Matthew F. Dixon
Igor Halperin
Paul Bilokon

Machine Learning in Finance

From Theory to Practice

Springer

Machine Learning in Finance

Matthew F. Dixon • Igor Halperin • Paul Bilokon

Machine Learning in Finance

From Theory to Practice

Springer

Matthew F. Dixon
Department of Applied Mathematics
Illinois Institute of Technology
Chicago, IL, USA

Igor Halperin
Tandon School of Engineering
New York University
Brooklyn, NY, USA

Paul Bilokon
Department of Mathematics
Imperial College London
London, UK

Additional material to this book can be downloaded from http://mypages.iit.edu/~mdixon7/book/ML_Finance_Codes-Book.zip

ISBN 978-3-030-41067-4 ISBN 978-3-030-41068-1 (eBook)
https://doi.org/10.1007/978-3-030-41068-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth.

—Arthur Conan Doyle

Introduction

Machine learning in finance sits at the intersection of a number of emergent and established disciplines including pattern recognition, financial econometrics, statistical computing, probabilistic programming, and dynamic programming. With the trend towards increasing computational resources and larger datasets, machine learning has grown into a central computational engineering field, with an emphasis placed on plug-and-play algorithms made available through open-source machine learning toolkits. Algorithm focused areas of finance, such as algorithmic trading have been the primary adopters of this technology. But outside of engineering-based research groups and business activities, much of the field remains a mystery.

A key barrier to understanding machine learning for non-engineering students and practitioners is the absence of the well-established theories and concepts that financial time series analysis equips us with. These serve as the basis for the development of financial modeling intuition and scientific reasoning. Moreover, machine learning is heavily entrenched in engineering ontology, which makes developments in the field somewhat intellectually inaccessible for students, academics, and finance practitioners from the quantitative disciplines such as mathematics, statistics, physics, and economics. Consequently, there is a great deal of misconception and limited understanding of the capacity of this field. While machine learning techniques are often effective, they remain poorly understood and are often mathematically indefensible. How do we place key concepts in the field of machine learning in the context of more foundational theory in time series analysis, econometrics, and mathematical statistics? Under which simplifying conditions are advanced machine learning techniques such as deep neural networks mathematically equivalent to well-known statistical models such as linear regression? How should we reason about the perceived benefits of using advanced machine learning methods over more traditional econometrics methods, for different financial applications? What theory supports the application of machine learning to problems in financial modeling? How does reinforcement learning provide a model-free approach to the Black–Scholes–Merton model for derivative pricing? How does Q-learning generalize discrete-time stochastic control problems in finance?

This book is written for advanced graduate students and academics in financial econometrics, management science, and applied statistics, in addition to quants and data scientists in the field of quantitative finance. We present machine learning as a non-linear extension of various topics in quantitative economics such as financial econometrics and dynamic programming, with an emphasis on novel algorithmic representations of data, regularization, and techniques for controlling the bias-variance tradeoff leading to improved out-of-sample forecasting. The book is presented in three parts, each part covering theory and applications. The first part presents supervised learning for cross-sectional data from both a Bayesian and frequentist perspective. The more advanced material places a firm emphasis on neural networks, including deep learning, as well as Gaussian processes, with examples in investment management and derivatives. The second part covers supervised learning for time series data, arguably the most common data type used in finance with examples in trading, stochastic volatility, and fixed income modeling. Finally, the third part covers reinforcement learning and its applications in trading, investment, and wealth management. We provide Python code examples to support the readers' understanding of the methodologies and applications. As a bridge to research in this emergent field, we present the frontiers of machine learning in finance from a researcher's perspective, highlighting how many well-known concepts in statistical physics are likely to emerge as research topics for machine learning in finance.

Prerequisites

This book is targeted at graduate students in data science, mathematical finance, financial engineering, and operations research seeking a career in quantitative finance, data science, analytics, and fintech. Students are expected to have completed upper section undergraduate courses in linear algebra, multivariate calculus, advanced probability theory and stochastic processes, statistics for time series (econometrics), and gained some basic introduction to numerical optimization and computational mathematics. Students shall find the later chapters of this book, on reinforcement learning, more accessible with some background in investment science. Students should also have prior experience with Python programming and, ideally, taken a course in computational finance and introductory machine learning. The material in this book is more mathematical and less engineering focused than most courses on machine learning, and for this reason we recommend reviewing the recent book, *Linear Algebra and Learning from Data* by Gilbert Strang as background reading.

Advantages of the Book

Readers will find this book useful as a bridge from well-established foundational topics in financial econometrics to applications of machine learning in finance. Statistical machine learning is presented as a non-parametric extension of financial econometrics and quantitative finance, with an emphasis on novel algorithmic representations of data, regularization, and model averaging to improve out-of-sample forecasting. The key distinguishing feature from classical financial econometrics and dynamic programming is the absence of an assumption on the data generation process. This has important implications for modeling and performance assessment which are emphasized with examples throughout the book. Some of the main contributions of the book are as follows:

- The textbook market is saturated with excellent books on machine learning. However, few present the topic from the prospective of financial econometrics and cast fundamental concepts in machine learning into canonical modeling and decision frameworks already well established in finance such as financial time series analysis, investment science, and financial risk management. Only through the integration of these disciplines can we develop an intuition into how machine learning theory informs the practice of financial modeling.
- Machine learning is entrenched in engineering ontology, which makes developments in the field somewhat intellectually inaccessible for students, academics, and finance practitioners from quantitative disciplines such as mathematics, statistics, physics, and economics. Moreover, financial econometrics has not kept pace with this transformative field, and there is a need to reconcile various modeling concepts between these disciplines. This textbook is built around powerful mathematical ideas that shall serve as the basis for a graduate course for students with prior training in probability and advanced statistics, linear algebra, times series analysis, and Python programming.
- This book provides financial market motivated and compact theoretical treatment of financial modeling with machine learning for the benefit of regulators, wealth managers, federal research agencies, and professionals in other heavily regulated business functions in finance who seek a more theoretical exposition to allay concerns about the "black-box" nature of machine learning.
- Reinforcement learning is presented as a model-free framework for stochastic control problems in finance, covering portfolio optimization, derivative pricing, and wealth management applications without assuming a data generation process. We also provide a model-free approach to problems in market microstructure, such as optimal execution, with Q-learning. Furthermore, our book is the first to present on methods of inverse reinforcement learning.
- Multiple-choice questions, numerical examples, and more than 80 end-of-chapter exercises are used throughout the book to reinforce key technical concepts.

- This book provides Python codes demonstrating the application of machine learning to algorithmic trading and financial modeling in risk management and equity research. These codes make use of powerful open-source software toolkits such as Google's TensorFlow and Pandas, a data processing environment for Python.

Overview of the Book

Chapter 1

Chapter 1 provides the industry context for machine learning in finance, discussing the critical events that have shaped the finance industry's need for machine learning and the unique barriers to adoption. The finance industry has adopted machine learning to varying degrees of sophistication. How it has been adopted is heavily fragmented by the academic disciplines underpinning the applications. We view some key mathematical examples that demonstrate the nature of machine learning and how it is used in practice, with the focus on building intuition for more technical expositions in later chapters. In particular, we begin to address many finance practitioner's concerns that neural networks are a "black-box" by showing how they are related to existing well-established techniques such as linear regression, logistic regression, and autoregressive time series models. Such arguments are developed further in later chapters.

Chapter 2

Chapter 2 introduces probabilistic modeling and reviews foundational concepts in Bayesian econometrics such as Bayesian inference, model selection, online learning, and Bayesian model averaging. We develop more versatile representations of complex data with probabilistic graphical models such as mixture models.

Chapter 3

Chapter 3 introduces Bayesian regression and shows how it extends many of the concepts in the previous chapter. We develop kernel-based machine learning methods—specifically Gaussian process regression, an important class of Bayesian machine learning methods—and demonstrate their application to "surrogate" models of derivative prices. This chapter also provides a natural point from which to

develop intuition for the role and functional form of regularization in a frequentist setting—the subject of subsequent chapters.

Chapter 4

Chapter 4 provides a more in-depth description of supervised learning, deep learning, and neural networks—presenting the foundational mathematical and statistical learning concepts and explaining how they relate to real-world examples in trading, risk management, and investment management. These applications present challenges for forecasting and model design and are presented as a reoccurring theme throughout the book. This chapter moves towards a more engineering style exposition of neural networks, applying concepts in the previous chapters to elucidate various model design choices.

Chapter 5

Chapter 5 presents a method for interpreting neural networks which imposes minimal restrictions on the neural network design. The chapter demonstrates techniques for interpreting a feedforward network, including how to rank the importance of the features. In particular, an example demonstrating how to apply interpretability analysis to deep learning models for factor modeling is also presented.

Chapter 6

Chapter 6 provides an overview of the most important modeling concepts in financial econometrics. Such methods form the conceptual basis and performance baseline for more advanced neural network architectures presented in the next chapter. In fact, each type of architecture is a generalization of many of the models presented here. This chapter is especially useful for students from an engineering or science background, with little exposure to econometrics and time series analysis.

Chapter 7

Chapter 7 presents a powerful class of probabilistic models for financial data. Many of these models overcome some of the severe stationarity limitations of the frequentist models in the previous chapters. The fitting procedure demonstrated is also different—the use of Kalman filtering algorithms for state-space models rather

than maximum likelihood estimation or Bayesian inference. Simple examples of hidden Markov models and particle filters in finance and various algorithms are presented.

Chapter 8

Chapter 8 presents various neural network models for financial time series analysis, providing examples of how they relate to well-known techniques in financial econometrics. Recurrent neural networks (RNNs) are presented as non-linear time series models and generalize classical linear time series models such as $AR(p)$. They provide a powerful approach for prediction in financial time series and generalize to non-stationary data. The chapter also presents convolution neural networks for filtering time series data and exploiting different scales in the data. Finally, this chapter demonstrates how autoencoders are used to compress information and generalize principal component analysis.

Chapter 9

Chapter 9 introduces Markov decision processes and the classical methods of dynamic programming, before building familiarity with the ideas of reinforcement learning and other approximate methods for solving MDPs. After describing Bellman optimality and iterative value and policy updates before moving to Q-learning, the chapter quickly advances towards a more engineering style exposition of the topic, covering key computational concepts such as greediness, batch learning, and Q-learning. Through a number of mini-case studies, the chapter provides insight into how RL is applied to optimization problems in asset management and trading. These examples are each supported with Python notebooks.

Chapter 10

Chapter 10 considers real-world applications of reinforcement learning in finance, as well as further advances the theory presented in the previous chapter. We start with one of the most common problems of quantitative finance, which is the problem of optimal portfolio trading in discrete time. Many practical problems of trading or risk management amount to different forms of dynamic portfolio optimization, with different optimization criteria, portfolio composition, and constraints. The chapter introduces a reinforcement learning approach to option pricing that generalizes the classical Black–Scholes model to a data-driven approach using Q-learning. It then presents a probabilistic extension of Q-learning called G-learning and shows how it

can be used for dynamic portfolio optimization. For certain specifications of reward functions, G-learning is semi-analytically tractable and amounts to a probabilistic version of linear quadratic regulators (LQRs). Detailed analyses of such cases are presented and we show their solutions with examples from problems of dynamic portfolio optimization and wealth management.

Chapter 11

Chapter 11 provides an overview of the most popular methods of inverse reinforcement learning (IRL) and imitation learning (IL). These methods solve the problem of optimal control in a data-driven way, similarly to reinforcement learning, however with the critical difference that now rewards are *not* observed. The problem is rather to learn the reward function from the observed behavior of an agent. As behavioral data without rewards are widely available, the problem of learning from such data is certainly very interesting. The chapter provides a moderate-level description of the most promising IRL methods, equips the reader with sufficient knowledge to understand and follow the current literature on IRL, and presents examples that use simple simulated environments to see how these methods perform when we know the "ground truth" rewards. We then present use cases for IRL in quantitative finance that include applications to trading strategy identification, sentiment-based trading, option pricing, inference of portfolio investors, and market modeling.

Chapter 12

Chapter 12 takes us forward to emerging research topics in quantitative finance and machine learning. Among many interesting emerging topics, we focus here on two broad themes. The first one deals with unification of supervised learning and reinforcement learning as two tasks of perception-action cycles of agents. We outline some recent research ideas in the literature including in particular information theory-based versions of reinforcement learning and discuss their relevance for financial applications. We explain why these ideas might have interesting practical implications for RL financial models, where feature selection could be done within the general task of optimization of a long-term objective, rather than outside of it, as is usually performed in "alpha-research."

The second topic presented in this chapter deals with using methods of reinforcement learning to construct models of market dynamics. We also introduce some advanced physics-based approaches for computations for such RL-inspired market models.

Source Code

Many of the chapters are accompanied by Python notebooks to illustrate some of the main concepts and demonstrate application of machine learning methods. Each notebook is lightly annotated. Many of these notebooks use TensorFlow. We recommend loading these notebooks, together with any accompanying Python source files and data, in Google Colab. Please see the appendices of each chapter accompanied by notebooks, and the `README.md` in the subfolder of each chapter, for further instructions and details.

Scope

We recognize that the field of machine learning is developing rapidly and to keep abreast of the research in this field is a challenging pursuit. Machine learning is an umbrella term for a number of methodology classes, including supervised learning, unsupervised learning, and reinforcement learning. This book focuses on supervised learning and reinforcement learning because these are the areas with the most overlap with econometrics, predictive modeling, and optimal control in finance. Supervised machine learning can be categorized as generative and discriminative. Our focus is on discriminative learners which attempt to partition the input space, either directly, through affine transformations or through projections onto a manifold. Neural networks have been shown to provide a universal approximation to a wide class of functions. Moreover, they can be shown to reduce to other well-known statistical techniques and are adaptable to time series data.

Extending time series models, a number of chapters in this book are devoted to an introduction to reinforcement learning (RL) and inverse reinforcement learning (IRL) that deal with problems of optimal control of such time series and show how many classical financial problems such as portfolio optimization, option pricing, and wealth management can naturally be posed as problems for RL and IRL. We present simple RL methods that can be applied for these problems, as well as explain how neural networks can be used in these applications.

There are already several excellent textbooks covering other classical machine learning methods, and we instead choose to focus on how to cast machine learning into various financial modeling and decision frameworks. We emphasize that much of this material is not unique to neural networks, but comparisons of alternative supervised learning approaches, such as random forests, are beyond the scope of this book.

Multiple-Choice Questions

Multiple-choice questions are included after introducing a key concept. The correct answers to all questions are provided at the end of each chapter with selected, partial, explanations to some of the more challenging material.

Exercises

The exercises that appear at the end of every chapter form an important component of the book. Each exercise has been chosen to reinforce concepts explained in the text, to stimulate the application of machine learning in finance, and to gently bridge material in other chapters. It is graded according to difficulty ranging from (*), which denotes a simple exercise which might take a few minutes to complete, through to (***), which denotes a significantly more complex exercise. Unless specified otherwise, all equations referenced in each exercise correspond to those in the corresponding chapter.

Instructor Materials

The book is supplemented by a separate Instructor's Manual which provides worked solutions to the end of chapter questions. Full explanations for the solutions to the multiple-choice questions are also provided. The manual provides additional notes and example code solutions for some of the programming exercises in the later chapters.

Acknowledgements

This book is dedicated to the late Mark Davis (Imperial College) who was an inspiration in the field of mathematical finance and engineering, and formative in our careers. Peter Carr, Chair of the Department of Financial Engineering at NYU Tandon, has been instrumental in supporting the growth of the field of machine learning in finance. Through providing speaker engagements and machine learning instructorship positions in the MS in Algorithmic Finance Program, the authors have been able to write research papers and identify the key areas required by a text book. Miquel Alonso (AIFI), Agostino Capponi (Columbia), Rama Cont (Oxford), Kay Giesecke (Stanford), Ali Hirsa (Columbia), Sebastian Jaimungal (University of Toronto), Gary Kazantsev (Bloomberg), Morton Lane (UIUC), Jörg Osterrieder (ZHAW) have established various academic and joint academic-industry workshops

and community meetings to proliferate the field and serve as input for this book. At the same time, there has been growing support for the development of a book in London, where several SIAM/LMS workshops and practitioner special interest groups, such as the Thalesians, have identified a number of compelling financial applications. The material has grown from courses and invited lectures at NYU, UIUC, Illinois Tech, Imperial College and the 2019 Bootcamp on Machine Learning in Finance at the Fields Institute, Toronto.

Along the way, we have been fortunate to receive the support of Tomasz Bielecki (Illinois Tech), Igor Cialenco (Illinois Tech), Ali Hirsa (Columbia University), and Brian Peterson (DV Trading). Special thanks to research collaborators and colleagues Kay Giesecke (Stanford University), Diego Klabjan (NWU), Nick Polson (Chicago Booth), and Harvey Stein (Bloomberg), all of whom have shaped our understanding of the emerging field of machine learning in finance and the many practical challenges. We are indebted to Sri Krishnamurthy (QuantUniversity), Saeed Amen (Cuemacro), Tyler Ward (Google), and Nicole Königstein for their valuable input on this book. We acknowledge the support of a number of Illinois Tech graduate students who have contributed to the source code examples and exercises: Xiwen Jing, Bo Wang, and Siliang Xong. Special thanks to Swaminathan Sethuraman for his support of the code development, to Volod Chernat and George Gvishiani who provided support and code development for the course taught at NYU and Coursera. Finally, we would like to thank the students and especially the organisers of the MSc Finance and Mathematics course at Imperial College, where many of the ideas presented in this book have been tested: Damiano Brigo, Antoine (Jack) Jacquier, Mikko Pakkanen, and Rula Murtada. We would also like to thank Blanka Horvath for many useful suggestions.

Chicago, IL, USA — Matthew F. Dixon
Brooklyn, NY, USA — Igor Halperin
London, UK — Paul Bilokon
December 2019

Contents

About the Authors

Matthew F. Dixon is an Assistant Professor of Applied Math at the Illinois Institute of Technology. His research in computational methods for finance is funded by Intel. Matthew began his career in structured credit trading at Lehman Brothers in London before pursuing academics and consulting for financial institutions in quantitative trading and risk modeling. He holds a Ph.D. in Applied Mathematics from Imperial College (2007) and has held postdoctoral and visiting professor appointments at Stanford University and UC Davis, respectively. He has published over 20 peer-reviewed publications on machine learning and financial modeling, has been cited in Bloomberg Markets and the Financial Times as an AI in fintech expert, and is a frequently invited speaker in Silicon Valley and on Wall Street. He has published R packages, served as a Google Summer of Code mentor, and is the co-founder of the Thalesians Ltd.

Igor Halperin is a Research Professor in Financial Engineering at NYU and an AI Research Associate at Fidelity Investments. He was previously an Executive Director of Quantitative Research at JPMorgan for nearly 15 years. Igor holds a Ph.D. in Theoretical Physics from Tel Aviv University (1994). Prior to joining the financial industry, he held postdoctoral positions in theoretical physics at the Technion and the University of British Columbia.

Paul Bilokon is CEO and Founder of Thalesians Ltd. and an expert in electronic and algorithmic trading across multiple asset classes, having helped build such businesses at Deutsche Bank and Citigroup. Before focusing on electronic trading, Paul worked on derivatives and has served in quantitative roles at Nomura, Lehman Brothers, and Morgan Stanley. Paul has been educated at Christ Church College, Oxford, and Imperial College. Apart from mathematical and computational finance, his academic interests include machine learning and mathematical logic.

Part I
Machine Learning with Cross-Sectional Data

Chapter 1
Introduction

This chapter introduces the industry context for machine learning in finance, discussing the critical events that have shaped the finance industry's need for machine learning and the unique barriers to adoption. The finance industry has adopted machine learning to varying degrees of sophistication. How it has been adopted is heavily fragmented by the academic disciplines underpinning the applications. We view some key mathematical examples that demonstrate the nature of machine learning and how it is used in practice, with the focus on building intuition for more technical expositions in later chapters. In particular, we begin to address many finance practitioner's concerns that neural networks are a "black-box" by showing how they are related to existing well-established techniques such as linear regression, logistic regression, and autoregressive time series models. Such arguments are developed further in later chapters. This chapter also introduces reinforcement learning for finance and is followed by more in-depth case studies highlighting the design concepts and practical challenges of applying machine learning in practice.

1 Background

In 1955, John McCarthy, then a young Assistant Professor of Mathematics, at Dartmouth College in Hanover, New Hampshire, submitted a proposal with Marvin Minsky, Nathaniel Rochester, and Claude Shannon for the Dartmouth Summer Research Project on Artificial Intelligence (McCarthy et al. 1955). These organizers were joined in the summer of 1956 by Trenchard More, Oliver Selfridge, Herbert Simon, Ray Solomonoff, among others. The stated goal was ambitious:

"The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_1

find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves." Thus the field of artificial intelligence, or AI, was born.

Since this time, AI has perpetually strived to outperform humans on various judgment tasks (Pinar Saygin et al. 2000). The most fundamental metric for this success is the Turing test—a test of a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human (Turing 1995). In recent years, a pattern of success in AI has emerged—one in which machines outperform in the presence of a large number of decision variables, usually with the best solution being found through evaluating an exponential number of candidates in a constrained high-dimensional space. Deep learning models, in particular, have proven remarkably successful in a wide field of applications (DeepMind 2016; Kubota 2017; Esteva et al. 2017) including image processing (Simonyan and Zisserman 2014), learning in games (DeepMind 2017), neuroscience (Poggio 2016), energy conservation (DeepMind 2016), skin cancer diagnostics (Kubota 2017; Esteva et al. 2017).

One popular account of this reasoning points to humans' perceived inability to process large amounts of information and make decisions beyond a few key variables. But this view, even if fractionally representative of the field, does no justice to AI or human learning. Humans are not being replaced any time soon. The median estimate for human intelligence in terms of gigaflops is about 10^4 times more than the machine that ran alpha-go. Of course, this figure is caveated on the important question of whether the human mind is even a Turing machine.

1.1 Big Data—Big Compute in Finance

The growth of machine-readable data to record and communicate activities throughout the financial system combined with persistent growth in computing power and storage capacity has significant implications for every corner of financial modeling. Since the financial crises of 2007–2008, regulatory supervisors have reoriented towards "data-driven" regulation, a prominent example of which is the collection and analysis of detailed contractual terms for the bank loan and trading book stress-testing programs in the USA and Europe, instigated by the crisis (Flood et al. 2016).

"Alternative data"—which refers to data and information outside of the usual scope of securities pricing, company fundamentals, or macroeconomic indicators—is playing an increasingly important role for asset managers, traders, and decision makers. Social media is now ranked as one of the top categories of alternative data currently used by hedge funds. Trading firms are hiring experts in machine learning with the ability to apply natural language processing (NLP) to financial news and other unstructured documents such as earnings announcement reports and SEC 10K reports. Data vendors such as Bloomberg, Thomson Reuters, and RavenPack are providing processed news sentiment data tailored for systematic trading models.

