206.6464.
35. K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012. URL: https://probml.github.io/pml-book/book0.html.
36. U. Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. SIAM, 2011.
37. G. C. Pflug. *Optimization of Stochastic Models*. Springer New York, NY, 1996. DOI: 10.1007/978-1-4613-1449-3.
38. A. Pinkus. "Approximation theory of the MLP model in neural networks". *Acta numerica*, 1999, pp. 143–195. DOI: 10.1017/S0962492900002919.
39. A. K. Polala and B. Hientzsch. "Parametric Differential Machine Learning for Pricing and Calibration", 2023. arXiv: 2302.06682.

40. M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". *Journal of Computational Physics*, 2019, pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045.

41. M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations", 2017. arXiv: 1711.10561.

42. M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations", 2017. arXiv: 1711.10566.

43. S. J. Reddi, S. Kale, and S. Kumar. "On the Convergence of Adam and Beyond". *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=ryQu7f-RZ.

44. S. Rifai et al. "Contractive auto-encoders: Explicit invariance during feature extraction". *Proceedings of the 28th international conference on international conference on machine learning*. 2011, pp. 833–840. URL: https://icml.cc/2011/papers/455_icmlpaper.pdf.

45. W. Schachermayer and J. Teichmann. "How close are the option pricing formulas of Bachelier and Black–Merton–Scholes?" *Mathematical Finance: an international journal of mathematics, statistics and financial economics* 1, 2008, pp. 155–170. DOI: 10.1111/j.1467-9965.2007.00326.x.

46. S. L. Smith et al. "On the Origin of Implicit Regularization in Stochastic Gradient Descent". *International Conference on Learning Representations*. 2021. arXiv: 2101.12176. URL: https://openreview.net/forum?id=rq_Qr0c1Hyo.

47. S. Srinivas and F. Fleuret. "Knowledge Transfer with Jacobian Matching". *Proceedings of the 35th International Conference on Machine Learning*. 2018, pp. 4723–4731. URL: https://proceedings.mlr.press/v80/srinivas18a.html.

48. I. Sutskever et al. "On the importance of initialization and momentum in deep learning". *Proceedings of the 30th International Conference on Machine Learning*. 3. Atlanta, Georgia, USA, 2013, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

49. S. Terakado. "On the option pricing formula based on the bachelier model". *SSRN*, 2019. DOI: 10.2139/ssrn.3428994.

50. T. Tieleman and G. Hinton. "Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude". *Neural Networks for Machine Learning*, 2012, pp. 26–31. URL: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf.

51. S. Vaswani, F. Bach, and M. Schmidt. "Fast and Faster Convergence of SGD for Over Parameterized Models and an Accelerated Perceptron". *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*. 2019, pp. 1195–1204. URL: https://proceedings.mlr.press/v89/vaswani19a.html.

52. J. P. de la Vega and H. Kellenbenz. *Confusion de Confusiones [1688]: Portions Descriptive of the Amsterdam Stock Exchange*. Publication of the Kress Library of Business and Economics, no. 13, 2013. URL: https://gwern.net/doc/economics/1688-delavega-confusionofconfusions.pdf.

53. N. S. Wadia, Y. Dandi, and M. I. Jordan. "A Gentle Introduction to Gradient-Based Optimization and Variational Inequalities for Machine Learning", 2023. arXiv: 2309.04877.

54. L. Yang, D. Zhang, and G. E. Karniadakis. "Physics-Informed Generative Adversarial Networks for Stochastic Differential Equations". *SIAM Journal on Scientific Computing* 1, 2020, pp. 292–317. DOI: 10.1137/18M1225409.

55. J. Yu et al. "Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems". *Computer Methods in Applied Mechanics and Engineering*, 2022. DOI: 10.1016/j.cma.2022.114823.

56. D. Zhang et al. "Quantifying total uncertainty in physics-informed neural networks for solving forward and inverse stochastic problems". *Journal of Computational Physics*, 2019. DOI: 10.1016/j.jcp.2019.07.048.

57. J. Zhuang et al. "Adabelief optimizer: Adapting stepsizes by the belief in observed gradients". *Advances in neural information processing systems*, 2020, pp. 18795–18806. arXiv: 2010.07468.