In de Prado (2019), some of the properties of these new, alternative datasets are explored: (a) many of these datasets are unstructured, non-numerical, and/or non-categorical, like news articles, voice recordings, or satellite images; (b) they tend to be high-dimensional (e.g., credit card transactions) and the number of variables may greatly exceed the number of observations; (c) such datasets are often sparse, containing NaNs (not-a-numbers); (d) they may implicitly contain information about networks of agents.

Furthermore, de Prado (2019) explains why classical econometric methods fail on such datasets. These methods are often based on linear algebra, which fail when the number of variables exceeds the number of observations. Geometric objects, such as covariance matrices, fail to recognize the topological relationships that characterize networks. On the other hand, machine learning techniques offer the numerical power and functional flexibility needed to identify complex patterns in a high-dimensional space offering a significant improvement over econometric methods.

The "black-box" view of ML is dismissed in de Prado (2019) as a misconception. Recent advances in ML make it applicable to the evaluation of plausibility of scientific theories; determination of the relative informational variables (usually referred to as features in ML) for explanatory and/or predictive purposes; causal inference; and visualization of large, high-dimensional, complex datasets.

Advances in ML remedy the shortcomings of econometric methods in goal setting, outlier detection, feature extraction, regression, and classification when it comes to modern, complex alternative datasets. For example, in the presence of p features there may be up to $2^p - p - 1$ multiplicative interaction effects. For two features there is only one such interaction effect, x_1x_2. For three features, there are $x_1x_2, x_1x_3, x_2x_3, x_1x_2x_3$. For as few as ten features, there are 1,013 multiplicative interaction effects. Unlike ML algorithms, econometric models do not "learn" the structure of the data. The model specification may easily miss some of the interaction effects. The consequences of missing an interaction effect, e.g. fitting $y_t = x_{1,t} + x_{2,t} + \epsilon_t$ instead of $y_t = x_{1,t} + x_{2,t} + x_{1,t}x_{2,t} + \epsilon_t$, can be dramatic. A machine learning algorithm, such as a decision tree, will recursively partition a dataset with complex patterns into subsets with simple patterns, which can then be fit independently with simple linear specifications. Unlike the classical linear regression, this algorithm "learns" about the existence of the $x_{1,t}x_{2,t}$ effect, yielding much better out-of-sample results.

There is a draw towards more empirically driven modeling in asset pricing research—using ever richer sets of firm characteristics and "factors" to describe and understand differences in expected returns across assets and model the dynamics of the aggregate market equity risk premium (Gu et al. 2018). For example, Harvey et al. (2016) study 316 "factors," which include firm characteristics and common factors, for describing stock return behavior. Measurement of an asset's risk premium is fundamentally a problem of prediction—the risk premium is the conditional expectation of a future realized excess return. Methodologies that can reliably attribute excess returns to tradable anomalies are highly prized. Machine learning provides a non-linear empirical approach for modeling realized security

returns from firm characteristics. Dixon and Polson (2019) review the formulation of asset pricing models for measuring asset risk premia and cast neural networks in canonical asset pricing frameworks.

1.2 Fintech

The rise of data and machine learning has led to a "fintech" industry, covering digital innovations and technology-enabled business model innovations in the financial sector (Philippon 2016). Examples of innovations that are central to fintech today include cryptocurrencies and the blockchain, new digital advisory and trading systems, peer-to-peer lending, equity crowdfunding, and mobile payment systems. Behavioral prediction is often a critical aspect of product design and risk management needed for consumer-facing business models; consumers or economic agents are presented with well-defined choices but have unknown economic needs and limitations, and in many cases do not behave in a strictly economically rational fashion. Therefore it is necessary to treat parts of the system as a black-box that operates under rules that cannot be known in advance.

1.2.1 Robo-Advisors

Robo-advisors are financial advisors that provide financial advice or portfolio management services with minimal human intervention. The focus has been on portfolio management rather than on estate and retirement planning, although there are exceptions, such as Blooom. Some limit investors to the ETFs selected by the service, others are more flexible. Examples include Betterment, Wealthfront, Wise-Banyan, FutureAdvisor (working with Fidelity and TD Ameritrade), Blooom, Motif Investing, and Personal Capital. The degree of sophistication and the utilization of machine learning are on the rise among robo-advisors.

1.2.2 Fraud Detection

In 2011 fraud cost the financial industry approximately $80 billion annually (Consumer Reports, June 2011). According to PwC's Global Economic Crime Survey 2016, 46% of respondents in the Financial Services industry reported being victims of economic crime in the last 24 months—a small increase from 45% reported in 2014. 16% of those that reported experiencing economic crime had suffered more than 100 incidents, with 6% suffering more than 1,000. According to the survey, the top 5 types of economic crime are asset misappropriation (60%, down from 67% in 2014), cybercrime (49%, up from 39% in 2014), bribery and corruption (18%, down from 20% in 2014), money laundering (24%, as in 2014), and accounting fraud (18%, down from 21% in 2014). Detecting economic crimes is

one of the oldest successful applications of machine learning in the financial services industry. See Gottlieb et al. (2006) for a straightforward overview of some of the classical methods: logistic regression, naïve Bayes, and support vector machines. The rise of electronic trading has led to new kinds of financial fraud and market manipulation. Some exchanges are investigating the use of deep learning to counter spoofing.

1.2.3 Cryptocurrencies

Blockchain technology, first implemented by Satoshi Nakamoto in 2009 as a core component of Bitcoin, is a distributed public ledger recording transactions. Its usage allows secure peer-to-peer communication by linking blocks containing hash pointers to a previous block, a timestamp, and transaction data. Bitcoin is a decentralized digital currency (cryptocurrency) which leverages the blockchain to store transactions in a distributed manner in order to mitigate against flaws in the financial industry.

In contrast to existing financial networks, blockchain based cryptocurrencies expose the entire transaction graph to the public. This openness allows, for example, the most significant agents to be immediately located (pseudonymously) on the network. By processing all financial interactions, we can model the network with a high-fidelity graph, as illustrated in Fig. 1.1 so that it is possible to characterize how the flow of information in the network evolves over time. This novel data representation permits a new form of financial econometrics—with the emphasis on the topological network structures in the microstructure rather than solely the covariance of historical time series of prices. The role of users, entities, and their interactions in formation and dynamics of cryptocurrency risk investment, financial predictive analytics and, more generally, in re-shaping the modern financial world is a novel area of research (Dyhrberg 2016; Gomber et al. 2017; Sovbetov 2018).

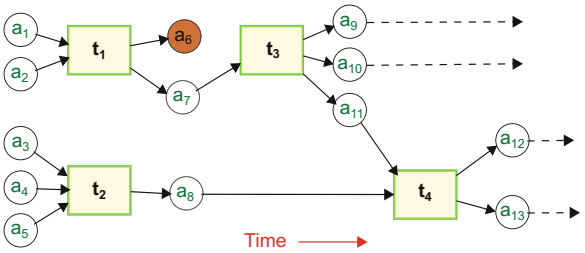


Fig. 1.1 A transaction–address graph representation of the Bitcoin network. Addresses are represented by circles, transactions with rectangles, and edges indicate a transfer of coins. Blocks order transactions in time, whereas each transaction with its input and output nodes represents an immutable decision that is encoded as a subgraph on the Bitcoin network. Source: Akcora et al. (2018)

2 Machine Learning and Prediction

With each passing year, finance becomes increasingly reliant on computational methods. At the same time, the growth of machine-readable data to monitor, record, and communicate activities throughout the financial system has significant implications for how we approach the topic of modeling. One of the reasons that AI and the set of computer algorithms for learning, referred to as "machine learning," have been successful is a result of a number of factors beyond computer hardware and software advances. Machines are able to model complex and high-dimensional data generation processes, sweep through millions of model configurations, and then robustly evaluate and correct the models in response to new information (Dhar 2013). By continuously updating and hosting a number of competing models, they prevent any one model leading us into a data gathering silo effective only for that market view. Structurally, the adoption of ML has even shifted our behavior—the way we reason, experiment, and shape our perspectives from data using ML has led to empirically driven trading and investment decision processes.

Machine learning is a broad area, covering various classes of algorithms for pattern recognition and decision-making. In **supervised learning**, we are given labeled data, i.e. pairs $(x_1, y_1), \ldots, (x_n, y_n)$, $x_1, \ldots, x_n \in X$, $y_1, \ldots, y_n \in Y$, and the goal is to learn the relationship between X and Y. Each observation x_i is referred to as a **feature vector** and y_i is the **label** or **response**. In **unsupervised learning**, we are given unlabeled data, $x_1, x_2, \ldots, x_n$ and our goal is to retrieve exploratory information about the data, perhaps grouping similar observations or capturing some hidden patterns. Unsupervised learning includes **cluster analysis** algorithms such as hierarchical clustering, k-means clustering, self-organizing maps, Gaussian mixture, and hidden Markov models and is commonly referred to as data mining. In both instances, the data could be financial time series, news documents, SEC documents, and textual information on important events. The third type of machine learning paradigm is **reinforcement learning** and is an algorithmic approach for enforcing Bellman optimality of a Markov Decision Process—defining a set of states and actions in response to a changing regime so as to maximize some notion of cumulative reward. In contrast to supervised learning, which just considers a single action at each point in time, reinforcement learning is concerned with the optimal sequence of actions. It is therefore a form of dynamic programming that is used for decisions leading to optimal trade execution, portfolio allocation, and liquidation over a given horizon.

Supervised learning addresses a fundamental prediction problem: Construct a non-linear predictor, $\hat{Y}(X)$, of an output, Y, given a high-dimensional input matrix $X = (X_1, \ldots, X_P)$ of P variables. Machine learning can be simply viewed as the study and construction of an input–output map of the form

$$Y = F(X) \text{ where } X = (X_1, \ldots, X_P).$$

$F(X)$ is sometimes referred to as the "data-feature" map. The output variable, Y, can be continuous, discrete, or mixed. For example, in a classification problem,

$F : X \to G$, where $G \in \mathcal{K} := \{0, \ldots, K-1\}$, K is the number of categories and $\hat{G}$ is the predictor.

Supervised machine learning uses a parameterized[1] model $g(X|\boldsymbol{\theta})$ over independent variables X, to predict the continuous or categorical output Y or G. The model is parameterized by one or more free parameters θ which are fitted to data. Prediction of categorical variables is referred to as *classification* and is common in pattern recognition. The most common approach to predicting categorical variables is to encode the response G as one or more binary values, then treat the model prediction as continuous.

? Multiple Choice Question 1

Select all the following correct statements:

1. Supervised learning involves learning the relationship between input and output variables.
2. Supervised learning requires a human supervisor to prepare labeled training data.
3. Unsupervised learning does not require a human supervisor and is therefore superior to supervised learning.
4. Reinforcement learning can be viewed as a generalization of supervised learning to Markov Decision Processes.

There are two different classes of supervised learning models, *discriminative* and *generative*. A discriminative model learns the decision boundary between the classes and implicitly learns the distribution of the output conditional on the input. A generative model explicitly learns the joint distribution of the input and output. An example of the former is a neural network or a decision tree and a restricted Boltzmann machine (RBM) is an example of the latter. Learning the joint distribution has the advantage that by the Bayes' rule, it can also give the conditional distribution of the output given the input, but also be used for other purposes such as selecting features based on the joint probability. Generative models are typically more difficult to build.

This book will mostly focus on discriminative models only, but the distinction should be made clear. A discriminative model predicts the probability of an output given an input. For example, if we are predicting the probability of a label $G = k, k \in \mathcal{K}$, then $g(\boldsymbol{x}|\boldsymbol{\theta})$ is a map $g : \mathbb{R}^p \to [0,1]^K$ and the outputs represent a discrete probability distribution over G referred to as a "one-hot" encoding—a K-vector of zeros with 1 at the kth position:

$$\hat{G}_k := \mathbb{P}(G = k \mid X = \boldsymbol{x}, \theta) = g_k(\boldsymbol{x}|\boldsymbol{\theta}) \tag{1.1}$$

[1]The model is referred to as *non-parametric* if the parameter space is infinite dimensional and *parametric* if the parameter space is finite dimensional.

and hence we have that

$$\sum_{k \in \mathcal{K}} g_k(\boldsymbol{x}|\boldsymbol{\theta}) = 1. \tag{1.2}$$

In particular, when G is dichotomous ($K = 2$), the second component of the model output is the conditional expected value of G

$$\hat{G} := \hat{G}_1 = g_1(\boldsymbol{x}|\boldsymbol{\theta}) = 0 \cdot \mathbb{P}(G = 0 \mid X = \boldsymbol{x}, \theta) + 1 \cdot \mathbb{P}(G = 1 \mid X = \boldsymbol{x}, \theta) = \mathbb{E}[G \mid X = \boldsymbol{x}, \theta]. \tag{1.3}$$

The conditional variance of G is given by

$$\sigma^2 := \mathbb{E}[(G - \hat{G})^2 \mid X = \boldsymbol{x}, \theta] = g_1(\boldsymbol{x}|\boldsymbol{\theta}) - (g_1(\boldsymbol{x}|\boldsymbol{\theta}))^2, \tag{1.4}$$

which is an inverted parabola with a maximum at $g_1(\boldsymbol{x}|\boldsymbol{\theta}) = 0.5$. The following example illustrates a simple discriminative model which, here, is just based on a set of fixed rules for partitioning the input space.

Example 1.1 Model Selection

Suppose $G \in \{A, B, C\}$ and the input $X \in \{0, 1\}^2$ are binary 2-vectors given in Table 1.1.

Table 1.1 Sample model data

G	$\boldsymbol{x}$
A	(0, 1)
B	(1, 1)
C	(1, 0)
C	(0, 0)

To match the input and output in this case, one could define a parameter-free step function $g(\boldsymbol{x})$ over $\{0, 1\}^2$ so that

$$g(\boldsymbol{x}) = \begin{cases} \{1, 0, 0\} & \text{if } \boldsymbol{x} = (0, 1) \\ \{0, 1, 0\} & \text{if } \boldsymbol{x} = (1, 1) \\ \{0, 0, 1\} & \text{if } \boldsymbol{x} = (1, 0) \\ \{0, 0, 1\} & \text{if } \boldsymbol{x} = (0, 0). \end{cases} \tag{1.5}$$

The discriminative model $g(\boldsymbol{x})$, defined in Eq. 1.5, specifies a set of fixed rules which predict the outcome of this experiment with 100% accuracy. Intuitively, it seems clear that such a model is flawed if the actual relation between inputs and outputs is non-deterministic. Clearly, a skilled analyst would typically not build such

a model. Yet, hard-wired rules such as this are ubiquitous in the finance industry such as rule-based technical analysis and heuristics used for scoring such as credit ratings.

If the model is allowed to be general, there is no reason why this particular function should be excluded. Therefore automated systems analyzing datasets such as this may be prone to construct functions like those given in Eq. 1.5 unless measures are taken to prevent it. It is therefore incumbent on the model designer to understand what makes the rules in Eq. 1.5 objectionable, with the goal of using a theoretically sound process to generalize the input–output map to other data.

Example 1.2 Model Selection (Continued)

Consider an alternate model for Table 1.1

$$h(\boldsymbol{x}) = \begin{cases} \{0.9, 0.05, 0.05\} & \text{if } \boldsymbol{x} = (0, 1) \\ \{0.05, 0.9, 0.05\} & \text{if } \boldsymbol{x} = (1, 1) \\ \{0.05, 0.05, 0.9\} & \text{if } \boldsymbol{x} = (1, 0) \\ \{0.05, 0.05, 0.9\} & \text{if } \boldsymbol{x} = (0, 0). \end{cases}$$

If this model were sampled, it would produce the data in Table 1.1 with probability $(0.9)^4 = 0.6561$. We can hardly exclude this model from consideration on the basis of the results in Table 1.1, so which one do we choose?

Informally, the heart of the model selection problem is that model g has excessively high confidence about the data, when that confidence is often not warranted. Many other functions, such as h, could have easily generated Table 1.1. Though there is only one model that can produce Table 1.1 with probability 1.0, there is a whole family of models that can produce the table with probability at least 0.66. Many of these plausible models do not assign overwhelming confidence to the results. To determine which model is best on average, we need to introduce another key concept.

2.1 Entropy

Model selection in machine learning is based on a quantity known as **entropy**. Entropy represents the amount of information associated with each event. To illustrate the concept of entropy, let us consider a non-fair coin toss. There are two outcomes, $\Omega = \{H, T\}$. Let Y be a Bernoulli random variable representing the coin flip with density $f(Y = 1) = \mathbb{P}(H) = p$ and $f(Y = 0) = \mathbb{P}(T) = 1 - p$. The (binary) entropy of Y under f is

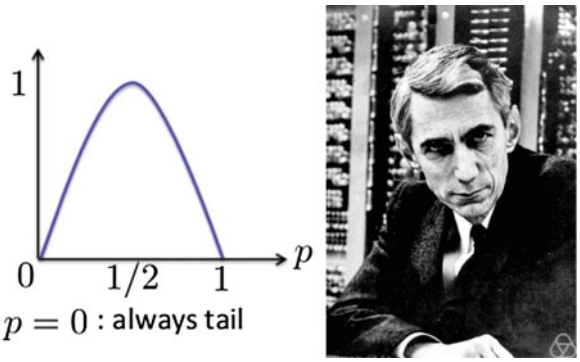


Fig. 1.2 (Left) This figure shows the binary entropy of a biased coin. If the coin is fully biased, then each flip provides no new information as the outcome is already known and hence the entropy is zero. (Right) The concept of entropy was introduced by Claude Shannon[2] in 1948[3] and was originally intended to represent an upper limit on the average length of a lossless compression encoding. Shannon's entropy is foundational to the mathematical discipline of information theory

$$\mathcal{H}(f) = -p \log_2 p - (1-p) log_2(1-p) \leq 1\text{bit}. \tag{1.6}$$

The reason why base 2 is chosen is so that the upper bound represents the number of bits needed to represent the outcome of the random variable, i.e. $\{0, 1\}$ and hence 1 bit.

The binary entropy for a biased coin is shown in Fig. 1.2. If the coin is fully biased, then each flip provides no new information as the outcome is already known. The maximum amount of information that can be revealed by a coin flip is when the coin is unbiased.

Let us now reintroduce our parameterized mass in the setting of the biased coin. Let us consider an i.i.d. discrete random variable $Y : \Omega \to \mathcal{Y} \subset \mathbb{R}$ and let

$$g(y|\theta) = \mathbb{P}(\omega \in \Omega; Y(\omega) = y)$$

denote a parameterized probability mass function for Y.

We can measure how different $g(y|\theta)$ is from the true density $f(y)$ using the cross-entropy

$$\mathcal{H}(f, g) := -\mathbb{E}_f \left[\log_2 g\right] = \sum_{y \in \mathcal{Y}} f(y) \log_2 g(y|\theta) \geq \mathcal{H}(f), \tag{1.7}$$

[2]Photo: Jacobs, Konrad [CC BY-SA 2.0 de (https://creativecommons.org/licenses/by-sa/2.0/de/deed.en)].

[3]C. Shannon, A Mathematical Theory of Communication, The Bell System Technical Journal, Vol. 27, pp. 379-423, 623-656, July, October, 1948.

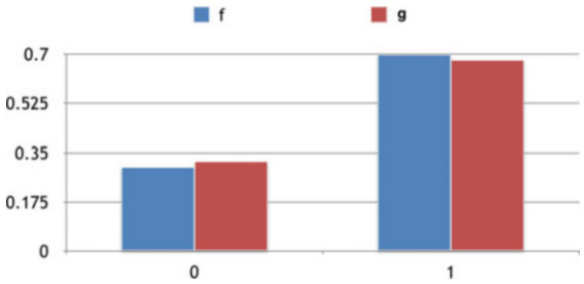


Fig. 1.3 A comparison of the true distribution, f, of a biased coin with a parameterized model g of the coin

so that $\mathcal{H}(f, f) = \mathcal{H}(f)$, where $\mathcal{H}(f)$ is the entropy of f:

$$\mathcal{H}(f) := -\mathbb{E}_f[log_2 f] = -\sum_{y \in \mathcal{Y}} f(y) log_2 f(y). \tag{1.8}$$

If $g(y|\theta)$ is a model of the non-fair coin with $g(Y = 1|\theta) = p_\theta$, $g(Y = 0|\theta) = 1 - p_\theta$. The cross-entropy is

$$\mathcal{H}(f, g) = -p \log_2 p_\theta - (1 - p) log_2(1 - p_\theta) \geq -p \log_2 p - (1 - p) log_2(1 - p). \tag{1.9}$$

Let us suppose that $p = 0.7$ and $p_\theta = 0.68$, as illustrated in Fig. 1.3, then the cross-entropy is

$$\mathcal{H}(f, g) = -0.3 \log_2(0.32) - 0.7 \log_2(0.68) = 0.8826322.$$

Returning to our experiment in Table 1.1, let us consider the cross-entropy of these models which, as you will recall, depends on inputs too. Model g completely characterizes the data in Table 1.1 and we interpret it here as the truth. Model h, however, only summarizes some salient aspects of the data, and there is a large family of tables that would be consistent with model h. In the presence of noise or strong evidence indicating that Table 1.1 was the only possible outcome, we should interpret models like h as a more plausible explanation of the actual underlying phenomenon.

Evaluating the cross-entropy between model h and model g, we get $-\log_2(0.9)$ for each observation in the table, which gives the negative log-likelihood when summed over all samples. The cross-entropy is at its minimum when $h = g$, we get $-\log_2(1.0) = 0$. If g were a parameterized model, then clearly minimizing cross-entropy or equivalently maximizing log-likelihood gives the maximum likelihood estimate of the parameter. We shall revisit the topic of parameter estimation in Chap. 2.

? Multiple Choice Question 2

Select all of the following statements that are correct:

1. Neural network classifiers are a discriminative model which output probabilistic weightings for each category, given an input feature vector.
2. If the data is independent and identically distributed (i.i.d.), then the output of a dichotomous classifier is a conditional probability of a Bernoulli random variable.
3. A θ-parameterized discriminative model for a biased coin dependent on the environment X can be written as $\{g_i(X|\theta)\}_{i=0}^{1}$.
4. A model of two biased coins, both dependent on the environment X, can be equivalently modeled with either the pair $\{g_i^{(1)}(X|\theta)\}_{i=0}^{1}$ and $\{g_i^{(2)}(X|\theta)\}_{i=0}^{1}$, or the multi-classifier $\{g_i(X|\theta)\}_{i=0}^{3}$.

2.2 Neural Networks

Neural networks represent the non-linear map $F(X)$ over a high-dimensional input space using hierarchical layers of abstractions. An example of a neural network is a feedforward network—a sequence of L layers[4] formed via composition:

> Deep Feedforward Networks

A deep feedforward network is a function of the form

$$\hat{Y}(X) := F_{W,b}(X) = \left(f^{(L)}_{W^{(L)},b^{(L)}} \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}}\right)(X),$$

where

- $f^{(l)}_{W^{(l)},b^{(l)}}(X) := \sigma^{(l)}(W^{(l)}X + b^{(l)})$ is a semi-affine function, where $\sigma^{(l)}$ is a univariate and continuous non-linear activation function such as $max(\cdot, 0)$ or $tanh(\cdot)$.
- $W = (W^{(1)}, \ldots, W^{(L)})$ and $b = (b^{(1)}, \ldots, b^{(L)})$ are weight matrices and offsets (a.k.a. biases), respectively.

[4] Note that we do not treat the input as a layer. So there are $L - 1$ hidden layers and an output layer.

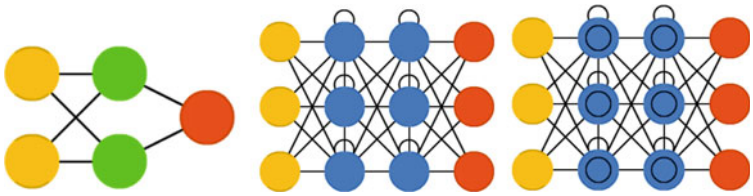


Fig. 1.4 Examples of neural networks architectures discussed in this book. Source: Van Veen, F. & Leijnen, S. (2019), "The Neural Network Zoo," Retrieved from https://www.asimovinstitute.org/neural-network-zoo. The input nodes are shown in yellow and represent the input variables, the green nodes are the hidden neurons and present hidden latent variables, the red nodes are the outputs or responses. Blue nodes denote hidden nodes with recurrence or memory. (**a**) Feedforward. (**b**) Recurrent. (**c**) Long short-term memory

An earlier example of a feedforward network architecture is given in Fig. 1.4a. The input nodes are shown in yellow and represent the input variables, the green nodes are the hidden neurons and present hidden latent variables, the red nodes are the outputs or responses. The activation functions are essential for the network to approximate non-linear functions. For example, if there is one hidden layer and $\sigma^{(1)}$ is the identify function, then

$$\hat{Y}(X) = W^{(2)}(W^{(1)}X + b^{(1)}) + b^{(2)} = W^{(2)}W^{(1)}X + W^{(2)}b^{(1)} + b^{(2)} = W'X + b' \tag{1.10}$$

is just linear regression, i.e. an affine transformation.[5] Clearly, if there are no hidden layers, the architecture recovers standard linear regression

$$Y = WX + b$$

and logistic regression $\phi(WX + b)$, where ϕ is a sigmoid or softmax function, when the response is continuous or categorical, respectively. Some of the terminology used here and the details of this model will be described in Chap. 4.

The theoretical roots of feedforward neural networks are given by the Kolmogorov–Arnold representation theorem (Arnold 1957; Kolmogorov 1957) of multivariate functions. Remarkably, Hornik et al. (1989) showed how neural networks, with one hidden layer, are universal approximators to non-linear functions.

Clearly there are a number of issues in any architecture design and inference of the model parameters (W, b). How many layers? How many neurons N_l in each hidden layer? How to perform "variable selection"? How to avoid over-fitting? The details and considerations given to these important questions will be addressed in Chap. 4.

[5] While the functional form of the map is the same as linear regression, neural networks do not assume a data generation process and hence inference is not identical to ordinary least squares regression.

3 Statistical Modeling vs. Machine Learning

Supervised machine learning is often an algorithmic form of statistical model estimation in which the data generation process is treated as an unknown (Breiman 2001). Model selection and inference is automated, with an emphasis on processing large amounts of data to develop robust models. It can be viewed as a highly efficient data compression technique designed to provide predictors in complex settings where relations between input and output variables are non-linear and input space is often high-dimensional. Machine learners balance filtering data with the goal of making accurate and robust decisions, often discrete and as a categorical function of input data.

This fundamentally differs from maximum likelihood estimators used in standard statistical models, which assume that the data was generated by the model and typically have difficulty with over-fitting, especially when applied to high-dimensional datasets. Given the complexity of modern datasets, whether they are limit order books or high-dimensional financial time series, it is increasingly questionable whether we can posit inference on the basis of a known data generation process. It is a reasonable assertion, even if an economic interpretation of the data generation process can be given, that the exact form cannot be known all the time.

The paradigm that machine learning provides for data analysis therefore is very different from the traditional statistical modeling and testing framework. Traditional fit metrics, such as R^2, t-values, p-values, and the notion of *statistical significance*, are replaced by out-of-sample forecasting and understanding the bias–variance tradeoff. Machine learning is data-driven and focuses on finding structure in large datasets. The main tools for variable or predictor selection are *regularization* and *dropout* which are discussed in detail in Chap. 4.

Table 1.2 contrasts maximum likelihood estimation-based inference with supervised machine learning. The comparison is somewhat exaggerated for ease of explanation. Rather the two approaches should be viewed as opposite ends of a continuum of methods. Linear regression techniques such as LASSO and ridge regression, or hybrids such as Elastic Net, fall somewhere in the middle, providing some combination of the explanatory power of maximum likelihood estimation while retaining out-of-sample predictive performance on high-dimensional datasets.

3.1 Modeling Paradigms

Machine learning and statistical methods can be further characterized by whether they are parametric or non-parametric. *Parametric models* assume some finite set of parameters and attempt to model the response as a function of the input variables and the parameters. Due to the finiteness of the parameter space, they have limited flexibility and cannot capture complex patterns in big data. As a general rule, examples of parametric models include ordinary least squares linear

Table 1.2 This table contrasts maximum likelihood estimation-based inference with supervised machine learning. The comparison is somewhat exaggerated for ease of explanation; however, the two should be viewed as opposite ends of a continuum of methods. Regularized linear regression techniques such as LASSO and ridge regression, or hybrids such as Elastic Net, provide some combination of the explanatory power of maximum likelihood estimation while retaining out-of-sample predictive performance on high-dimensional datasets

Property	Statistical inference	Supervised machine learning
Goal	Causal models with explanatory power	Prediction performance, often with limited explanatory power
Data	The data is generated by a model	The data generation process is unknown
Framework	Probabilistic	Algorithmic and Probabilistic
Expressibility	Typically linear	Non-linear
Model selection	Based on information criteria	Numerical optimization
Scalability	Limited to lower-dimensional data	Scales to high-dimensional input data
Robustness	Prone to over-fitting	Designed for out-of-sample performance
Diagnostics	Extensive	Limited

regression, polynomial regression, mixture models, neural networks, and hidden Markov models.

Non-parametric models treat the parameter space as infinite dimensional—this is equivalent to introducing a hidden or latent function. The model structure is, for the most part, not specified a priori and they can grow in complexity with more data. Examples of non-parametric models include kernel methods such as support vector machines and Gaussian processes, the latter will be the focus of Chap. 3.

Note that there is a gray area in whether neural networks are parametric or non-parametric and it strictly depends on how they are fitted. For example, it is possible to treat the parameter space in a neural network as infinite dimensional and hence characterize neural networks as non-parametric (see, for example, Philipp and Carbonell (2017)). However, this is an exception rather than the norm.

While on the topic of modeling paradigms, it is helpful to further distinguish between *probabilistic models*, the subject of the next two chapters, and deterministic models, the subject of Chaps. 4, 5, and 8. The former treats the parameters as random and the latter assumes that the parameters are given.

Within probabilistic modeling, a particular niche is occupied by the so-called *state-space models*. In these models one assumes the existence of a certain unobserved, latent, process, whose evolution drives a certain observable process. The evolution of the latent process and the dependence of the observable process on the latent process may be given in stochastic, probabilistic terms, which places the state-space models within the realm of probabilistic modeling.

Note, somewhat counter to the terminology, that a deterministic model may produce a probabilistic output, for example, a logistic regression gives the probability that the response is positive given the input variables. The choice of whether to use a probabilistic or deterministic model is discussed further in the next chapter

and falls under the more general and divisive topic of "Bayesian versus frequentist modeling."

3.2 *Financial Econometrics and Machine Learning*

Machine learning generalizes parametric methods in financial econometrics. A taxonomy of machine learning in econometrics in shown in Fig. 1.5 together with the section references to the material in the first two parts of this book.

When the data is a time series, neural networks can be configured with recurrence to build memory into the model. By relaxing the modeling assumptions needed for econometrics techniques, such as ARIMA (Box et al. 1994) and GARCH models (Bollerslev 1986), recurrent neural networks provide a semi-parametric or even non-parametric extension of classical time series methods. That use, however, comes with much caution. Whereas financial econometrics is built on rigorous experimental design methods such as the estimation framework of Box and Jenkins (1976), recurrent neural networks have grown from the computational engineering literature and many engineering studies overlook essential diagnostics such as Dickey–Fuller tests for verifying stationarity of the time series, a critical aspect of financial time series modeling. We take an integrative approach, showing how to cast recurrent neural networks into financial econometrics frameworks such as Box-Jenkins.

More formally, if the input–output pairs $\mathcal{D} = \{X_t, Y_t\}_{t=1}^{N}$ are autocorrelated observations of X and Y at times $t = 1, \ldots, N$, then the fundamental prediction problem can be expressed as a sequence prediction problem: construct a non-linear

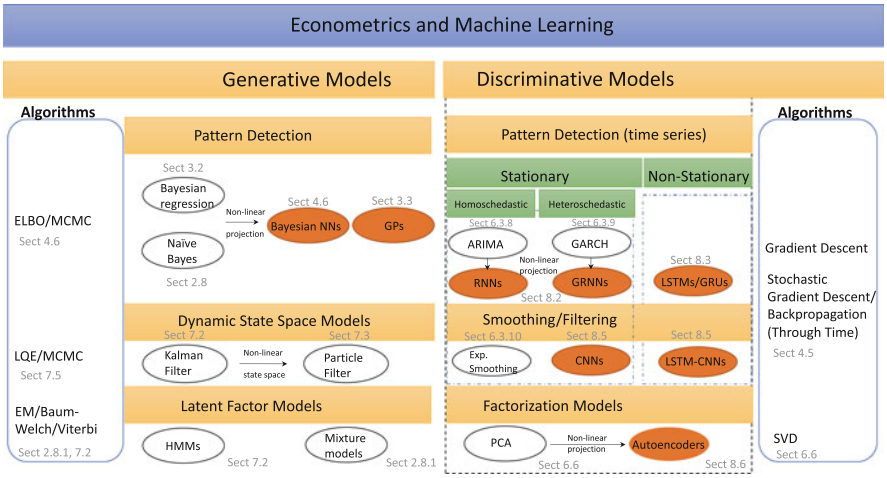


Fig. 1.5 Overview of how machine learning generalizes parametric econometrics, together with the section references to the material in the first two parts of this book

times series predictor, $\hat{Y}(\mathcal{X}_t)$, of an output, Y, using a high-dimensional input matrix of T length sub-sequences $\mathcal{X}_t$:

$$\hat{Y}_t = F(\mathcal{X}_t) \text{ where } \mathcal{X}_t := seq_{T,0}(X_t) := (X_{t-T+1}, \ldots, X_t), \tag{1.11}$$

where X_{t-j} is a jth lagged observation of X_t, $X_{t-j} = L^j[X_j]$, for $0 = 1, \ldots, T-1$. Sequence learning, then, is just a composition of a non-linear map and a vectorization of the lagged input variables. If the data is i.i.d., then no sequence is needed (i.e., $T = 1$), and we recover the standard cross-sectional prediction problem which can be approximated with a feedforward neural network model.

Recurrent neural networks (RNNs), shown in Fig. 1.4b, are time series methods or sequence learners which have achieved much success in applications such as natural language understanding, language generation, video processing, and many other tasks (Graves 2012). There are many types of RNNs—we will just concentrate on simple RNN models for brevity of notation. Like multivariate structural autoregressive models, RNNs apply an autoregressive function $f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_t)$ to each input sequence $\mathcal{X}_t$, where T denotes the look back period at each time step—the maximum number of lags. However, rather than directly imposing a linear autocovariance structure, a RNN provides a flexible functional form to directly model the predictor, $\hat{Y}$.

A simple RNN can be understood as an unfolding of a single hidden layer neural network (a.k.a. Elman network (Elman 1991)) over all time steps in the sequence, $j = 0, \ldots, T$. For each time step, j, this function $f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_{t,j})$ generates a hidden state Z_{t-j} from the current input X_{t-j} and the previous hidden state Z_{t-j-1} and $\mathcal{X}_{t,j} = seq_{T,j}(X_t) \subset \mathcal{X}_t$ which appears in general form as:

$$\begin{aligned}
\text{response:} \quad & \hat{Y}_t = f^{(2)}_{W^{(2)},b^{(2)}}(Z_t) := \sigma^{(2)}(W^{(2)} Z_t + b^{(2)}), \\
\text{hidden states:} \quad & Z_{t-j} = f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_{t,j}) \\
& := \sigma^{(1)}(W^{(1)}_z Z_{t-j-1} + W^{(1)}_x X_{t-j} + b^{(1)}), \ j \in \{T, \ldots, 0\},
\end{aligned}$$

where $\sigma^{(1)}$ is an activation function such as $\tanh(x)$ and $\sigma^{(2)}$ is either a softmax function, sigmoid function, or identity function depending on whether the response is categorical, binary, or continuous, respectively. The connections between the extremal inputs X_t and the H hidden units are weighted by the time invariant matrix $W^{(1)}_x \in \mathbb{R}^{H \times P}$. The recurrent connections between the H hidden units are weighted by the time invariant matrix $W^{(1)}_z \in \mathbb{R}^{H \times H}$. Without such a matrix, the architecture is simply a single layered feedforward network without memory—each independent observation X_t is mapped to an output $\hat{Y}_t$ using the same hidden layer.

It is important to note that a plain RNN, de-facto, is not a deep network. The recurrent layer has the deceptive appearance of being a deep network when "unfolded," i.e. viewed as being repeatedly applied to each new input, X_{t-j}, so that $Z_{t-j} = \sigma^{(1)}(W^{(1)}_z Z_{t-j-1} + W^{(1)}_x X_{t-j})$. However the same recurrent weights

remain fixed over all repetitions—there is only one recurrent layer with weights $W_z^{(1)}$.

The amount of memory in the model is equal to the sequence length T. This means that the maximum lagged input that affects the output, $\hat{Y}_t$, is X_{t-T}. We shall see later in Chap. 8 that RNNs are simply non-linear autoregressive models with exogenous variables (NARX). In the special case of the univariate time series prediction $\hat{X}_t = F(\mathcal{X}_{t-1})$, using $T = p$ previous observations $\{X_{t-i}\}_{i=1}^{p}$, only one neuron in the recurrent layer with weight ϕ and no activation function, a RNN is an AR(p) model with zero drift and geometric weights:

$$\hat{X}_t = (\phi_1 L + \phi_2 L^2 + \cdots + \phi_p L^p)[X_t],\ \phi_i := \phi^i,$$

with $|\phi| < 1$ to ensure that the model is stationary. The order p can be found through autocorrelation tests of the residual if we make the additional assumption that the error $X_t - \hat{X}_t$ is Gaussian. Example tests include the Ljung–Box and Lagrange multiplier tests. However, the over-reliance on parametric diagnostic tests should be used with caution since the conditions for satisfying the tests may not be satisfied on complex time series data. Because the weights are time independent, plain RNNs are static time series models and not suited to non-covariance stationary time series data.

Additional layers can be added to create deep RNNs by stacking them on top of each other, using the hidden state of the RNN as the input to the next layer. However, RNNs have difficulty in learning long-term dynamics, due in part to the vanishing and exploding gradients that can result from propagating the gradients down through the many unfolded layers of the network. Moreover, RNNs like most methods in supervised machine learning are inherently designed for stationary data. Oftentimes, financial time series data is non-stationary.

In Chap. 8, we shall introduce gated recurrent units (GRUs) and long short term memory (LSTM) networks, the latter is shown in Fig. 1.4c as a particular form of recurrent network which provide a solution to this problem by incorporating memory units. In the language of time series modeling, we shall construct dynamic RNNs which are suitable for non-stationary data. More precisely, we shall see that these architecture shall learn when to forget previous hidden states and when to update hidden states given new information.

This ability to model hidden states is of central importance in financial time series modeling and applications in trading. Mixture models and hidden Markov models have historically been the primary probabilistic methods used in quantitative finance and econometrics to model regimes and are reviewed in Chap. 2 and Chap. 7 respectively. Readers are encouraged to review Chap. 2, before reading Chap. 7.

? Multiple Choice Question 3

Select all the following correct statements:

1. A linear recurrent neural network with a memory of p lags is an autoregressive model $AR(p)$ with non-parametric error.

2. Recurrent neural networks, as time series models, are guaranteed to be stationary, for any choice of weights.
3. The amount of memory in a shallow recurrent network corresponds to the number of times a single perceptron layer is unfolded.
4. The amount of memory in a deep recurrent network corresponds to the number of perceptron layers.

3.3 Over-fitting

Undoubtedly the pivotal concern with machine learning, and especially deep learning, is the propensity for over-fitting given the number of parameters in the model. This is why skill is needed to fit deep neural networks.

In frequentist statistics, over-fitting is addressed by penalizing the likelihood function with a penalty term. A common approach is to select models based on Akaike's information criteria (Akaike 1973), which assumes that the model error is Gaussian. The penalty term is in fact a sample bias correction term to the Kullback–Leibler divergence (the relative Entropy) and is applied post-hoc to the unpenalized maximized loss likelihood.

Machine learning methods such as least absolute shrinkage and selection operator (LASSO) and ridge regression more conveniently directly optimize a loss function with a penalty term. Moreover the approach is not restricted to modeling error distributional assumptions. LASSO or L_1 regularization favors sparser parameterizations, whereas ridge regression or L_2 reduces the magnitude of the parameters. Regularization is arguably the most important aspect of why machine learning methods have been so successful in finance and other distributions. Conversely, its absence is why neural networks fell out-of-favor in the finance industry in the 1990s.

Regularization and information criteria are closely related, a key observation which enables us to express model selection in terms of information entropy and hence root our discourse in the works of Shannon (1948), Wiener (1964), Kullback and Leibler (1951). How to choose weights, the concept of regularization for model selection, and cross-validation is discussed in Chap. 4.

It turns out that the choice of priors in Bayesian modeling provides a probabilistic analog to LASSO and ridge regression. L_2 regularization is equivalent to a Gaussian prior and L_1 is an equivalent to a Laplacian prior. Another important feature of Bayesian models is that they have a natural mechanism for prevention of over-fitting built-in. Introductory Bayesian modeling is covered extensively in Chap. 2.

4 Reinforcement Learning

Recall that supervised learning is essentially a paradigm for inferring the parameters of a map between input data and an output through minimizing an error over training samples. Performance generalization is achieved through estimating regularization parameters on cross-validation data. Once the weights of a network are learned, they are not updated in response to new data. For this reason, supervised learning can be considered as an "offline" form of learning, i.e. the model is fitted offline. Note that we avoid referring to the model as static since it is possible, under certain types of architectures, to create a dynamical model in which the map between input and output changes over time. For example, as we shall see in Chap. 8, a LSTM maintains a set of hidden state variables which result in a different form of the map over time.

In such learning, a "teacher" provides an exact right output for each data point in a training set. This can be viewed as "feedback" from the teacher, which for supervised learning amounts to informing the agent with the correct label each time the agent classifies a new data point in the training dataset. Note that this is opposite to unsupervised learning, where there is no teacher to provide correct answers to a ML algorithm, which can be viewed as a setting with no teacher, and, respectively, no feedback from a teacher.

An alternative learning paradigm, referred to as "reinforcement learning," exists which models a sequence of decisions over state space. The key difference of this setting from supervised learning is feedback from the teacher is somewhat in between of the two extremes of unsupervised learning (no feedback at all) and supervised learning that can be viewed as feedback by providing the right labels. Instead, such partial feedback is provided by "rewards" which encourage a desired behavior, but without explicitly instructing the agent what exactly it should do, as in supervised learning.

The simplest way to reason about reinforcement learning is to consider machine learning tasks as a problem of an agent interacting with an environment, as illustrated in Fig. 1.6.

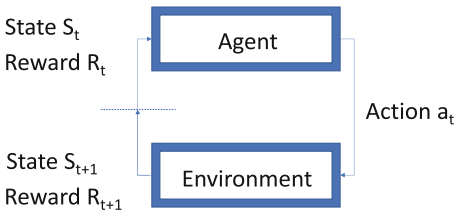


Fig. 1.6 This figure shows a reinforcement learning agent which performs actions at times $t_0, \ldots, t_n$. The agent perceives the environment through the state variable S_t. In order to perform better on its task, feedback on an action a_t is provided to the agent at the next time step in the form of a reward R_t

The agent learns about the environment in order to perform better on its task, which can be formulated as the problem of performing an optimal **action**. If an action performed by an agent is always the same and does not impact the environment, in this case we simply have a perception task, because learning about the environment helps to improve performance on this task. For example, you might have a model for prediction of mortgage defaults where the action is to compute the default probability for a given mortgage. The agent, in this case, is just a predictive model that produces a number and there is measurement of how the model impacts the environment. For example, if a model at a large mortgage broker predicted that all borrowers will default, it is very likely that this would have an impact on the mortgage market, and consequently future predictions. However, this feedback is ignored as the agent just performs perception tasks, ideally suited for supervised learning. Another example is in trading. Once an action is taken by the strategy there is feedback from the market which is referred to as "market impact."

Such a learner is configured to maximize a long-run utility function under some assumptions about the environment. One simple assumption is to treat the environment as being fully observable and evolving as a first-order Markov process. A Markov Decision Process (MDP) is then the simplest modeling framework that allows us to formalize the problem of reinforcement learning. A task solved by MDPs is the problem of **optimal control**, which is the problem of choosing action variables over some period of time, in order to maximize some objective function that depends both on the future states and action taken. In a discrete-time setting, the state of the environment $S_t \in \mathcal{S}$ is used by the learner (a.k.a. agent) to decide which action $a_t \in \mathcal{A}(S_t)$ to take at each time step. This decision is made dynamic by updating the probabilities of selecting each action conditioned on S_t. These conditional probabilities $\pi_t(a|s)$ are referred to as the agent's **policy**. The mechanism for updating the policy as a result of its learning is as follows: one time step later and as a consequence of its action, the learner receives a reward defined by a **reward function**, an immediate reward given the current state S_t and action taken a_t.

As a result of the dynamic environment and the action of the agent, we transition to a new state S_{t+1}. A reinforcement learning method specifies how to change the policy so as to maximize the total amount of reward received over the long-run. The constructs for reinforcement learning will be formalized in Chap. 9 but we shall informally discuss some of the challenges of reinforcement learning in finance here.

Most of the impressive progress reported recently with reinforcement learning by researchers and companies such as Google's DeepMind or OpenAI, such as playing video games, walking robots, self-driving cars, etc., assumes complete observability, using Markovian dynamics. The much more challenging problem, which is a better setting for finance, is how to formulate reinforcement learning for partially observable environments, where one or more variables are hidden.

Another, more modest, challenge exists in how to choose the optimal policy when no environment is fully observable but the dynamic process for how the states evolve over time is unknown. It may be possible, for simple problems, to reason about how the states evolve, perhaps adding constraints on the state-action space. However,

the problem is especially acute in high-dimensional discrete state spaces, arising from, say, discretizing continuous state spaces. Here, it is typically intractable to enumerate all combinations of states and actions and it is hence not possible to exactly solve the optimal control problem. Chapter 9 will present approaches for approximating the optimal control problem. In particular, we will turn to neural networks to approximate an action function known as a "Q-function." Such an approach is referred to as "Q-Learning" and more recently, with the use of deep learning to approximate the Q-function, is referred to as "Deep Q-Learning."

To fix ideas, we consider a number of examples to illustrate different aspects of the problem formulation and challenge in applying reinforcement learning. We start with arguably the most famous toy problem used to study stochastic optimal control theory, the "multi-armed bandit problem." This problem is especially helpful in developing our intuition of how an agent must balance the competing goals of exploring different actions versus exploitation of known outcomes.

Example 1.3 Multi-armed Bandit Problem

Suppose there is a fixed and finite set of n actions, a.k.a. arms, denoted $\mathcal{A}$. Learning proceeds in rounds, indexed by $t = 1, \ldots, T$. The number of rounds T, a.k.a. the time horizon, is fixed and known in advance. In each round, the agent picks an arm a_t and observes the reward $R_t(a_t)$ for the chosen arm only. For avoidance of doubt, the agent does not observe rewards for other actions that could have been selected. If the goal is to maximize total reward over all rounds, how should the agent choose an arm?

Suppose the rewards R_t are independent and identical random variables with distribution $\boldsymbol{\nu} \in [0, 1]^n$ and mean $\boldsymbol{\mu}$. The best action is then the distribution with the maximum mean μ^*.

The difference between the player's accumulated reward and the maximum the player (a.k.a. the "cumulative regret") could have obtained had she known all the parameters is

$$\bar{R}_T = T\mu^* - \mathbb{E}\sum_{t\in[T]} R_t.$$

Intuitively, an agent should pick arms that performed well in the past, yet the agent needs to ensure that no good option has been missed.

The theoretical origins of reinforcement learning are in stochastic dynamic programming. In this setting, an agent must make a sequence of decisions under uncertainty about the reward. If we can characterize this uncertainty with probability distributions, then the problem is typically much easier to solve. We shall assume that the reader has some familiarity with dynamic programming—the extension to stochastic dynamic programming is a relatively minor conceptual development. Note that Chap. 9 will review pertinent aspects of dynamic programming, including

Bellman optimality. The following optimal payoff example will likely just serve as a simple review exercise in dynamic programming, albeit with uncertainty introduced into the problem. As we follow the mechanics of solving the problem, the example exposes the inherent difficulty of relaxing our assumptions about the distribution of the uncertainty.

Example 1.4 Uncertain Payoffs

A strategy seeks to allocate \$600 across 3 markets and is equally profitable once the position is held, returning 1% of the size of the position over a short trading horizon $[t, t+1]$. However, the markets vary in liquidity and there is a lower probability that the larger orders will be filled over the horizon. The amount allocated to each market must be either $K = \{100, 200, 300\}$.

Strategy	Allocation	Fill probability
M_1	100	0.8
	200	0.7
	300	0.6
M_2	100	0.75
	200	0.7
	300	0.65
M_3	100	0.75
	200	0.75
	300	0.6

Strategy	Allocation	Return
M_1	100	0.8
	200	1.4
	300	1.8
M_2	100	0.75
	200	1.4
	300	1.95
M_3	100	0.75
	200	1.5
	300	1.8

The optimal allocation problem under uncertainty is a stochastic dynamic programming problem. We can define value functions $v_i(x)$ for total allocation amount x for each stage of the problem, corresponding to the markets. We then find the optimal allocation using the backward recursive formulae:

$$
\begin{aligned}
v_3(x) &= R_3(x), \forall x \in K, \\
v_2(x) &= \max_{k \in K} \{R_2(k) + v_3(x-k)\}, \forall x \in K + 200, \\
v_1(x) &= \max_{k \in K} \{R_1(k) + v_2(x-k)\}, x = 600,
\end{aligned}
$$

The left-hand side of the table below tabulates the values of $R_2 + v_3$ corresponding to the second stage of the backward induction for each pair (M_2, M_3).

(continued)

Example 1.4 (continued)

$R_2 + v_3$	M_2		
M_3	100	200	300
100	1.5	2.15	2.7
200	2.25	2.9	3.45
300	2.55	3.2	3.75

M_1	(M_2^*, M_3^*)	v_2	R_1	$R_1 + v_2$
100	(300, 200)	3.45	0.8	4.25
200	(200, 200)	2.9	1.4	4.3
300	(100, 200)	2.25	1.8	4.05

The right-hand side of the above table tabulates the values of $R_1 + v_2$ corresponding to the third and final stage of the backward induction for each tuple (M_1, M_2^*, M_3^*).

In the above example, we can see that the allocation {200, 200, 200} maximizes the reward to give $v_1(600) = 4.3$. While this exercise is a straightforward application of a Bellman optimality recurrence relation, it provides a glimpse of the types of stochastic dynamic programming problems that can be solved with reinforcement learning. In particular, if the fill probabilities are unknown but must be learned over time by observing the outcome over each period $[t_i, t_{i+1})$, then the problem above cannot be solved by just using backward recursion. Instead we will move to the framework of reinforcement learning and attempt to learn the best actions given the data. Clearly, in practice, the example is much too simple to be representative of real-world problems in finance—the profits will be unknown and the state space is significantly larger, compounding the need for reinforcement learning. However, it is often very useful to benchmark reinforcement learning on simple stochastic dynamic programming problems with closed-form solutions.

In the previous example, we assumed that the problem was static—the variables in the problem did not change over time. This is the so-called static allocation problem and is somewhat idealized. Our next example will provide a glimpse of the types of problems that typically arise in optimal portfolio investment where random variables are dynamic. The example is also seated in more classical finance theory, that of a "Markowitz portfolio" in which the investor seeks to maximize a risk-adjusted long-term return and the wealth process is self-financing.[6]

Example 1.5 Optimal Investment in an Index Portfolio

Let S_t be a time-t price of a risky asset such as a sector exchange-traded fund (ETF). We assume that our setting is discrete time, and we denote different time steps by integer valued-indices $t = 0, \ldots, T$, so there are $T + 1$ values on our discrete-time grid. The discrete-time random evolution of the risky asset S_t is

(continued)

[6] A wealth process is self-financing if, at each time step, any purchase of an additional quantity of the risky asset is funded from the bank account. Vice versa, any proceeds from a sale of some quantity of the asset go to the bank account.

Example 1.5 (continued)

$$S_{t+1} = S_t\,(1+\phi_t)\,, \tag{1.12}$$

where ϕ_t is a random variable whose probability distribution may depend on the current asset value S_t. To ensure non-negativity of prices, we assume that ϕ_t is bounded from below $\phi_t \geq -1$.

Consider a wealth process W_t of an investor who starts with an initial wealth $W_0 = 1$ at time $t = 0$ and, at each period $t = 0, \ldots, T-1$ allocates a fraction $u_t = u_t(S_t)$ of the total portfolio value to the risky asset, and the remaining fraction $1 - u_t$ is invested in a risk-free bank account that pays a risk-free interest rate $r_f = 0$. We will refer to a set of decision variables for all time steps as a *policy* $\pi := \{u_t\}_{t=0}^{T-1}$. The wealth process is self-financing and so the wealth at time $t+1$ is given by

$$W_{t+1} = (1-u_t)\,W_t + u_t W_t\,(1+\phi_t)\,. \tag{1.13}$$

This produces the one-step return

$$r_t = \frac{W_{t+1} - W_t}{W_t} = u_t \phi_t. \tag{1.14}$$

Note this is a random function of the asset price S_t. We define one-step rewards R_t for $t = 0, \ldots, T-1$ as risk-adjusted returns

$$R_t = r_t - \lambda \mathrm{Var}\left[r_t | S_t\right] = u_t\phi_t - \lambda u_t^2 \mathrm{Var}\left[\phi_t | S_t\right], \tag{1.15}$$

where λ is a risk-aversion parameter.[a] We now consider the problem of maximization of the following concave function of the control variable u_t:

$$V^{\pi}(s) = \max_{u_t} \mathbb{E}\left[\sum_{t=0}^{T} R_t \,\middle|\, S_t = s\right] = \max_{u_t} \mathbb{E}\left[\sum_{t=0}^{T} u_t\phi_t - \lambda u_t^2 \mathrm{Var}\left[\phi_t | S_t\right] \,\middle|\, S_t = s\right]. \tag{1.16}$$

Equation 1.16 defines an optimal investment problem for $T-1$ steps faced by an investor whose objective is to optimize risk-adjusted returns over each period. This optimization problem is equivalent to maximizing the long-run

(continued)

[a] Note, for avoidance of doubt, that the risk-aversion parameter must be scaled by a factor of $\frac{1}{2}$ to ensure consistency with the finance literature.

Example 1.5 (continued)

returns over the period $[0, T]$. For each $t = T-1, T-2, \ldots, 0$, the optimality condition for action u_t is now obtained by maximization of $V^{\pi}(s)$ with respect to u_t:

$$u_t^* = \frac{\mathbb{E}[\phi_t | S_t]}{2\lambda \mathrm{Var}[\phi_t | S_t]}, \tag{1.17}$$

where we allow for short selling in the ETF (i.e., $u_t < 0$) and borrowing of cash $u_t > 1$.

This is an example of a stochastic optimal control problem for a portfolio that maximizes its cumulative risk-adjusted return by repeatedly rebalancing between cash and a risky asset. Such problems can be solved using means of dynamic programming or reinforcement learning. In our problem, the dynamic programming solution is given by an analytical expression (1.17). Chapter 9 will present more complex settings including reinforcement learning approaches to optimal control problems, as well as demonstrate how expressions like the optimal action of Eq. 1.17 can be computed in practice.

? Multiple Choice Question 4

Select all the following correct statements:

1. The name "Markov processes" first historically appeared as a result of a misspelled name "Mark-Off processes" that was previously used for random processes that describe learning in certain types of video games, but has become a standard terminology since then.
2. The goal of (risk-neutral) reinforcement learning is to maximize the expected total reward by choosing an optimal policy.
3. The goal of (risk-neutral) reinforcement learning is to neutralize risk, i.e. make the variance of the total reward equal zero.
4. The goal of risk-sensitive reinforcement learning is to teach a RL agent to pick action policies that are most prone to risk of failure. Risk-sensitive RL is used, e.g. by venture capitalists and other sponsors of RL research, as a tool to assess the feasibility of new RL projects.

5 Examples of Supervised Machine Learning in Practice

The practice of machine learning in finance has grown somewhat commensurately with both theoretical and computational developments in machine learning. Early adopters have been the quantitative hedge funds, including Bridgewater Associates,

Renaissance Technologies, WorldQuant, D.E. Shaw, and Two Sigma who have embraced novel machine learning techniques although there are mixed degrees of adoption and a healthy skepticism exists that machine learning is a panacea for quantitative trading. In 2015, Bridgewater Associates announced a new artificial intelligence unit, having hired people from IBM Watson with expertise in deep learning. Anthony Ledford, chief scientist at MAN AHL: "It's at an early stage. We have set aside a pot of money for test trading. With deep learning, if all goes well, it will go into test trading, as other machine learning approaches have." Winton Capital Management's CEO David Harding: "People started saying, 'There's an amazing new computing technique that's going to blow away everything that's gone before.' There was also a fashion for genetic algorithms. Well, I can tell you none of those companies exist today—not a sausage of them."

Some qualifications are needed to more accurately assess the extent of adoption. For instance, there is a false line of reasoning that ordinary least squares regression and logistic regression, as well as Bayesian methods, are machine learning techniques. Only if the modeling approach is algorithmic, without positing a data generation process, can the approach be correctly categorized as machine learning. So regularized regression without use of parametric assumptions on the error distribution is an example of machine learning. Unregularized regression with, say, Gaussian error is not a machine learning technique. The functional form of the input–output map is the same in both cases, which is why we emphasize that the functional form of the map is not a sufficient condition for distinguishing ML from statistical methods.

With that caveat, we shall view some examples that not only illustrate some of the important practical applications of machine learning prediction in algorithmic trading, high-frequency market making, and mortgage modeling but also provide a brief introduction to applications that will be covered in more detail in later chapters.

5.1 Algorithmic Trading

Algorithmic trading is a natural playground for machine learning. The idea behind algorithmic trading is that trading decisions should be based on data, not intuition. Therefore, it should be viable to automate this decision-making process using an algorithm, either specified or learned. The advantages of algorithmic trading include complex market pattern recognition, reduced human produced error, ability to test on historic data, etc. In recent times, as more and more information is being digitized, the feasibility and capacity of algorithmic trading has been expanding drastically. The number of hedge funds, for example, that apply machine learning for algorithmic trading is steadily increasing.

Here we provide a simple example of how machine learning techniques can be used to improve traditional algorithmic trading methods, but also provide new trading strategy suggestions. The example here is not intended to be the "best" approach, but rather indicative of more out-of-the-box strategies that machine learning facilitates, with the emphasis on minimizing out-of-sample error by pattern matching through efficient compression across high-dimensional datasets.

Momentum strategies are one of the most well-known algo-trading strategies; In general, strategies that predict prices from historic price data are categorized as momentum strategies. Traditionally momentum strategies are based on certain regression-based econometric models, such as ARIMA or VAR (see Chap. 6). A drawback of these models is that they impose strong linearity which is not consistently plausible for time series of prices. Another caveat is that these models are parametric and thus have strong bias which often causes underfitting. Many machine learning algorithms are both non-linear and semi/non-parametric, and therefore prove complementary to existing econometric models.

In this example we build a simple momentum portfolio strategy with a feedforward neural network. We focus on the S&P 500 stock universe, and assume we have daily close prices for all stocks over a ten-year period.[7]

Problem Formulation

The most complex practical aspect of machine learning is how to choose the input ("features") and output. The type of desired output will determine whether a regressor or classifier is needed, but the general rule is that it must be actionable (i.e., tradable). Suppose our goal is to invest in an equally weighted, long only, stock portfolio only if it beats the S&P 500 index benchmark (which is a reasonable objective for a portfolio manager). We can therefore label the portfolio at every observation t based on the mean directional excess return of the portfolio:

$$G_t = \begin{cases} 1 & \frac{1}{N}\sum_i r^i_{t+h,t} - \tilde{r}_{t+h,t} \geq \epsilon, \\ 0 & \frac{1}{N}\sum_i r^i_{t+h,t} - \tilde{r}_{t+h,t} < 0, \end{cases} \tag{1.18}$$

where $r^i_{t+h,t}$ is the return of stock i between times t and $t+h$, $\tilde{r}_{t+h,t}$ is the return of the S&P 500 index in the same period, and ϵ is some target next period excess portfolio return. Without loss of generality, we could invest in the universe ($N = 500$), although this is likely to have adverse practical implications such as excessive transaction costs. We could easily just have restricted the number of stocks to a subset, such as the top decile of performing stocks in the last period. Framed this way, the machine learner is thus informing us when our stock selection strategy will outperform the market. It is largely agnostic to how the stocks are selected, provided the procedure is systematic and based solely on the historic data provided to the classifier. It is further worth noting that the map between the decision to hold the customized portfolio has a non-linear relationship with the past returns of the universe.

To make the problem more concrete, let us set $h = 5$ days. The algorithmic strategy here is therefore automating the decision to invest in the customized

[7]The question of how much data is needed to train a neural network is a central one, with the immediate concern being insufficient data to avoid over-fitting. The amount of data needed is complex to assess; however, it is partly dependent on the number of edges in the network and can be assessed through bias–variance analysis, as described in Chap. 4.

Table 1.3 Training samples for a classification problem

Date	X_1	X_2	...	X_{500}	G
2007-01-03	0.051	−0.035		0.072	0
2017-01-04	−0.092	0.125		−0.032	0
2017-01-05	0.021	0.063		−0.058	1
...					
2017-12-29	0.093	−0.023		0.045	1
2017-12-30	0.020	0.019		0.022	1
2017-12-31	−0.109	0.025		−0.092	1

portfolio or the S&P 500 index every week based on the previous 5-day realized returns of all stocks. To apply machine learning to this decision, the problem translates into finding the weights in the network between past returns and the decision to invest in the equally weighted portfolio. For avoidance of doubt, we emphasize that the interpretation of the optimal weights differs substantially from Markowitz's mean–variance portfolios, which simply finds the portfolio weights to optimize expected returns for a given risk tolerance. Here, we either invest equal amounts in all stocks of the portfolio or invest the same amount in the S&P 500 index and the weights in the network signify the relevance of past stock returns in the expected excess portfolio return outperforming the market.

Data

Feature engineering is always important in building models and requires careful consideration. Since the original price data does not meet several machine learning requirements, such as stationarity and i.i.d. distributional properties, one needs to engineer input features to prevent potential "garbage-in-garbage-out" phenomena. In this example, we take a simple approach by using only the 5-day realized returns of all S&P 500 stocks.[8] Returns are scale-free and no further standardization is needed. So for each time t, the input features are

$$\mathbf{X}_t = \left[r^1_{t,t-5}, \ldots, r^{500}_{t,t-5} \right]. \tag{1.19}$$

Now we can aggregate the features and labels into a panel indexed by date. Each column is an entry in Eq. 1.19, except for the last column which is the assigned label from Eq. 1.18, based on the realized excess stock returns of the portfolio. An example of the labeled input data (X, G) is shown in Table 1.3.

The process by which we train the classifier and evaluate its performance will be described in Chap. 4, but this example illustrates how algo-trading strategies can be crafted around supervised machine learning. Our model problem could be tailored

[8] Note that the composition of the S&P 500 changes over time and so we should interpret a feature as a fixed symbol.

for specific risk-reward and performance reporting metrics such as, for example, Sharpe or information ratios meeting or exceeding a threshold.

ϵ is typically chosen to be a small value so that the labels are not too imbalanced. As the value ϵ is increased, the problem becomes an "outlier prediction problem"—a highly imbalanced classification problem which requires more advanced sampling and interpolation techniques beyond an off-the-shelf classifier.

In the next example, we shall turn to another important aspect of machine learning in algorithmic trading, namely execution. How the trades are placed is a significant aspect of algorithmic trading strategy performance, not only to minimize price impact of market taking strategies but also for market making. Here we shall look to transactional data to perfect the execution, an engineering challenge by itself just to process market feeds of tick-by-tick exchange transactions. The example considers a market making application but could be adapted for price impact and other execution considerations in algorithmic trading by moving to a reinforcement learning framework.

A common mistake is to assume that building a predictive model will result in a profitable trading strategy. Clearly, the consideration given to reliably evaluating machine learning in the context of trading strategy performance is a critical component of its assessment.

5.2 High-Frequency Trade Execution

Modern financial exchanges facilitate the electronic trading of instruments through an instantaneous double auction. At each point in time, the market demand and the supply can be represented by an electronic limit order book, a cross-section of orders to execute at various price levels away from the market price as illustrated in Table 1.4.

Electronic market makers will quote on both sides of the market in an attempt to capture the bid–ask spread. Sometimes a large market order, or a succession of smaller markets orders, will consume an entire price level. This is why the market price fluctuates in liquid markets—an effect often referred to by practitioners as a "price-flip." A market maker can take a loss if only one side of the order is filled as a result of an adverse price movement.

Figure 1.7 (left) illustrates a typical mechanism resulting in an adverse price movement. A snapshot of the limit order book at time t, before the arrival of a market order, and after at time $t+1$ is shown in the left and right panels, respectively. The resting orders placed by the market marker are denoted with the "+" symbol—red denotes a bid and blue denotes an ask quote. A buy market order subsequently arrives and matches the entire resting quantity of best ask quotes. Then at event time $t + 1$ the limit order book is updated—the market maker's ask has been filled (blue minus symbol) and the bid now rests away from the inside market. The market marker may systematically be forced to cancel the bid and buy back at a higher price, thus taking a loss.

Table 1.4 This table shows a snapshot of the limit order book of S&P 500 e-mini futures (ES). The top half ("sell-side") shows the ask volumes and prices and the lower half ("buy side") shows the bid volumes and prices. The quote levels are ranked by the most competitive at the center (the "inside market"), outward to the least competitive prices at the top and bottom of the limit order book. Note that only five bid or ask levels are shown in this example, but the actual book is much deeper

Bid	Price	Ask
	2170.25	1284
	2170.00	1642
	2169.75	1401
	2169.50	1266
	2169.25	290
477	2169.00	
1038	2168.75	
950	2168.50	
1349	2168.25	
1559	2168.00	

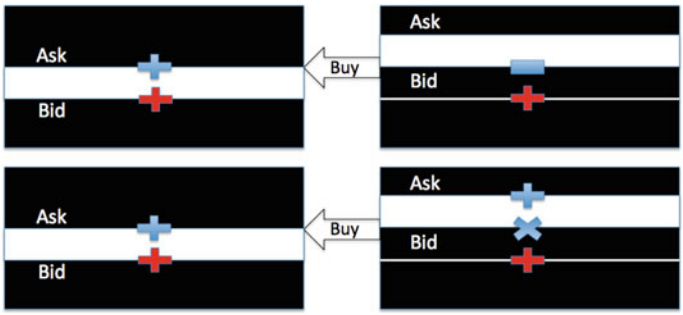


Fig. 1.7 (Top) A snapshot of the limit order book is taken at time t. Limit orders placed by the market marker are denoted with the "+" symbol—red denotes a bid and blue denotes an ask. A buy market order subsequently arrives and matches the entire resting quantity of best ask quotes. Then at event time $t + 1$ the limit order book is updated. The market maker's ask has been filled (blue minus symbol) and the bid rests away from the inside market. (Bottom) A pre-emptive strategy for avoiding adverse price selection is illustrated. The ask is requoted at a higher ask price. In this case, the bid is not replaced and the market maker may capture a tick more than the spread if both orders are filled

Machine learning can be used to predict these price movements (Kearns and Nevmyvaka 2013; Kercheval and Zhang 2015; Sirignano 2016; Dixon et al. 2018; Dixon 2018b,a) and thus to potentially avoid adverse selection. Following Cont and de Larrard (2013) we can treat queue sizes at each price level as input variables. We can additionally include properties of market orders, albeit in a form which our machines deem most relevant to predicting the direction of price movements (a.k.a. feature engineering). In contrast to stochastic modeling, we do not impose conditional distributional assumptions on the independent variables (a.k.a. features) nor assume that price movements are Markovian. Chapter 8 presents a RNN for

mid-price prediction from the limit order book history which is the starting point for the more in-depth study of Dixon (2018b) which includes market orders and demonstrates the superiority of RNNs compared to other time series methods such as Kalman filters.

We reiterate that the ability to accurately predict does not imply profitability of the strategy. Complex issues concerning queue position, exchange matching rules, latency, position constraints, and price impact are central considerations for practitioners. The design of profitable strategies goes beyond the scope of this book but the reader is referred to de Prado (2018) for the pitfalls of backtesting and designing algorithms for trading. Dixon (2018a) presents a framework for evaluating the performance of supervised machine learning algorithms which accounts for latency, position constraints, and queue position. However, supervised learning is ultimately not the best machine learning approach as it cannot capture the effect of market impact and is too inflexible to incorporate more complex strategies. Chapter 9 presents examples of reinforcement learning which demonstrate how to capture market impact and also how to flexibly formulate market making strategies.

5.3 Mortgage Modeling

Beyond the data rich environment of algorithmic trading, does machine learning have a place in finance? One perspective is that there simply is not sufficient data for some "low-frequency" application areas in finance, especially where traditional models have failed catastrophically. The purpose of this section is to serve as a sobering reminder that long-term forecasting goes far beyond merely selecting the best choice of machine learning algorithm and why there is no substitute for strong domain knowledge and an understanding of the limitations of data.

In the USA, a mortgage is a loan collateralized by real-estate. Mortgages are used to securitize financial instruments such as mortgage backed securities and collateralized mortgage obligations. The analysis of such securities is complex and has changed significantly over the last decade in response to the 2007–2008 financial crises (Stein 2012).

Unless otherwise specified, a mortgage will be taken to mean a "residential mortgage," which is a loan with payments due monthly that is collateralized by a single family home. Commercial mortgages do exist, covering office towers, rental apartment buildings, and industrial facilities, but they are different enough to be considered separate classes of financial instruments. Borrowing money to buy a house is one of the most common, and largest balance, loans that an individual borrower is ever likely to commit to. Within the USA alone, mortgages comprise a staggering $15 trillion dollars in debt. This is approximately the same balance as the total federal debt outstanding (Fig. 1.8).

Within the USA, mortgages may be repaid (typically without penalty) at will by the borrower. Usually, borrowers use this feature to refinance their loans in favorable interest rate regimes, or to liquidate the loan when selling the underlying house. This

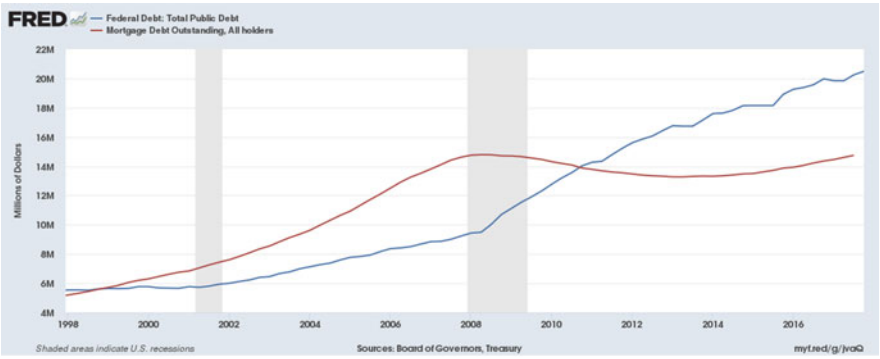


Fig. 1.8 Total mortgage debt in the USA compared to total federal debt, millions of dollars, unadjusted for inflation. Source: https://fred.stlouisfed.org/series/MDOAH, https://fred.stlouisfed.org/series/GFDEBTN

Table 1.5 At any time, the states of any US style residential mortgage is in one of the several possible states

Symbol	Name	Definition
P	Paid	All balances paid, loan is dissolved
C	Current	All payments due have been paid
3	30-days delinquent	Mortgage is delinquent by one payment
6	60-days delinquent	Delinquent by 2 payments
9	90+ delinquent	Delinquent by 3 or more payments
F	Foreclosure	Foreclosure has been initiated by the lender
R	Real-Estate-Owned (REO)	The lender has possession of the property
D	Default liquidation	Loan is involuntarily liquidated for nonpayment

has the effect of moving a great deal of financial risk off of individual borrowers, and into the financial system. It also drives a lively and well developed industry around modeling the behavior of these loans.

The mortgage model description here will generally follow the comprehensive work in Sirignano et al. (2016), with only a few minor deviations.

Any US style residential mortgage, in each month, can be in one of the several possible states listed in Table 1.5.

Consider this set of K available states to be $\mathbb{K} = \{P, C, 3, 6, 9, F, R, D\}$. Following the problem formulation in Sirignano et al. (2016), we will refer to the status of loan n at time t as $U_t^n \in \mathbb{K}$, and this will be represented as a probability vector using a standard one-hot encoding.

If $X = (X_1, \ldots, X_P)$ is the input matrix of P explanatory variables, then we define a probability transition density function $g : \mathbb{R}^P \to [0, 1]^{K \times K}$ parameterized by θ so that

$$\mathbb{P}(U_{t+1}^n = i \mid U_t^n = j, X_t^n) = g_{i,j}(X_t^n \mid \theta), \forall i, j \in \mathbb{K}. \tag{1.20}$$

Note that $g(X_t^n \mid \theta)$ is a time in-homogeneous $K \times K$ Markov transition matrix. Also, not all transitions are even conceptually possible—there are non-commutative states. For instance, a transition from C to 6 is not possible since a borrower cannot miss two payments in a single month. Here we will write $p_{(i,j)} := g_{i,j}(X_t^n \mid \theta)$ for ease of notation and because of the non-commutative state transitions where $p_{(i,j)} = 0$, the Markov matrix takes the form:

$$g(X_t^n \mid \theta) = \begin{bmatrix} 1 & p_{(c,p)} & p_{(3,p)} & 0 & 0 & 0 & 0 & 0 \\ 0 & p_{(c,c)} & p_{(3,c)} & p_{(6,c)} & p_{(9,c)} & p_{(f,c)} & 0 & 0 \\ 0 & p_{(c,3)} & p_{(3,3)} & p_{(6,3)} & p_{(9,3)} & p_{(f,3)} & 0 & 0 \\ 0 & 0 & p_{(3,6)} & p_{(6,6)} & p_{(9,6)} & p_{(f,6)} & 0 & 0 \\ 0 & 0 & 0 & p_{(6,9)} & p_{(9,9)} & p_{(f,9)} & 0 & 0 \\ 0 & 0 & 0 & p_{(6,f)} & p_{(9,f)} & p_{(f,f)} & 0 & 0 \\ 0 & 0 & 0 & p_{(6,r)} & p_{(9,r)} & p_{(f,r)} & p_{(r,r)} & 0 \\ 0 & 0 & 0 & p_{(6,d)} & p_{(9,d)} & p_{(f,d)} & p_{(r,d)} & 1 \end{bmatrix}.$$

Our classifier $g_{i,j}(X_t^n \mid \theta)$ can thus be constructed so that only the probability of transition between the commutative states are outputs and we can apply softmax functions on a subset of the outputs to ensure that $\sum_{j \in \mathbb{K}} g_{i,j}(X_t^n \mid \theta) = 1$ and hence the transition probabilities in each row sum to one.

For the purposes of financial modeling, it is important to realize that both states P and D are loan liquidation terminal states. However, state P is considered to be voluntary loan liquidation (e.g., prepayment due to refinance), whereas state D is considered to be involuntary liquidation (e.g., liquidation via foreclosure and auction). These states are not distinguishable in the mortgage data itself, but rather the driving force behind liquidation must be inferred from the events leading up to the liquidation.

One contributor to mortgage model misprediction in the run up to the 2008 financial crisis was that some (but not all) modeling groups considered loans liquidating from deep delinquency (e.g., status 9) to be the transition $9 \to P$ if no losses were incurred. However, behaviorally, these were typically defaults due to financial hardship, and they would have had losses in a more difficult house price regime. They were really $9 \to D$ transitions that just happened to be lossless due to strong house price gains over the life of the mortgage. Considering them to be voluntary prepayments (status P) resulted in systematic over-prediction of prepayments in the aftermath of major house price drops. The matrix above therefore explicitly excludes this possibility and forces delinquent loan liquidation to be always considered involuntary.

The reverse of this problem does not typically exist. In most states it is illegal to force a borrower into liquidation until at least 2 payments have been missed. Therefore, liquidation from C or 3 is always voluntary, and hence $C \to P$ and $3 \to P$. Except in cases of fraud or severe malfeasance, it is almost never economically advantageous for a lender to force liquidation from status 6, but it is not illegal. Therefore the transition $3 \to D$ is typically a data error, but $6 \to D$ is merely very rare.

Example 1.6 Parameterized Probability Transitions

If loan n is current in time period t, then

$$\mathbb{P}(U_t^n) = (0, 1, 0, 0, 0, 0, 0, 0)^T. \tag{1.21}$$

If we have $p_{(c,p)} = 0.05$, $p_{(c,c)} = 0.9$, and $p_{(c,3)} = 0.05$, then

$$\mathbb{P}(U_{t+1}^n \mid X_t^n) = g(X_t^n \mid \theta) \cdot \mathbb{P}(U_t^n) = (0.05, 0.9, 0.05, 0, 0, 0, 0, 0)^T. \tag{1.22}$$

Common mortgage models sometimes use additional states, often ones that are (without additional information) indistinguishable from the states listed above. Table 1.6 describes a few of these.

The reason for including these is the same as the reason for breaking out states like REO, status R. It is known on theoretical grounds that some model regressors from X_t^n should not be relevant for R. For instance, since the property is now owned by the lender, and the loan itself no longer exists, the interest rate (and rate incentive) of the original loan should no longer have a bearing on the outcome. To avoid over-fitting due to highly colinear variables, these known-useless variables are then excluded from transitions models starting in status R.

This is the same reason status T is sometimes broken out, especially for logistic regressions. Without an extra status listed in this way, strong rate disincentives could drive prepayments in the model to (almost) zero, but we know that people die and divorce in all rate regimes, so at least some minimal level of premature loan liquidations must still occur based on demographic factors, not financial ones.

5.3.1 Model Stability

Unlike many other models, mortgage models are designed to accurately predict events a decade or more in the future. Generally, this requires that they be built on regressors that themselves can be accurately predicted, or at least hedged. Therefore, it is common to see regressors like FICO at origination, loan age in months, rate incentive, and loan-to-value (LTV) ratio. Often LTV would be called MTMLTV if it is marked-to-market against projected or realized housing price moves. Of these regressors, original FICO is static over the life of the loan, age is deterministic,

Table 1.6 A brief description of mortgage states

Symbol	Name	Overlaps with	Definition
T	Turnover	P	Loan prepaid due to non-financial life event
U	Curtailment	C	Borrower overpaid to reduce loan principal

Table 1.7 Loan originations by year (Freddie Mac, FRM30)

Year	Loans originated
1999	976,159
2000	733,567
2001	1,542,025
2002	1,403,515
2003	2,063,488
2004	1,133,015
2005	1,618,748
2006	1,300,559
2007	1,238,814
2008	1,237,823
2009	1,879,477
2010	1,250,484
2011	1,008,731
2012	1,249,486
2013	1,375,423
2014	942,208

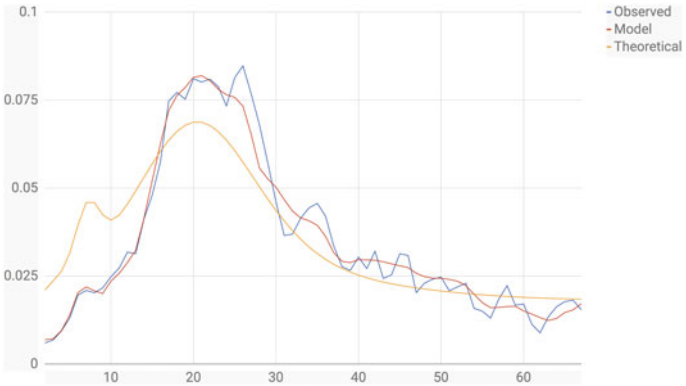


Fig. 1.9 Sample mortgage model predicting $C \rightarrow 3$ and fit on loans originated in 2001 and observed until 2006, by loan age (in months). The prepayment probabilities are shown on the y-axis

rates can be hedged, and MTMLTV is rapidly driven down by loan amortization and inflation thus eliminating the need to predict it accurately far into the future.

Consider the Freddie Mac loan level dataset of 30 year fixed rate mortgages originated through 2014. This includes each monthly observation from each loan present in the dataset. Table 1.7 shows the loan count by year for this dataset.

When a model is fit on 1 million observations from loans originated in 2001 and observed until the end of 2006, its $C \rightarrow P$ probability charted against age is shown in Fig. 1.9.

In Fig. 1.9 the curve observed is the actual prepayment probability of the observations with the given age in the test dataset, "Model" is the model prediction,

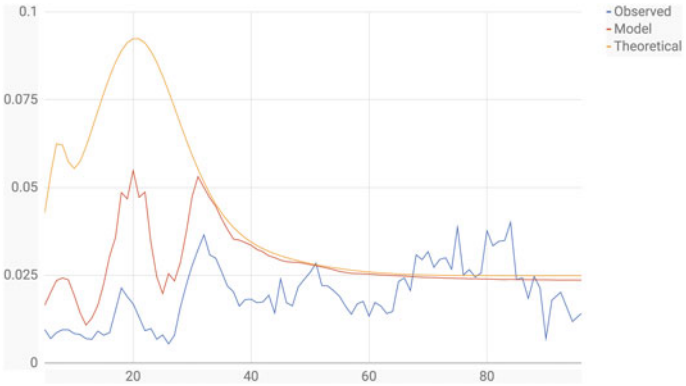


Fig. 1.10 Sample mortgage model predicting $C \rightarrow 3$ and fit on loans originated in 2006 and observed until 2015, by loan age (in months). The prepayment probabilities are shown on the y-axis

and "Theoretical" is the response to age by a theoretical loan with all other regressors from X_t^n held constant. Two observations are worth noting:

1. The marginal response to age closely matches the model predictions; and
2. The model predictions match actual behavior almost perfectly.

This is a regime where prepayment behavior is largely driven by age. When that same model is run on observations from loans originated in 2006 (the peak of housing prices before the crisis), and observed until 2015, Fig. 1.10 is produced.

Three observations are warranted from this figure:

1. The observed distribution is significantly different from Fig. 1.9;
2. The model predicted a decline of 25%, but the actual decline was approximately 56%; and
3. Prepayment probabilities are largely indifferent to age.

The regime shown here is clearly not driven by age. In order to provide even this level of accuracy, the model had to extrapolate far from any of the available data and "imagine" a regime where loan age is almost irrelevant to prepayment. This model meets with mixed success. This particular one was fit on only 8 regressors, a more complicated model might have done better, but the actual driver of this inaccuracy was a general tightening of lending standards. Moreover, there was no good data series available before the crisis to represent lending standards.

This model was reasonably accurate even though almost 15 years separated the start of the fitting data from the end of the projection period, and a lot happened in that time. Mortgage models in particular place a high premium on model stability, and the ability to provide as much accuracy as possible even though the underlying distribution may have changed dramatically from the one that generated the fitting data. Notice also that cross-validation would not help here, as we cannot draw testing data from the distribution we care about, since that distribution comes from the future.

Most importantly, this model shows that the low-dimensional projections of this (moderately) high-dimensional problem are extremely deceptive. No modeler would have chosen a shape like the model prediction from Fig. 1.9 as function of age. That prediction arises due to the interaction of several variables, interactions that are not interpretable from one-dimensional plots such as this. As we will see in subsequent chapters, such complexities in data are well suited to machine learning, but not without a cost. That cost is understanding the "bias–variance tradeoff" and understanding machine learning with sufficient rigor for its decisions to be defensible.

6 Summary

In this chapter, we have identified some of the key elements of supervised machine learning. Supervised machine learning

1. is an algorithmic approach to statistical inference which, crucially, does not depend on a data generation process;
2. estimates a parameterized map between inputs and outputs, with the functional form defined by the methodology such as a neural network or a random forest;
3. automates model selection, using regularization and ensemble averaging techniques to iterate through possible models and arrive at a model with the best out-of-sample performance; and
4. is often well suited to large sample sizes of high-dimensional non-linear covariates.

The emphasis on out-of-sample performance, automated model selection, and absence of a pre-determined parametric data generation process is really the key to machine learning being a more robust approach than many parametric, financial econometrics techniques in use today. The key to adoption of machine learning in finance is the ability to run machine learners alongside their parametric counterparts, observing over time the differences and limitations of parametric modeling based on in-sample fitting metrics. Statistical tests must be used to characterize the data and guide the choice of algorithm, such as, for example, tests for stationary. See Dixon and Halperin (2019) for a checklist and brief but rounded discussion of some of the challenges in adopting machine learning in the finance industry.

Capacity to readily exploit a wide form of data is their other advantage, but only if that data is sufficiently high quality and adds a new source of information. We close this chapter with a reminder of the failings of forecasting models during the financial crisis of 2008 and emphasize the importance of avoiding siloed data extraction. The application of machine learning requires strong scientific reasoning skills and is not a panacea for commoditized and automated decision-making.

7 Exercises

Exercise 1.1: Market Game**
Suppose that two players enter into a market game. The rules of the game are as follows: Player 1 is the market maker, and Player 2 is the market taker. In each round, Player 1 is provided with information $\boldsymbol{x}$, and must choose and declare a value $\alpha \in (0, 1)$ that determines how much it will pay out if a binary event G occurs in the round. $G \sim Bernoulli(p)$, where $p = g(\boldsymbol{x}|\boldsymbol{\theta})$ for some unknown parameter $\boldsymbol{\theta}$.

Player 2 then enters the game with a \$1 payment and chooses one of the following payoffs:

$$V_1(G, p) = \begin{cases} \frac{1}{\alpha} & \text{with probability } p \\ 0 & \text{with probability } (1 - p) \end{cases}$$

or

$$V_2(G, p) = \begin{cases} 0 & \text{with probability } p \\ \frac{1}{(1-\alpha)} & \text{with probability } (1 - p) \end{cases}$$

1. Given that α is known to Player 2, state the strategy[9] that will give Player 2 an expected payoff, over multiple games, of \$1 without knowing p.
2. Suppose now that p is known to both players. In a given round, what is the optimal choice of α for Player 1?
3. Suppose Player 2 knows with complete certainty, that G will be 1 for a particular round, what will be the payoff for Player 2?
4. Suppose Player 2 has complete knowledge in rounds $\{1, \ldots, i\}$ and can reinvest payoffs from earlier rounds into later rounds. Further suppose without loss of generality that $G = 1$ for each of these rounds. What will be the payoff for Player 2 after i rounds? You may assume that the each game can be played with fractional dollar costs, so that, for example, if Player 2 pays Player 1 \$1.5 to enter the game, then the payoff will be $1.5V_1$.

Exercise 1.2: Model Comparison**
Recall Example 1.2. Suppose additional information was added such that it is no longer possible to predict the outcome with 100% probability. Consider Table 1.8 as the results of some experiment.

Now if we are presented with $\boldsymbol{x} = (1, 0)$, the result could be B or C. Consider three different models applied to this value of $\boldsymbol{x}$ which encode the value A, B, or C.

[9] The strategy refers the choice of weight if Player 2 is to choose a payoff $V = wV_1 + (1 - w)V_2$, i.e. a weighted combination of payoffs V_1 and V_2.

Table 1.8 Sample model data

G	$\boldsymbol{x}$
A	(0, 1)
B	(1, 1)
B	(1, 0)
C	(1, 0)
C	(0, 0)

$$f((1,0)) = (0,1,0), \text{ Predicts B with 100\% certainty.} \tag{1.23}$$

$$g((1,0)) = (0,0,1), \text{ Predicts C with 100\% certainty.} \tag{1.24}$$

$$h((1,0)) = (0,0.5,0.5), \text{ Predicts B or C with 50\% certainty.} \tag{1.25}$$

1. Show that each model has the same total absolute error, over the samples where $\boldsymbol{x} = (1, 0)$.
2. Show that all three models assign the same average probability to the values from Table 1.8 when $\boldsymbol{x} = (1, 0)$.
3. Suppose that the market game in Exercise 1 is now played with models f or g. B or C each triggers two separate payoffs, V_1 and V_2, respectively. Show that the losses to Player 1 are unbounded when $\boldsymbol{x} = (1, 0)$ and $\alpha = 1 - p$.
4. Show also that if the market game in Exercise 1 is now played with model h, the losses to Player 1 are bounded.

Exercise 1.3: Model Comparison**

Example 1.1 and the associated discussion alluded to the notion that some types of models are more common than others. This exercise will explore that concept briefly.

Recall Table 1.1 from Example 1.1:

G	$\boldsymbol{x}$
A	(0, 1)
B	(1, 1)
C	(1, 0)
C	(0, 0)

For this exercise, consider two models "similar" if they produce the same projections for G when applied to the values of $\boldsymbol{x}$ from Table 1.1 with probability strictly greater than 0.95.

In the following subsections, the goal will be to produce sets of mutually dissimilar models that all produce Table 1.1 with a given likelihood.

1. How many similar models produce Table 1.1 with likelihood 1.0?
2. Produce at least 4 dissimilar models that produce Table 1.1 with likelihood 0.9.
3. How many dissimilar models can produce Table 1.1 with likelihood exactly 0.95?

Exercise 1.4*: Likelihood Estimation
When the data is i.i.d., the negative of log-likelihood function (the "error function") for a binary classifier is the *cross-entropy*

$$E(\boldsymbol{\theta}) = -\sum_{i=1}^{n} G_i ln\ (g_1(\mathbf{x}_i \mid \boldsymbol{\theta})) + (1 - G_i) ln\ (g_0(\mathbf{x}_i \mid \boldsymbol{\theta})).$$

Suppose now that there is a probability π_i that the class label on a training data point $\mathbf{x}_i$ has been correctly set. Write down the error function corresponding to the negative log-likelihood. Verify that the error function in the above equation is obtained when $\pi_i = 1$. Note that this error function renders the model robust to incorrectly labeled data, in contrast to the usual least squares error function.

Exercise 1.5: Optimal Action**
Derive Eq. 1.17 by setting the derivative of Eq. 1.16 with respect to the time-t action u_t to zero. Note that Eq. 1.17 gives a non-parametric expression for the optimal action u_t in terms of a ratio of two conditional expectations. To be useful in practice, the approach might need some further modification as you will use in the next exercise.

Exercise 1.6*: Basis Functions**
Instead of non-parametric specifications of an optimal action in Eq. 1.17, we can develop a *parametric* model of optimal action. To this end, assume we have a set of basic functions $\psi_k(S)$ with $k = 1, \ldots, K$. Here K is the total number of basis functions—the same as the dimension of your model space.

We now define the optimal action $u_t = u_t(S_t)$ in terms of coefficients θ_k of expansion over basis functions Ψ_k (for example, we could use spline basis functions, Fourier bases, etc.) :

$$u_t = u_t(S_t) = \sum_{k=1}^{K} \theta_k(t) \Psi_k(S_t).$$

Compute the optimal coefficients $\theta_k(t)$ by substituting the above equation for u_t into Eq. 1.16 and maximizing it with respect to a set of weights $\theta_k(t)$ for a t-th time step.

Appendix

Answers to Multiple Choice Questions

Question 1
Answer: 1, 2.

Answer 3 is incorrect. While it is true that unsupervised learning does not require a human supervisor to train the model, it is false to presume that the approach is superior.

Answer 4 is incorrect. Reinforcement learning cannot be viewed as a generalization of supervised learning to Markov Decision Processes. The reason is that reinforcement learning uses rewards to reinforce decisions, rather than labels to define the correct decision. For this reason, reinforcement learning uses a weaker form of supervision.

Question 2
Answer: 1,2,3.

Answer 4 is incorrect. Two separate binary models $\{g_i^{(1)}(X|\theta)\}_{i=0}^1$ and $\{g_i^{(2)}(X|\theta)\}_{i=0}^1$ will, in general, not produce the same output as a single, multi-class, model $\{g_i(X|\theta)\}_{i=0}^3$. Consider, as a counter example, the logistic models $g_0^{(1)} = g_0(X|\theta_1) = \frac{\exp\{-X^T\theta_1\}}{1+\exp\{-X^T\theta_1\}}$ and $g_0^{(2)} = g_0(X|\theta_2) = \frac{\exp\{-X^T\theta_2\}}{1+\exp\{-X^T\theta_2\}}$, compared with the multi-class model

$$g_i(X|\boldsymbol{\theta}') = \text{softmax}(\exp\{X^T\boldsymbol{\theta}'\}) = \frac{\exp\{(X^T\boldsymbol{\theta}')_i\}}{\sum_{k=0}^K \exp\{(X^T\boldsymbol{\theta}')_k\}}. \tag{1.26}$$

If we set $\theta_1 = \boldsymbol{\theta}'_0 - \boldsymbol{\theta}'_1$ and $\boldsymbol{\theta}'_2 = \boldsymbol{\theta}'_3 = 0$, then the multi-class model is equivalent to Model 1. Similarly if we set $\theta_2 = \boldsymbol{\theta}'_2 - \boldsymbol{\theta}'_3$ and $\boldsymbol{\theta}'_0 = \boldsymbol{\theta}'_1 = 0$, then the multi-class model is equivalent to Model 2. However, we cannot simultaneously match the outputs of Model 1 and Model 2 with the multi-class model.

Question 3
Answer: 1,2,3.

Answer 4 is incorrect. The layers in a deep recurrent network provide more expressibility between each lagged input and the hidden state variable, but are unrelated to the amount of memory in the network. The hidden layers in any multilayered perceptron are not the hidden state variables in our time series model. It is the degree of unfolding, i.e. number of hidden state vectors which determines the amount of memory in any recurrent network.

Question 4
Answer: 2.

References

Akaike, H. (1973). *Information theory and an extension of the maximum likelihood principle* (pp. 267–281).

Akcora, C. G., Dixon, M. F., Gel, Y. R., & Kantarcioglu, M. (2018). Bitcoin risk modeling with blockchain graphs. *Economics Letters, 173*(C), 138–142.

Arnold, V. I. (1957). *On functions of three variables* (Vol. 114, pp. 679–681).

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics, 31*, 307–327.

Box, G. E. P., & Jenkins, G. M. (1976). *Time series analysis, forecasting, and control*. San Francisco: Holden-Day.

Box, G. E. P., Jenkins, G. M., & Reinsel, G. C. (1994). *Time series analysis, forecasting, and control* (third ed.). Englewood Cliffs, NJ: Prentice-Hall.

Breiman, L. (2001). Statistical modeling: the two cultures (with comments and a rejoinder by the author). *Statistical Science, 16*(3), 199–231.

Cont, R., & de Larrard, A. (2013). Price dynamics in a Markovian limit order market. *SIAM Journal on Financial Mathematics, 4*(1), 1–25.

de Prado, M. (2018). *Advances in financial machine learning*. Wiley.

de Prado, M. L. (2019). Beyond econometrics: A roadmap towards financial machine learning. *SSRN*. Available at SSRN: https://ssrn.com/abstract=3365282 or http://dx.doi.org/10.2139/ssrn.3365282.

DeepMind (2016). DeepMind AI reduces Google data centre cooling bill by 40%. https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.

DeepMind (2017). The story of AlphaGo so far. https://deepmind.com/research/alphago/.

Dhar, V. (2013, December). Data science and prediction. *Commun. ACM, 56*(12), 64–73.

Dixon, M. (2018a). A high frequency trade execution model for supervised learning. *High Frequency, 1*(1), 32–52.

Dixon, M. (2018b). Sequence classification of the limit order book using recurrent neural networks. *Journal of Computational Science, 24*, 277–286.

Dixon, M., & Halperin, I. (2019). *The four horsemen of machine learning in finance*.

Dixon, M., Polson, N., & Sokolov, V. (2018). Deep learning for spatio-temporal modeling: Dynamic traffic flows and high frequency trading. *ASMB*.

Dixon, M. F., & Polson, N. G. (2019, Mar). Deep fundamental factor models. *arXiv e-prints*, arXiv:1903.07677.

Dyhrberg, A. (2016). Bitcoin, gold and the dollar – a GARCH volatility analysis. *Finance Research Letters*.

Elman, J. L. (1991, Sep). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning, 7*(2), 195–225.

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., et al. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *Nature, 542*(7639), 115–118.

Flood, M., Jagadish, H. V., & Raschid, L. (2016). Big data challenges and opportunities in financial stability monitoring. *Financial Stability Review,* (20), 129–142.

Gomber, P., Koch, J.-A., & Siering, M. (2017). Digital finance and fintech: current research and future research directions. *Journal of Business Economics, 7*(5), 537–580.

Gottlieb, O., Salisbury, C., Shek, H., & Vaidyanathan, V. (2006). Detecting corporate fraud: An application of machine learning. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.7470.

Graves, A. (2012). *Supervised sequence labelling with recurrent neural networks*. Studies in Computational intelligence. Heidelberg, New York: Springer.

Gu, S., Kelly, B. T., & Xiu, D. (2018). *Empirical asset pricing via machine learning*. Chicago Booth Research Paper 18–04.

Harvey, C. R., Liu, Y., & Zhu, H. (2016). . . . and the cross-section of expected returns. *The Review of Financial Studies, 29*(1), 5–68.

Hornik, K., Stinchcombe, M., & White, H. (1989, July). Multilayer feedforward networks are universal approximators. *Neural Netw., 2*(5), 359–366.

Kearns, M., & Nevmyvaka, Y. (2013). Machine learning for market microstructure and high frequency trading. *High Frequency Trading - New Realities for Traders*.

Kercheval, A., & Zhang, Y. (2015). Modeling high-frequency limit order book dynamics with support vector machines. *Journal of Quantitative Finance, 15*(8), 1315–1329.

Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Dokl. Akad. Nauk SSSR, 114*, 953–956.

Kubota, T. (2017, January). Artificial intelligence used to identify skin cancer.

Kullback, S., & Leibler, R. A. (1951, 03). On information and sufficiency. *Ann. Math. Statist., 22*(1), 79–86.

McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (1955, August). A proposal for the Dartmouth summer research project on artificial intelligence. http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.

Philipp, G., & Carbonell, J. G. (2017, Dec). Nonparametric neural networks. *arXiv e-prints*, arXiv:1712.05440.

Philippon, T. (2016). *The fintech opportunity*. CEPR Discussion Papers 11409, C.E.P.R. Discussion Papers.

Pinar Saygin, A., Cicekli, I., & Akman, V. (2000, November). Turing test: 50 years later. *Minds Mach., 10*(4), 463–518.

Poggio, T. (2016). Deep learning: mathematics and neuroscience. *A Sponsored Supplement to Science Brain-Inspired intelligent robotics: The intersection of robotics and neuroscience*, 9–12.

Shannon, C. (1948). A mathematical theory of communication. *Bell System Technical Journal, 27*.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.

Sirignano, J., Sadhwani, A., & Giesecke, K. (2016, July). Deep learning for mortgage risk. *ArXiv e-prints*.

Sirignano, J. A. (2016). Deep learning for limit order books. *arXiv preprint arXiv:1601.01987*.

Sovbetov, Y. (2018). Factors influencing cryptocurrency prices: Evidence from Bitcoin, Ethereum, Dash, Litcoin, and Monero. *Journal of Economics and Financial Analysis, 2*(2), 1–27.

Stein, H. (2012). Counterparty risk, CVA, and Basel III.

Turing, A. M. (1995). *Computers & thought*. Chapter Computing Machinery and Intelligence (pp. 11–35). Cambridge, MA, USA: MIT Press.

Wiener, N. (1964). *Extrapolation, interpolation, and smoothing of stationary time series*. The MIT Press.

Chapter 2
Probabilistic Modeling

This chapter introduces probabilistic modeling and reviews foundational concepts in Bayesian econometrics such as Bayesian inference, model selection, online learning, and Bayesian model averaging. We then develop more versatile representations of complex data with probabilistic graphical models such as mixture models.

1 Introduction

Not only is statistical inference from data intrinsically uncertain, but the type of data and relationships in the data that we seek to model are growing ever more complex. In this chapter, we turn to probabilistic modeling, a class of statistical models, which are broadly designed to characterize uncertainty and allow the expression of causality between variables. Probabilistic modeling is a meta-class of models, including generative modeling—a class of statistical inference models which maximizes the joint distribution, $p(X, Y)$, and Bayesian modeling, employing either maximum likelihood estimation or "fully Bayesian" inference. Probabilistic graphical models put the emphasis on causal modeling to simplify statistical inference of parameters from data. This chapter shall focus on the constructs of probabilistic modeling, as they relate to the application of both unsupervised and supervised machine learning in financial modeling.

While it seems natural to extend the previous chapters directly to a probabilistic neural network counterpart, it turns out that this does not develop the type of intuitive explanation of complex data that is needed in finance. It also turns out that neural networks are not a natural fit for probabilistic modeling. In other words, neural networks are well suited to pointwise estimation but lead to many difficulties in a probabilistic setting. In particular, they tend to be very data intensive—offsetting one of the major advantages of Bayesian modeling.

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_2

We will explore probabilistic modeling through the introduction of probabilistic graphical models—a data structure which is convenient for understanding the relationship between a multitude of different classes of models, both discriminative and generative. This representation will lead us neatly to Bayesian kernel learning, the subject of Chap. 3. We begin by introducing the reader to elementary topics in Bayesian modeling, which is a well-established approach for characterizing uncertainty in, for example, trading and risk modeling. Starting with simple probabilistic models of the data, we review some of the main constructs necessary to apply Bayesian methods in practice.

The application of probabilistic models to time series modeling, using filtering and hidden variables to dynamically represent the data is presented in Chap. 7.

Chapter Objectives

The key learning points of this chapter are:

- Apply Bayesian inference to data using simple probabilistic models;
- Understand how linear regression with probabilistic weights can be viewed as a simple probabilistic graphical model; and
- Develop more versatile representations of complex data with probabilistic graphical models such as mixture models and hidden Markov models.

Note that section headers ending with * are more mathematically advanced, often requiring some background in analysis and probability theory, and can be skipped by the less mathematically inclined reader.

2 Bayesian vs. Frequentist Estimation

Bayesian data analysis is distinctly different from classical (or "frequentist") analysis in its treatment of probabilities, and in its resulting treatment of model parameters when compared to classical parametric analysis.[1]

Bayesian analysts formulate probabilistic statements about uncertain events before collecting any additional evidence (i.e., "data"). These ex-ante probabilities (or, more generally, probability distributions plus underlying parameters) are called *priors*.

This notion of *subjective probabilities* is absent in classical estimation. In the classical world, all estimation and inference is based solely on observed data.

[1] Throughout the first part of this chapter we will largely remain within the realm of parametric analysis. However, we shall later see examples of Bayesian methods for non- and semi-parametric modeling.

Both Bayesian and classical econometricians aim to learn more about a set of parameters, say $\boldsymbol{\theta}$. In the classical mindset, $\boldsymbol{\theta}$ contains fixed but unknown elements, usually associated with an underlying population of interest (e.g., the mean and variance for credit card debt among US college students). Bayesians share with classicals the interest in $\boldsymbol{\theta}$ and the definition of the population of interest.

However, they assign ex ante a prior probability to $\boldsymbol{\theta}$, labeled $p(\boldsymbol{\theta})$, which usually takes the form of a probability distribution with "known" moments. For example, Bayesians might state that the aforementioned debt amount has a normal distribution with mean \$3000 and standard deviation of \$1500. This prior may be based on previous research, related findings in the published literature, or it may be completely arbitrary. In any case, it is an inherently subjective construct.

Both schools then develop a theoretical framework that relates $\boldsymbol{\theta}$ to observed data, say a "dependent variable" $\mathbf{y}$, and a matrix of explanatory variables $\mathbf{X}$. This relationship is formalized via a likelihood function, say $p(\mathbf{y} \mid \boldsymbol{\theta}, \mathbf{X})$ to stay with Bayesian notation. To stress, this likelihood function takes the exact same analytical form for both schools.

The classical analyst then collects a sample of observations from the underlying population of interest and, combining these data with the formulated statistical model, produces an estimate of $\boldsymbol{\theta}$, say $\hat{\boldsymbol{\theta}}$. Any and all uncertainty surrounding the accuracy of this estimate is solely related to the notion that results are based on a sample, not data for the entire population. A different sample (of equal size) may produce slightly different estimates. Classicals express this uncertainty via "standard errors" assigned to each element of $\hat{\boldsymbol{\theta}}$. They also have a strong focus on the behavior of $\hat{\boldsymbol{\theta}}$ as the sample size increases. The behavior of estimators under increasing sample size falls under the heading of "asymptotic theory."

The properties of most estimators in the classical world can only be assessed "asymptotically," i.e. are only understood for the hypothetical case of an infinitely large sample. Also, virtually all specification tests used by frequentists hinge on asymptotic theory. This is a major limitation when the data size is finite.

Bayesians, in turn, combine prior and likelihood via Bayes' rule to derive the *posterior distribution* of $\boldsymbol{\theta}$ as

$$p(\boldsymbol{\theta} \mid \mathbf{y}, \mathbf{X}) = \frac{p(\boldsymbol{\theta}, \mathbf{y} \mid \mathbf{X})}{p(\mathbf{y} \mid \mathbf{X})} = \frac{p(\boldsymbol{\theta})\, p(\mathbf{y} \mid \boldsymbol{\theta}, \mathbf{X})}{p(\mathbf{y} \mid \mathbf{X})} \propto p(\boldsymbol{\theta})\, p(\mathbf{y} \mid \boldsymbol{\theta}, \mathbf{X}). \tag{2.1}$$

Bayesian Modeling

Bayesian modeling is not about point estimation of a parameter value, $\boldsymbol{\theta}$, but rather updating and sharpening our subjective beliefs (our "prior") about $\boldsymbol{\theta}$ from the sample data. Thus, the sample data should contribute to "learning" about $\boldsymbol{\theta}$.

Bayesian Learning

Simply put, *the posterior distribution is just an updated version of the prior*. More specifically, the posterior is proportional to the prior multiplied by the likelihood. The likelihood carries all the current information about the parameters and the data. If the data has high informational content (i.e., allows for substantial learning about $\boldsymbol{\theta}$), the posterior will generally look very different from the prior. In most cases, it is much "tighter" (i.e., has a much smaller variance) than the prior. There is no room in Bayesian analysis for the classical notions of "sampling uncertainty," and less a priori focus on the "asymptotic behavior" of estimators.[2]

Taking the Bayesian paradigm to its logical extreme, Duembgen and Rogers (2014) suggest to "estimate nothing." They propose the replacement of the industry-standard estimation-based paradigm of calibration with an approach based on Bayesian techniques, wherein a posterior is iteratively obtained from a prior, namely stochastic filtering and MCMC. Calibration attempts to estimate, and then uses the estimates as if they were known true values—ignoring the estimation error. On the contrary, an approach based on a systematic application of the Bayesian principle is consistent: "There is never any doubt about what we should be doing to hedge or to mark-to-market a portfolio of derivatives, and whatever we do today will be consistent with what we did before, and with what we will do in the future." Moreover, Bayesian model comparison methods enable one to easily compare models of very different types.

Marginal Likelihood

The term in the denominator of Eq. 2.1 is called the "marginal likelihood," it is not a function of $\boldsymbol{\theta}$, and can usually be ignored for most components of Bayesian analysis. Thus, we usually work only with the numerator (i.e., prior times likelihood) for inference about $\boldsymbol{\theta}$. From Eq. 2.1 we know that this expression is proportional ("$\propto$ ") to the actual posterior. However, the marginal likelihood is crucial for model comparison, so we will learn a few methods to derive it as a by-product of or following the actual posterior analysis. For some choices of prior and likelihood there exist analytical solutions for this term.

In summary, frequentists start with a "blank mind" regarding $\boldsymbol{\theta}$. They collect data to produce an estimate $\hat{\boldsymbol{\theta}}$. They formalize the characteristics and uncertainty of $\hat{\boldsymbol{\theta}}$ for a finite sample context (if possible) and a hypothetical large sample (asymptotic) case.

Bayesians collect data to *update a prior*, i.e. a pre-conceived probabilistic notion regarding $\boldsymbol{\theta}$.

[2] However, at times Bayesian analysis does rest on asymptotic results. Naturally, the general notion that a larger sample, i.e. more empirical information, is better than a small one also holds for Bayesian analysis.

3 Frequentist Inference from Data

Let us begin this section with a simple example which illustrates frequentist inference.

Example 2.1 Bernoulli Trials Example

Consider an experiment consisting of a single coin flip. We set the random variable Y to 0 if tails come up and 1 if heads come up. Then the probability density of Y is given by

$$p(y \mid \theta) = \theta^y (1-\theta)^{1-y},$$

where $\theta \in [0, 1]$ is the probability of heads showing up.

You will recognize Y as a *Bernoulli random variable*. We view p as a function of y, but parameterized by the given parameter θ, hence the notation, $p(y \mid \theta)$.

More generally, suppose that we perform n such independent experiments (tosses) on the same coin. Denote these n realizations of Y as

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \in \{0, 1\}^n,$$

where, for $1 \leq i \leq n$, y_i is the result of the ith toss. What is the probability density of $\mathbf{y}$?

Since the coin tosses are independent, the probability density of $\mathbf{y}$, i.e. the joint probability density of $y_1, y_2, \ldots, y_n$, is given by the product rule

$$p(\mathbf{y} \mid \theta) = p(y_1, y_2, \ldots, y_n \mid \theta) = \prod_{i=1}^{n} \theta^{y_i} (1-\theta)^{1-y_i} = \theta^{\sum y_i} (1-\theta)^{n - \sum y_i}.$$

Suppose that we have tossed the coin $n = 50$ times (performed $n = 50$ Bernoulli trials) and recorded the results of the trials as

```
0 0 1 0 0 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 0
```

How can we estimate θ given these data?

Both the frequentists and Bayesians regard the density $p(\mathbf{y} \mid \theta)$ as a *likelihood*. Bayesians maintain this notation, whereas frequentists reinterpret $p(\mathbf{y} \mid \theta)$, which is

a function of $\mathbf{y}$ (given the parameters $\boldsymbol{\theta}$: in our case, there is a single parameter, so θ is univariate, but this does not have to be the case) as a function of θ (given the specific sample $\mathbf{y}$), and write

$$\mathcal{L}(\theta) := \mathcal{L}(\theta \,|\, \mathbf{y}) := p(\mathbf{y} \,|\, \theta).$$

Notice that we have merely reinterpreted this probability density, whereas its functional form remains the same, in our case:

$$\mathcal{L}(\theta) = \theta^{\sum y_i}(1-\theta)^{n-\sum y_i}.$$

Likelihood

Likelihood is one of the seminal ideas of the frequentist school. It was introduced by one of its founding fathers, Sir Ronald Aylmer Fisher: "What has now appeared is that the mathematical concept of probability is ... inadequate to express our mental confidence or [lack of confidence] in making ... inferences, and that the mathematical quantity which usually appears to be appropriate for measuring our order of preference among different possible populations does not in fact obey the laws of probability. To distinguish it from probability, I have used the term 'likelihood' to designate this quantity..."—R.A. Fisher, *Statistical Methods for Research Workers*.

It is generally more convenient to work with the log of likelihood—the *log-likelihood*. Since ln is a monotonically increasing function of its argument, the same values of θ maximize the log-likelihood as the ones that maximize the likelihood.

$$\ln \mathcal{L}(\theta) = \ln\left\{\theta^{\sum y_i}(1-\theta)^{n-\sum y_i}\right\} = \left(\sum y_i\right)\ln\theta + \left(n - \sum y_i\right)\ln(1-\theta).$$

In order to find the value of θ that maximizes this expression, we differentiate with respect to θ and solve for the value of θ that sets the (partial) derivative to zero.

$$\frac{\partial}{\partial\theta}\ln\mathcal{L}(\theta) = \frac{\sum y_i}{\theta} + \frac{n - \sum y_i}{\theta - 1}.$$

Equating this to zero and solving for θ, we obtain the *maximum likelihood estimate* for θ:

$$\hat{\theta}_{\mathrm{ML}} = \frac{\sum y_i}{n}.$$

To confirm that this value does indeed *maximize* the log-likelihood, we take the second derivative with respect to θ,

$$\frac{\partial^2}{\partial \theta^2} \ln \mathcal{L}(\theta) = -\frac{\sum y_i}{\theta^2} - \frac{n - \sum y_i}{(\theta - 1)^2} < 0.$$

Since this quantity is strictly negative for all $0 < \theta < 1$, it is negative at $\hat{\theta}_{\mathrm{ML}}$, and we do indeed have a maximum.

Example 2.2 Bernoulli Trials Example (continued)

Note that $\hat{\theta}_{\mathrm{ML}}$ depends only on the sum of y_is, we can answer our question: if in a sequence of 50 coin tosses exactly twelve heads come up, then

$$\hat{\theta}_{\mathrm{ML}} = \frac{\sum y_i}{n} = \frac{12}{50} = 0.24.$$

A frequentist approach gives at a *single* value (a single "point") as our estimate, 0.24—in this sense we are performing *point estimation*. When we apply a Bayesian approach to the same problem, we shall see that the Bayesian estimate is a probability distribution, rather than a single point.

Despite some mathematical formalism, the answer is intuitively obvious. If we toss a coin fifty times, and out of those twelve times it lands with heads up, it is natural to estimate the probability of getting heads as $\frac{12}{50}$. It is encouraging that the result of our calculation agrees with our intuition and common sense.

4 Assessing the Quality of Our Estimator: Bias and Variance

When we obtained our maximum likelihood estimate, we plugged in a specific number for $\sum y_i$, 12. In this sense the estimator is an ordinary function. However, we could also view it as a function of the *random* sample,

$$\hat{\theta}_{\mathrm{ML}} = \frac{\sum Y_i}{n},$$

each Y_i being a random variable. A function of a random variable is itself a random variable, so we can compute its expectation and variance.

In particular, an expectation of the *error*

$$\mathbf{e} = \hat{\boldsymbol{\theta}} - \boldsymbol{\theta}$$

is known as *bias*,

$$\text{bias}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) = \mathbb{E}(\mathbf{e}) = \mathbb{E}\left[\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\right] = \mathbb{E}\left[\hat{\boldsymbol{\theta}}\right] - \mathbb{E}\left[\boldsymbol{\theta}\right].$$

As frequentists, we view the true value of θ as a single, deterministic, fixed point, so we take it outside of the expectation:

$$\text{bias}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) = \mathbb{E}\left[\hat{\boldsymbol{\theta}}\right] - \boldsymbol{\theta}.$$

In our case it is

$$\begin{aligned}\mathbb{E}[\hat{\theta}_{\text{ML}} - \theta] = \mathbb{E}[\hat{\theta}_{\text{ML}}] - \theta = \mathbb{E}\left[\frac{\sum Y_i}{n}\right] - \theta = \frac{1}{n}\sum \mathbb{E}[Y_i] - \theta \\ = \frac{1}{n} \cdot n(\theta \cdot 1 + (1-\theta) \cdot 0) - \theta = 0,\end{aligned}$$

we see that the bias is zero, so this particular maximum likelihood estimator is *unbiased* (otherwise it would be *biased*).

What about the variance of this estimator?

$$\text{Var}[\hat{\theta}_{\text{ML}}] = \text{Var}\left[\frac{\sum Y_i}{n}\right] \overset{\text{independence}}{=} \frac{1}{n^2}\sum \text{Var}[Y_i] = \frac{1}{n^2} \cdot n \cdot \theta(1-\theta) = \frac{1}{n}\theta(1-\theta),$$

and we see that the variance of the estimator depends on the *true* value of θ.

For multivariate $\boldsymbol{\theta}$, it is useful to examine the *error covariance matrix* given by

$$\mathbf{P} = \mathbb{E}[\mathbf{e}\mathbf{e}^\intercal] = \mathbb{E}\left[(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})^\intercal\right].$$

When estimating $\boldsymbol{\theta}$, our goal is to minimize the estimation error. This error can be expressed using loss functions. Supposing our parameter vector $\boldsymbol{\theta}$ takes values on some space Θ, a *loss function* $L(\boldsymbol{\theta})$ is a mapping from $\Theta \times \Theta$ into $\mathbb{R}$ which quantifies the "loss" incurred by estimating $\boldsymbol{\theta}$ with $\hat{\boldsymbol{\theta}}$.

We have already seen loss functions in earlier chapters, but we shall restate the definitions here for completeness. One frequently used loss function is the *absolute error*,

$$L_1(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := \|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\|_2 = \sqrt{(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})^\intercal(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})},$$

where $\|\cdot\|_2$ is the Euclidean norm (it coincides with the absolute value when $\Theta \subseteq \mathbb{R}$). One advantage of the absolute error is that it has the same units as $\boldsymbol{\theta}$.

We use the *squared error* perhaps even more frequently than the *absolute error*:

$$L_2(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := \|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\|_2^2 = (\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})^\intercal(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}).$$

While the squared error has the disadvantage compared to the absolute error of being expressed in *quadratic* units of $\boldsymbol{\theta}$, rather than the units of $\boldsymbol{\theta}$, it does not contain the cumbersome $\sqrt{\cdot}$ and is therefore easier to deal with mathematically.

The expected value of a loss function is known as the *statistical risk* of the estimator.

The statistical risks corresponding to the above loss functions are, respectively, the *mean absolute error*,

$$\mathrm{MAE}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := R_1(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := \mathbb{E}\left[L_1(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta})\right] := \mathbb{E}\left[\|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\|_2\right] = \mathbb{E}\left[\sqrt{(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})^{\intercal}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})}\right],$$

and, by far the most commonly used, *mean squared error (MSE)*,

$$\mathrm{MSE}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := R_2(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) := \mathbb{E}\left[L_2(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta})\right] := \mathbb{E}\left[\|\hat{\boldsymbol{\theta}} - \boldsymbol{\theta}\|_2^2\right] = \mathbb{E}\left[(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})^{\intercal}(\hat{\boldsymbol{\theta}} - \boldsymbol{\theta})\right].$$

The square root of the mean squared error is called the *root mean squared error (RMSE)*. The *minimum mean squared error (MMSE)* estimator is the estimator that minimizes the mean squared error.

5 The Bias–Variance Tradeoff (Dilemma) for Estimators

It can easily be shown that the mean squared error separates into a variance and bias term:

$$\mathrm{MSE}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta}) = \mathrm{trVar}\left[\hat{\boldsymbol{\theta}}\right] + \|\mathrm{bias}(\hat{\boldsymbol{\theta}}, \boldsymbol{\theta})\|_2^2,$$

where $tr(\cdot)$ is the trace operator. In the case of a scalar θ, this expression simplifies to

$$\mathrm{MSE}(\hat{\theta}, \theta) = \mathrm{Var}\left[\hat{\theta}\right] + \mathrm{bias}(\hat{\theta}, \theta)^2.$$

In other words, the MSE is equal to the sum of the variance of the estimator and the squared bias.

The *bias–variance tradeoff* or *bias–variance dilemma* consists in the need to minimize these two sources of error, the variance and bias of an estimator, in order to minimize the mean squared error. Sometimes there is a tradeoff between minimizing bias and minimizing variance to achieve the least possible MSE. The concept of a bias–variance tradeoff in machine learning will be revisited in Chap. 4, within the context of statistical learning theory.

6 Bayesian Inference from Data

As before, let $\boldsymbol{\theta}$ be the parameter of some statistical model and let $\mathbf{y} = y_1, \ldots, y_n$ be n i.i.d. observations of some random variable Y. We capture our subjective assumptions about the model parameter $\boldsymbol{\theta}$, before observing the data, in the form of a prior probability distribution $p(\boldsymbol{\theta})$. Bayes' rule converts a prior probability into a posterior probability by incorporating the *evidence* provided by the observed data:

$$p(\boldsymbol{\theta} \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid \boldsymbol{\theta})}{p(\mathbf{y})} p(\boldsymbol{\theta})$$

allows us to evaluate the uncertainty in $\boldsymbol{\theta}$ after we have observed $\mathbf{y}$. This uncertainty is characterized by the posterior probability $p(\boldsymbol{\theta} \mid \mathbf{y})$. The effect of the observed data is expressed through $p(\mathbf{y} \mid \boldsymbol{\theta})$—a function of $\boldsymbol{\theta}$ referred to as the *likelihood function*. It expresses how likely the observed dataset was generated by a model with parameter $\boldsymbol{\theta}$.

Let us summarize some of the notation that will be important:

- The prior is $p(\boldsymbol{\theta})$;
- The likelihood is $p(\mathbf{y} \mid \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta})$, since the data is i.i.d.;
- The marginal likelihood $p(\mathbf{y}) = \int p(\mathbf{y} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta}$ is the likelihood with the dependency on $\boldsymbol{\theta}$ marginalized out; and
- The posterior is $p(\boldsymbol{\theta} \mid \mathbf{y})$.

Bayesian Inference

Informally, Bayesian inference involves the following steps:

1. Formulate your statistical model as a collection of probability distributions conditional on different values for a parameter θ, about which you wish to learn;
2. Organize your beliefs about θ into a (prior) probability distribution;
3. Collect the data and insert them into the family of distributions given in Step 1;
4. Use Bayes' rule to calculate your new beliefs about θ; and
5. Criticize your model and revise your modeling assumptions.

The following example shall illustrate the application of Bayesian inference for the Bernoulli parameter θ.

Example 2.3 Bernoulli Trials Example (continued)

θ is a probability, so it is bounded and must belong to the interval $[0, 1]$. We could assume that all values of θ in $[0, 1]$ are equally likely. Thus our prior could be that θ is uniformly distributed on $[0, 1]$, i.e. $\theta \sim \mathcal{U}(a = 0, b = 1)$.

This assumption would constitute an application of *Laplace's principle of indifference*, also known as the *principle of insufficient reason*: when faced with multiple possibilities, whose probabilities are unknown, assume that the probabilities of all possibilities are equal.

In the context of Bayesian estimation, applying Laplace's principle of indifference constitutes what is known as an *uninformative prior*. Our goal is, however, not to rely too much on the prior, but use the likelihood to proceed to a posterior based on new information.

The pdf of the uniform distribution, $\mathcal{U}(a, b)$, is given by

$$p(\theta) = \frac{1}{b-a}$$

if $\theta \in [a, b]$ and zero elsewhere. In our case, $a = 0, b = 1$, and so our uninformative uniform prior is given by

$$p(\theta) = 1, \ \forall \theta \in [0, 1].$$

Let us derive the posterior based on this prior assumption. Bayes' theorem tells us that

$$\text{posterior} \propto \text{likelihood} \cdot \text{prior},$$

where $\propto$ stands for "proportional to," so the left- and right-hand side are equal up to a normalizing constant which depends on the data but not on θ. The posterior is

$$p(\theta \mid x_{1:n}) \propto p(x_{1:n} \mid \theta) p(\theta) = \theta^{\sum x_i} (1-\theta)^{n - \sum x_i} \cdot 1.$$

If the prior is uniform, i.e. $p(\theta) = 1$, then after $n = 5$ trials we see from the data that

$$p(\theta \mid x_{1:n}) \propto \theta (1-\theta)^4. \tag{2.2}$$

After 10 trials we have

(continued)

Example 2.3 (continued)

$$p(\theta \mid x_{1:n}) \propto \theta(1-\theta)^4 \times \theta(1-\theta)^4 = \theta^2(1-\theta)^8. \tag{2.3}$$

From the shape of the resulting pdf, we recognize it as the pdf of the Beta distribution[a]

$$\text{Beta}\left(\theta \mid \sum x_i, n - \sum x_i\right),$$

and we immediately know that the missing normalizing constant factor is

$$\frac{1}{B\left(\sum x_i, n - \sum x_i\right)} = \frac{\Gamma\left(\sum x_i\right)\Gamma\left(n - \sum x_i\right)}{\Gamma(n)}.$$

Let us now assume that we have tossed the coin fifty times and, out of those fifty coin tosses, we get heads on twelve. Then our posterior distribution becomes

$$\theta \mid x_{1:n} \sim \text{Beta}(\theta \mid 12, 38).$$

Then, from the properties of this distribution,

$$\mathbb{E}[\theta \mid x_{1:n}] = \frac{\sum x_i}{\sum x_i + (n - \sum x_i)} = \frac{\sum x_i}{n} = \frac{12}{12+38} = \frac{12}{50} = 0.24,$$

$$\text{Var}[\theta \mid x_{1:n}] = \frac{\left(\sum x_i\right)\left(n - \sum x_i\right)}{\left(\sum x_i + n - \sum x_i\right)^2\left(\sum x_i + n - \sum x_i + 1\right)} \tag{2.4}$$

$$= \frac{n\sum x_i - \left(\sum x_i\right)^2}{n^2(n+1)} = \frac{12 \cdot 38}{(12+38)^2(12+38+1)} = \frac{456}{127500} = 0.00357647058. \tag{2.5}$$

The standard deviation being, in units of probability, $\sqrt{\frac{456}{127500}} = 0.05980360012$.

Notice that the mean of the posterior, 0.24, matches the frequentist maximum likelihood estimate of θ, $\hat{\theta}_{\text{ML}}$, and our intuition. Again, it is not unreasonable to assume that the probability of getting heads is 0.24 if we observe heads on twelve out of fifty coin tosses.

[a] The function's argument is now θ, not x_i, so it is not the pdf of a Bernoulli distribution.

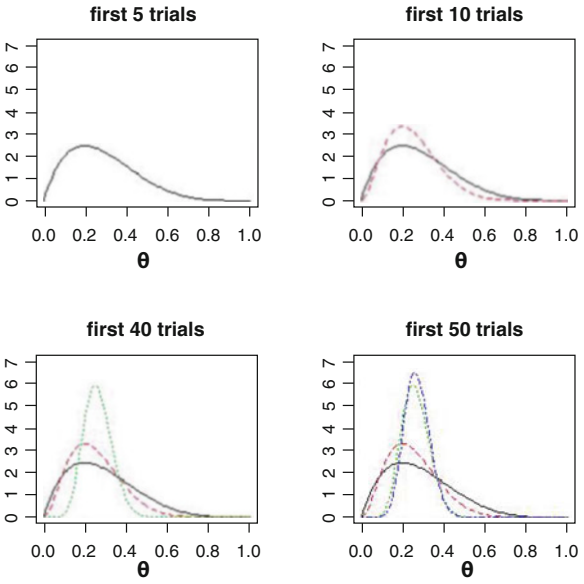


Fig. 2.1 The posterior distribution of θ against successive numbers of trials. The x-axis shows the values of theta. The shape of the distribution tightens as the Bayesian model is observed to "learn"

Note that we did not need to evaluate the marginal likelihood in the example above, only the θ dependent terms were evaluated for the purpose of the plot. Thus each plot in Fig. 2.1 is only representative of the posterior up to a scaling.

! The Principle of Indifference

In practice, the principle of indifference should be used with great care, as we are assuming a property of the data strictly greater than we know. Saying "the probabilities of the outcomes are equally likely" contains strictly more information than "I don't know what the probabilities of the outcomes are."

If someone tosses a coin and then covers it with her hand, asking you, "heads or tails?" it is probably relatively sensible to assume that the two possibilities are equally likely, effectively assuming that the coin is unbiased.

If an investor asks you, "Will the stock price of XYZ increase?" you should think twice before applying Laplace's principle of indifference and replying "Well, there is a 50% chance that XYZ will grow, you can either long or short XYZ." Clearly there are other important considerations such as the amount by which the stock could increase versus decrease, limits on portfolio exposure to market risk factors, and anticipation of other market events such as earnings announcements. In other words, the implications of going long or short will not necessarily be equal.

6.1 A More Informative Prior: The Beta Distribution

Continuing with the above example, let us question our prior. Is it somewhat *too* uninformative? After all, most coins in the world are probably close to being unbiased. We could use a Beta(α, β) prior instead of the Uniform prior. Picking $\alpha = \beta = 2$, for example, will give a distribution on $[0, 1]$ centered on $\frac{1}{2}$, incorporating a prior assumption that the coin is unbiased.

The pdf of this prior is given by

$$p(\theta) = \frac{1}{B(\alpha, \beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}, \forall\theta \in [0, 1],$$

and so the posterior becomes

$$\begin{aligned} p(\theta \mid x_{1:n}) &\propto p(x_{1:n} \mid \theta)p(\theta) \\ &= \theta^{\sum x_i}(1-\theta)^{n-\sum x_i} \cdot \frac{1}{B(\alpha, \beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1} \propto \theta^{(\alpha+\sum x_i)-1}(1-\theta)^{(\beta+n-\sum x_i)-1}, \end{aligned}$$

which we recognize as a pdf of the distribution

$$\text{Beta}\left(\theta \mid \alpha + \sum x_i, \beta + n - \sum x_i\right).$$

Why did we pick this prior distribution? One reason is that its pdf is defined over the compact interval $[0, 1]$, unlike, for example, the normal distribution, which has tails extending to $-\infty$ and $+\infty$. Another reason is that we are able to choose parameters which center the pdf at $\theta = \frac{1}{2}$, incorporating the prior assumption that the coin is unbiased.

If we initially assume a Beta$(\theta \mid \alpha = 2, \beta = 2)$ prior, then the posterior expectation is

$$\begin{aligned} \mathbb{E}[\theta \mid x_{1:n}] &= \frac{\alpha + \sum x_i}{\alpha + \sum x_i + \beta + n - \sum x_i} = \frac{\alpha + \sum x_i}{\alpha + \beta + n} \\ &= \frac{2 + 12}{2 + 2 + 50} = \frac{7}{27} \approx 0.259. \end{aligned}$$

Notice that both the prior and posterior belong to the same probability distribution family. In Bayesian estimation theory we refer to such prior and posterior as *conjugate distributions* (with respect to this particular likelihood function).

Unsurprisingly, since now our prior assumption is that the coin is unbiased, $\frac{12}{50} < \mathbb{E}[\theta \mid x_{1:n}] < \frac{1}{2}$.

Perhaps surprisingly, we are also somewhat more certain about the posterior (its variance is smaller) than when we assumed the uniform prior.

Notice that the results of Bayesian estimation are sensitive—to varying degree in each specific case—to the choice of prior distribution:

$$p(\theta \mid \alpha, \beta) = \frac{(\alpha + \beta - 1)!}{(\alpha - 1)!(\beta - 1)!}\theta^{\alpha-1}(1-\theta)^{\beta-1} = \Gamma(\alpha, \beta)\theta^{\alpha-1}(1-\theta)^{\beta-1}. \tag{2.6}$$

So for the above example, this marginal likelihood would be evaluated with $\alpha = 13$ and $\beta = 39$ since there are 12 observed 1s and 38 observed 0s.

6.2 Sequential Bayesian updates

In the previous section we saw that, starting with the prior

$$\text{Beta}\,(\theta \mid \alpha, \beta)\,,$$

we arrived at the Beta-distributed posterior,

$$\text{Beta}\left(\theta \mid \alpha + \sum x_i, \beta + n - \sum x_i\right).$$

What would happen if, instead of observing all twelve coin tosses at once, we (i) considered each coin toss in turn; (ii) obtained our posterior; and (iii) used that posterior as a prior for an update based on the information from the next coin toss?

The above two formulae give the answer to this question. We start with our initial prior,

$$\text{Beta}\,(\theta \mid \alpha, \beta)\,,$$

then, substituting $n = 1$ into the second formula, we get

$$\text{Beta}\,(\theta \mid \alpha + x_1, \beta + 1 - x_1)\,.$$

Using this posterior as a prior before the second coin toss, we obtain the next posterior as

$$\text{Beta}\,(\theta \mid \alpha + x_1 + x_2, \beta + 2 - x_1 - x_2)\,.$$

Proceeding along these lines, after all ten coin tosses, we end up with

$$\text{Beta}\left(\theta \mid \alpha + \sum x_i, \beta + n - \sum x_i\right),$$

the same result that we would have attained if we processed all ten coin tosses as a single "batch," as we did in the previous section.

This insight forms the basis for a *sequential* or *iterative* application of Bayes' theorem—sequential Bayesian updates—the foundation for real-time *Bayesian filtering*. In machine learning, this mechanism for updating our beliefs in response to new data is referred to as "online learning."

6.2.1 Online Learning

An important aspect of Bayesian learning is the capacity to update the posterior in response to the arrival of new data, $\mathbf{y}'$. The posterior over $\mathbf{y}$ now becomes the prior, and the new posterior is updated to

$$p(\theta \mid \mathbf{y}', \mathbf{y}) = \frac{p(\mathbf{y}' \mid \theta) p(\theta \mid \mathbf{y})}{\int_{\theta \in \Theta} p(\mathbf{y}' \mid \theta) p(\theta \mid \mathbf{y}) d\theta}. \tag{2.7}$$

6.2.2 Prediction

In auto-correlated data, often encountered in financial econometrics, it is common to use Bayesian models for prediction. We can write that the density of the new predicted value y' given the previous data y is the expected value of the likelihood of the new data under the posterior density $p(\theta \mid y)$:

$$p(y' \mid y) = \mathbb{E}_{\theta \mid y}[p(y' \mid y, \theta)] = \int_{\theta \in \Theta} p(y' \mid y, \theta) p(\theta \mid y) d\theta. \tag{2.8}$$

? Multiple Choice Question 1

Which of the following statements are true:

1. A frequentist performs statistical inference by finding the best fit parameters. The Bayesian finds the distribution of the parameters assuming a prior.
2. Frequentist inference can be regarded as a special case of Bayesian inference when the prior is a Dirac delta-function.
3. Bayesian inference is well suited to online learning, an experimental design under which the model is continuously updated as new data arrives.
4. Prediction, under Bayesian inference, is the conditional expectation of the predicted variable under the posterior distribution of the parameter.

6.3 Practical Implications of Choosing a Classical or Bayesian Estimation Framework

If the sample size is large and the likelihood function "well-behaved" (which usually means a simple function with a clear maximum, plus a small dimension for $\boldsymbol{\theta}$), classical and Bayesian analysis are essentially on the same footing and will produce virtually identical results. This is because the likelihood function and empirical data will dominate any prior assumptions in the Bayesian approach.

If the sample size is large but the dimensionality of $\boldsymbol{\theta}$ is high and the likelihood function is less tractable (which usually means highly non-linear, with local maxima, flat spots, etc.), a Bayesian approach may be preferable purely from a computational standpoint. It can be very difficult to attain reliable estimates via maximum likelihood estimation (MLE) techniques, but it is usually straightforward to derive a posterior distribution for the parameters of interest using Bayesian estimation approaches, which often operate via sequential draws from known distributions.

If the sample size is small, Bayesian analysis can have substantial advantages over a classical approach. First, Bayesian results do not depend on asymptotic theory to hold for their interpretability. Second, the Bayesian approach combines the sparse data with subjective priors. Well-informed priors can increase the accuracy and efficiency of the model. Conversely, of course, poorly chosen priors[3] can produce misleading posterior inference in this case. Thus, under small sample conditions, the choice between Bayesian and classical estimation often distills to a choice between trusting the asymptotic properties of estimators and trusting one's priors.

7 Model Selection

Beyond the inference challenges described above, there are a number of problems with the classical approach to model selection which Bayesian statistics solves. For example, it has been shown by Breiman (2001) that the following three linear regression models have a residual sum of squares (RSS) which are all within 1%:

$$\text{Model 1} \quad \hat{Y} = 2.1 + 3.8X_3 - 0.6X_8 + 83.2X_{13} - 2.1X_{17} + 3.2X_{27}, \tag{2.9}$$

$$\text{Model 2} \quad \hat{Y} = -8.9 + 4.6X_5 + 0.01X_6 + 12.0X_{15} + 17.5X_{21} + 0.2X_{22}, \tag{2.10}$$

$$\text{Model 3} \quad \hat{Y} = -76.7 + 9.3X_2 + 22.0X_7 - 13.2X_8 + 3.4X_{11} + 7.2X_{28}. \tag{2.11}$$

[3]For example, priors that place substantial probability mass on practically infeasible ranges of θ—this often happens inadvertently when parameter transformations are involved in the analysis.

You could, for example, think of each model being used to find the fair price of an asset Y, where each X_i are the contemporaneous (i.e., measured at the same time) firm characteristics.

- Which model is better?
- How would your interpretation of which variables are the most important change between models?
- Would you arrive at different conclusions about the market signals if you picked, say, Model 1 versus Model 2?
- How would you eliminate some of the ambiguity resulting from this outcome of statistical inference?

Of course one direction is to simply analyze the F-scores of each independent variable and select the model which has the most statistically significant fitted coefficients. But this is unlikely to reliably discern the models when the fitted coefficients are comparable in statistical significance.

It is well known that the goodness-of-fit measures, such as RSS's and F-scores, do not scale well to more complex datasets where there are several independent variables. This leads to modelers drawing different conclusions about the same data, and is famously known as the "Rashomon effect." Yet many studies and models in finance are still built this way and make use of information criterion and regularization techniques such as Akaike's information criteria (AIC).

A limitation for more robust frequentist model comparison is the requirement that the models being compared are "nested." That is, one model should be a subset of the other model being compared, e.g.

$$\text{Model 1} \quad \hat{Y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 \tag{2.12}$$

$$\text{Model 2} \quad \hat{Y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_{11} X_1^2. \tag{2.13}$$

Model 1 is nested in Model 2 and we refer to the model selection as a "nested model selection." In contrast to classical model selection, Bayesian model selection need not be restricted to nested models.

7.1 Bayesian Inference

We now consider the more general setting—selection and updating of several candidate models in response to a dataset $\mathbf{y}$. The "model" can be any data model, not just a regression, and the notation used here reflects that. In Bayesian inference, a model is a family of probability distributions, each of which can explain the observed data. More precisely, a model $\mathcal{M}$ is the set of likelihoods $p(\boldsymbol{x}_n \mid \boldsymbol{\theta})$ over all possible parameter values Θ.

For example, consider the case of flipping a coin n times with an unknown bias $\theta \in \Theta \equiv [0, 1]$. The data $\boldsymbol{x}_n = \{x_i\}_{i=1}^n$ is now i.i.d. Bernoulli and if we observe the number of heads $X = x$, the model is the family of binomial distributions

$$\mathcal{M} := \{\mathbb{P}[X = x \mid n, \theta] = \binom{n}{x}\theta^x(1-\theta)^{n-x}\}_{\theta\in\Theta}. \tag{2.14}$$

Each one of these distributions is a potential explanation of the observed head count x. In the Bayesian method, we maintain a belief over which elements in the model are considered plausible by reasoning about $p(\boldsymbol{\theta} \mid \boldsymbol{x}_n)$. See Example 1.1 for further details of this experiment.

We start by re-writing the Bayesian inference formula with explicit inclusion of model indexes. You will see that we have dropped $\mathbf{X}$ since the exact composition of explanatory data is implicitly covered by model index $\mathcal{M}_i$:

$$p(\boldsymbol{\theta}_i \mid \boldsymbol{x}_n, \mathcal{M}_i) = \frac{p(\boldsymbol{\theta}_i \mid \mathcal{M}_i)\, p(\boldsymbol{x}_n \mid \boldsymbol{\theta}_i, \mathcal{M}_i)}{p(\boldsymbol{x}_n \mid \mathcal{M}_i)} \quad i = 1, 2. \tag{2.15}$$

This expression shows that differences across models can occur due to differing priors for $\boldsymbol{\theta}$ and/or differences in the likelihood function. The marginal likelihood in the denominator will usually also differ across models.

7.2 Model Selection

So far, we just considered parameter inference when the model has already been selected. The Bayesian setting offers a very flexible framework for the comparison of competing models—this is formally referred to as "model selection." The models do not have to be nested—all that is required is that the competing specifications share the same $\boldsymbol{x}_n$.

Suppose there are two models, denoted $\mathcal{M}_1$ and $\mathcal{M}_2$, each associated with a respective set of parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. We seek the most "probable" model given the observed data $\boldsymbol{x}_n$. We first apply Bayes' rule to derive an expression for the *posterior model probability*

$$p(\mathcal{M}_i \mid \boldsymbol{x}_n) = \frac{p(\mathcal{M}_i)\, p(\boldsymbol{x}_n \mid \mathcal{M}_i)}{\sum_j p\left(\boldsymbol{x}_n \mid \mathcal{M}_j\right) p\left(\mathcal{M}_j\right)} \quad i = 1, 2. \tag{2.16}$$

Here $p(\mathcal{M}_i)$ is a prior distribution over models that we have selected; a common practice is to set this to a uniform distribution over the models. The value $p(\boldsymbol{x}_n \mid \mathcal{M}_i)$ is a marginal likelihood function—a likelihood function over the space of models in which the parameters have been marginalized out:

$$p(\boldsymbol{x}_n \mid \mathcal{M}_i) = \int_{\boldsymbol{\theta}_i \in \boldsymbol{\Theta}_i} p(\boldsymbol{x}_n \mid \boldsymbol{\theta}_i, \mathcal{M}_i) p(\boldsymbol{\theta}_i \mid \mathcal{M}_i) d\boldsymbol{\theta}_i . \tag{2.17}$$

From a sampling perspective, this marginal likelihood can be interpreted as the probability that the model could have generated the observed data, under the chosen prior belief over its parameters. More precisely, the marginal likelihood can be viewed as the probability of generating $\boldsymbol{x}_n$ from a model $\mathcal{M}_i$ whose parameters $\boldsymbol{\theta}_i \in \boldsymbol{\Theta}_i$ are sampled at random from the prior $p(\boldsymbol{\theta}_i \mid \mathcal{M}_i)$. For this reason, it is often referred to here as the *model evidence* and plays an important role in model selection that we will see later.

We can now construct the *posterior odds ratio* for the two models as

$$\frac{p\left(\mathcal{M}_1 \mid \boldsymbol{x}_n\right)}{p\left(\mathcal{M}_2 \mid \boldsymbol{x}_n\right)} = \frac{p\left(\mathcal{M}_1\right) p\left(\boldsymbol{x}_n \mid \mathcal{M}_1\right)}{p\left(\mathcal{M}_2\right) p\left(\boldsymbol{x}_n \mid \mathcal{M}_2\right)}, \tag{2.18}$$

which is simply the prior odds multiplied by the ratio of the evidence for each model.

Under equal model priors (i.e., $p\left(\mathcal{M}_1\right) = p\left(\mathcal{M}_2\right)$) this reduces to the *Bayes' factor* for Model 1 vs. 2, i.e.

$$B_{1,2} = \frac{p\left(\boldsymbol{x}_n \mid \mathcal{M}_1\right)}{p\left(\boldsymbol{x}_n \mid \mathcal{M}_2\right)}, \tag{2.19}$$

which is simply the ratio of marginal likelihoods for the two models. Since Bayes' factors can become quite large, we usually prefer to work with its logged version

$$log B_{1,2} = \log p\left(\boldsymbol{x}_n \mid \mathcal{M}_1\right) - \log p\left(\boldsymbol{x}_n \mid \mathcal{M}_2\right). \tag{2.20}$$

The derivation of BFs and thus model comparison is straightforward if expressions for marginal likelihoods are analytically known or can be easily derived. However, often this can be quite tricky, and we will learn a few techniques to compute marginal likelihoods in this book.

7.3 Model Selection When There Are Many Models

Suppose now that a set of models $\{\mathcal{M}_i\}$ may be used to explain the data $\boldsymbol{x}_n$. $\boldsymbol{\theta}_i$ represents the parameters of model $\mathcal{M}_i$. Which model is "best"?

We answer this question by estimating the posterior distribution over models:

$$p(\mathcal{M}_i \mid \boldsymbol{x}_n) = \frac{\int_{\boldsymbol{\theta}_i \in \boldsymbol{\Theta}_i} p(\boldsymbol{x}_n \mid \boldsymbol{\theta}_i, \mathcal{M}_i) p(\boldsymbol{\theta}_i \mid \mathcal{M}_i) d\boldsymbol{\theta}_i \, p(\mathcal{M}_i)}{\sum_j p(\boldsymbol{x}_n \mid \mathcal{M}_j) p(\mathcal{M}_j)}. \tag{2.21}$$

Table 2.1 Jeffreys' scale is used to assess the comparative strength of evidence in favor of one model over another

|lnB|	relative odds	favoured model's probability	Interpretation
< 1.0	< 3:1	< 0.750	not worth mentioning
< 2.5	< 12:1	0.923	weak
< 5.0	< 150:1	0.993	moderate
> 5.0	> 150:1	> 0.993	strong

As before we can compare any two models via the *posterior odds*, or if we assume equal priors, by the BFs. Model selection is always relative rather than absolute. We must always pick a reference model $\mathcal{M}_2$ and decide whether model $\mathcal{M}_1$ has more strength. We use Jeffreys' scale to assess the strength of evidence as shown in Table 2.1.

Example 2.4 Model Selection

You compare two models for explaining the behavior of a coin. The first model, $\mathcal{M}_1$, assumes that the probability of a head is fixed to 0.5. Notice that this model does not have any parameters. The second model, $\mathcal{M}_2$, assumes the probability of a head is set to an unknown $\theta \in \Theta = (0, 1)$ with a uniform prior on $\theta : p(\theta \mid \mathcal{M}_2) = 1$. For simplicity, we additionally choose a uniform model prior $p(\mathcal{M}_1) = p(\mathcal{M}_2)$.

Suppose we flip the coin $n = 200$ times and observe $X = 115$ heads. Which model should we prefer in light of this data? We compute the model evidence for each model. The model evidence for $\mathcal{M}_1$

$$p(x \mid \mathcal{M}_1) = \binom{n}{x} \frac{1}{2^{200}} \approx 0.005956. \tag{2.22}$$

The model evidence of $\mathcal{M}_2$ requires integrating over θ:

$$p(x \mid \mathcal{M}_2) = \int_0^1 p(x \mid \theta, \mathcal{M}_2) p(\theta \mid \mathcal{M}_2) d\theta \tag{2.23}$$

$$= \int_0^1 \binom{n}{x} \theta^{115} (1-\theta)^{200-115} d\theta \tag{2.24}$$

$$= \frac{1}{201} \approx 0.004975. \tag{2.25}$$

(continued)

Example 2.4 (continued)

Note that we have used the definition of the Beta density function

$$p(\theta \mid \alpha, \beta) = \frac{(\alpha + \beta - 1)!}{(\alpha - 1)!(\beta - 1)!}\theta^{\alpha-1}(1-\theta)^{\beta-1} \tag{2.26}$$

to evaluate the integral in the marginal density function above.

The Bayes' factor in favor of $\mathcal{M}_1$ is 1.2 and thus $|lnB| = 0.182$ and there is no evidence in favor of $\mathcal{M}_1$.

! Frequentist Approach

An interesting aside here is that a frequentist hypothesis test would reject the null hypothesis $\theta = 0.5$ at the $\alpha = 0.05$ level. The probability of generating at least 115 heads under model $\mathcal{M}_1$ is approximately 0.02. The probability of generating at least 115 tails is also 0.02. So a two-sided test would give a p-value of approximately 4%.

! Hyperparameters

We note in passing that the prior distribution in the example above does not involve any parameterization. If the prior is a parameterized distribution, then the parameters of the prior are referred to as *hyperparameters*. The distributions of the hyperparameters are known as *hyperpriors*. "Bayesian hierarchical modeling" is a statistical model written in multiple levels (hierarchical form) that estimates the parameters of the posterior distribution using the Bayesian method.

7.4 Occam's Razor

The model evidence performs a vital role in the prevention of model over-fitting. Models that are too simple are unlikely to generate the dataset. On the other hand, models that are too complex can generate many possible data sets, but they are unlikely to generate any particular dataset at random. Bayesian inference therefore automates the determination of model complexity using the training data $\boldsymbol{x}_n$ alone and does not need special "fixes" (a.k.a regularization and information criteria) to prevent over-fitting. The underlying philosophical principle of selecting the simplest model, if given a choice, is known as "Occam's razor" (Fig. 2.2).

We maintain a belief over which parameters in the model we consider plausible by reasoning with the posterior

$$p(\boldsymbol{\theta}_i \mid \boldsymbol{x}_n, \mathcal{M}_i) = \frac{p(\boldsymbol{x}_n \mid \boldsymbol{\theta}_i, \mathcal{M}_i)\, p(\boldsymbol{\theta}_i \mid \mathcal{M}_i)}{p(\boldsymbol{x}_n \mid \mathcal{M}_i)}, \tag{2.27}$$

and we may choose the parameter value which maximizes the posterior distribution (MAP).

7.5 Model Averaging

Marginal likelihoods can also be used to derive *model weights* in *Bayesian model averaging (BMA)*. Informally, the intuition behind BMA is that we are never fully convinced that a single model is the correct one for our analysis at hand. There are usually several (and often millions of) competing specifications. To explicitly incorporate this notion of "model uncertainty," one can estimate every model separately, compute relative probability weights for each model, and then generate model-averaged posterior distributions for the parameters (and predictions)

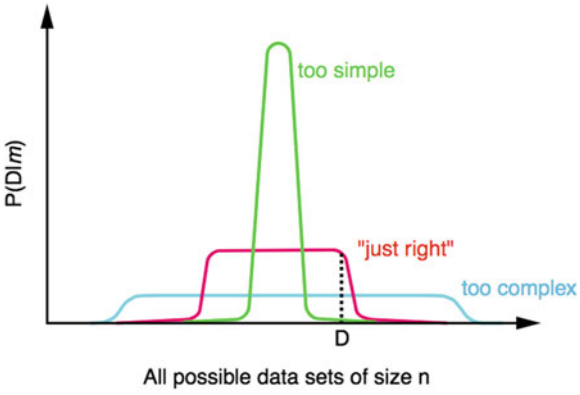


Fig. 2.2 The model evidence $p(D \mid m)$ performs a vital role in the prevention of model over-fitting. Models that are too simple are unlikely to generate the dataset. Models that are too complex can generate many possible data sets, but they are unlikely to generate any particular dataset at random. Source: Rasmussen and Ghahramani (2001)

of interest. *We often choose BMA when there is not strong enough evidence for any particular model*. Prediction of a new point y_* under BMA is given over m models as the weighted average

$$p(y_*|y) = \sum_{i=1}^{m} p(y_*|y, \mathcal{M}_i) p(\mathcal{M}_i|y). \tag{2.28}$$

Note that model-averaged prediction would be cumbersome to accomplish in a classical framework, and thus constitutes another advantage of employing a Bayesian estimation approach.

? Multiple Choice Question 2

Which of the following statements are true:

1. Bayesian inference is ideally suited to model selection because the model evidence effectively penalizes over-parameterized models.
2. The principle of Occam's razor is to simply choose the model with the least bias.
3. Bayesian model averaging uses the uncertainty from the model to weight the output from each model.
4. Bayesian model averaging is a method of consensus voting between models—the best candidate model is selected for each new observation.
5. Hierarchical Bayesian modeling involves nesting Bayesian models through parameterizations of prior distributions and their distributions.

8 Probabilistic Graphical Models

Graphical models (a.k.a. Bayesian networks) are a method for representing relationships between random variables in a probabilistic model. They provide a useful tool for big data, providing graphical representations of complex datasets.

To see how graphical models arise, we can revisit the familiar perceptron model from the previous chapter in a probabilistic framework, i.e. the network weights are now assumed to be probabilistic. As a starting point, consider a logistic regression classifier with probabilistic output:

$$\mathbb{P}[G \mid X] = \sigma(U) = \frac{1}{1 + e^{-U}},\ U = wX + b,\ G \in \{0, 1\},\ X \in \mathbb{R}^p. \tag{2.29}$$

By Bayes' law, we know that the posterior probabilities must be given by the likelihood, prior and evidence:

$$\mathbb{P}[G \mid X] = \frac{\mathbb{P}[X \mid G]\mathbb{P}[G]}{\mathbb{P}[X]} = \frac{1}{1 + e^{-\left(\log\left(\frac{\mathbb{P}[X \mid G])}{\mathbb{P}[X \mid G^c]}\right) + \log\left(\frac{\mathbb{P}[G]}{\mathbb{P}[G^c]}\right)\right)}}, \tag{2.30}$$

where G^c is the complement of G. So the outputs are only posterior probabilities when the weights and biases are, respectively, likelihood and log-odds ratios:

$$w_j = \frac{\mathbb{P}\left[X_j \mid G\right]}{\mathbb{P}\left[X_j \mid G^c\right]},\ \forall j \in \{1, \ldots, p\}, \qquad b = \log\left(\frac{\mathbb{P}[G]}{\mathbb{P}[G^c]}\right). \tag{2.31}$$

In particular, the $X_j's$ must be *conditionally independent* over G; otherwise, the outputs from the logistic regression are not the true posterior probabilities. Put differently, the posteriors of the input given the class can only be found when the input is mutually independent given the class G. In this case, the logistic regression is a naive Bayes' classifier—a type of generative model which models the joint distribution as the products of marginals, $\mathbb{P}[X, G] = \mathbb{P}[G]\prod_{j=1}^{p}\mathbb{P}\left[X_j \mid G\right]$. Hence, under this data assumption, logistic regression is the discriminative counterpart to naive Bayes. Figure 2.3b shows an example of an equivalent logistic regression models and naive Bayes' binary classifier for the case when the inputs are binary.

Furthermore, if the conditional density functions of the inputs, $\mathbb{P}\left[X_j \mid G\right]$, are Gaussian (but not necessarily identical), then we can establish equivalence between logistic regression and a Gaussian naive Bayes' classifier. See Exercise 2.7 for establishing the equivalence when the inputs are binary.

The graphical model captures the causal process (Pearl, 1988) by which the observed data was generated. For this reason, such models are often called generative models.

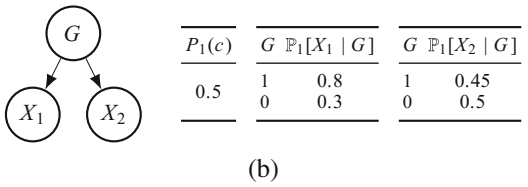


Fig. 2.3 Logistic regression $f_w(X) = \sigma(w^T X)$ and an equivalent naive Bayes' classifier. (**a**) Logistic regression weights and resulting predictions. (**b**) A naive Bayes' classifier with the same probabilistic output

Naive Bayes' classification is the simplest form of a probabilistic graphical model (PGM) with a directed graph $\mathcal{G} = (\mathcal{X}, E)$, where the edges, E, represent the conditional dependence between the random variables $\mathcal{X} = (X, Y)$. For example, Fig. 2.3b shows the dependence of the response G on X in the naive Bayes' classifier. Such graphs, provided they are directed, are often referred to as "Bayesian networks." Such graphical model captures the causal process by which the observed data was generated. If the graph is undirected—an undirected graphical model (UGM), as in restricted Boltzmann machines (RBMs), then the network is referred to as a "Markov network" or Markov random field.

RBMs have the specific restriction that there are no observed–observed and hidden–hidden node connections. RBMs are an example of continuous latent variable models which find the probability of an observed variable by marginalizing over the continuous latent variable, Z,

$$p(x) = \int p(x \mid z)p(z)dz. \tag{2.32}$$

This type of graphical model corresponds to that of factor analysis. Other related types of graphical models include mixture models.

8.1 Mixture Models

A standard mixture probability density of a continuous and independently (but not identically) distributed random variable X, whose value is denoted by x, is defined as

$$p(x; \upsilon) = \sum_{k=1}^{K} \pi_k p(x; \theta_k). \tag{2.33}$$

The mixture density has K components (or states) and is defined by the parameter set $\upsilon = \{\theta, \pi\}$, where $\pi = \{\pi_1, \cdots, \pi_K\}$ is the set of weights given to each component and $\theta = \{\theta_1, \cdots, \theta_K\}$ is the set of parameters describing each component distribution. A well-known mixture model is the Gaussian mixture model (GMM):

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}\left(x; \mu_k, \sigma_k^2\right), \tag{2.34}$$

where each component parameter vector θ_k is the mean and variance parameters, μ_k and σ_k^2.

When X is discrete, the graphical model is referred to as a "discrete mixture model." Examples of GMMs are common in finance. Risk managers, for example, speak in terms of "correlation breakdowns," "market contagion," and "volatility shocks." Finger (1997) presents a two-component GMM for modeling risk under normal and stressed market scenarios which has become standard methodology for stressed Value-at-Risk and Economic Capital modeling in the investment banking sector. Mixture models can also be used to cluster data and have a non-probabilistic analog called the K-means algorithm which is a well-known unsupervised learning method used in finance and other fields.

Before such a model can be fitted, it is necessary to introduce an additional variable which represents the current state of the data, i.e. which of the mixture component distributions is the current observation drawn from.

8.1.1 Hidden Indicator Variable Representation of Mixture Models

Let us first suppose that the independent random variable, X, has been observed over N data points, $\mathbf{x}_N = \{x_1, \cdots, x_N\}$. The set is assumed to be generated by a K-component mixture model.

To indicate the mixture component from which a sample was drawn, we introduce an independent hidden (a.k.a. latent) discrete random variable, $S \in \{1, \ldots, K\}$. For each observation x_i, the value of S is denoted as s_i, and is encoded as a binary vector of length K. We set the vector's k-th component, $(s_i)_k = 1$ to indicate that the k-th mixture component is selected, while all other states are set to 0. As a consequence,

$$1 = \sum_{k=1}^{K} (s_i)_k. \tag{2.35}$$

We can now specify the joint probability distribution of X and S in terms of a marginal density $p(s_i; \boldsymbol{\pi})$ and a conditional density $p(x_i \mid s_i; \theta)$ as

$$p(\mathbf{x}_n, \mathbf{s}_n; \upsilon) = \prod_{i}^{N} p(x_i | s_i; \theta) p(s_i; \boldsymbol{\pi}), \tag{2.36}$$

where the marginal densities $p(s_i; \pi)$ are drawn from a multinomial distribution that is parameterized by the mixing weights $\boldsymbol{\pi} = \{\pi_1, \cdots, \pi_K\}$:

$$p(s_i; \boldsymbol{\pi}) = \prod_{k=1}^{K} \pi_k^{(s_i)_k}, \tag{2.37}$$

or, more simply,

$$\mathbb{P}\left[(s_i)_k = 1\right] = \pi_k. \tag{2.38}$$

Naturally the mixing weights, $\pi_k \in [0, 1]$, must satisfy

$$1 = \sum_{k=1}^{K} \pi_k. \tag{2.39}$$

8.1.2 Maximum Likelihood Estimation

The maximum likelihood method of estimating mixture models is known as the expectation–maximization (EM) algorithm. The goal of the EM is to maximize the likelihood of the data given the model, i.e. maximize

$$\mathcal{L}(\upsilon) = \log \left\{ \sum_{s} p(\mathbf{x}_n, \mathbf{s}_n; \upsilon) \right\} = \sum_{i=1}^{N} \sum_{k=1}^{K} (s_i)_k \log \left\{ \pi_k \, p(x_i; \theta_k) \right\}. \tag{2.40}$$

If the sequence of states $\mathbf{s}_n$ were known, then the estimation of the model parameters π, θ would be straightforward; conditioned on the state variables and the observations, Eq. 2.40 could be maximized with respect to the model parameters. However, the value of the state variable is unknown. This suggests an alternative two-stage iterated optimization algorithm: If we know the *expected* value of S, one could use this expectation in the first step to perform a weighted maximum likelihood estimation of Eq. 2.40 with respect to the model parameters. These estimates will be incorrect since the expectation S is inaccurate. So, in the second step, one could update the expected value of all S pretending the model parameters $\upsilon := (\pi, \theta)$ are known and held fixed at their values from the past iteration. This is precisely the strategy of the expectation–maximization (EM) algorithm—a statistically self-consistent, iterative, algorithm for maximum likelihood estimation. With the context of mixture models, the EM algorithm is outlined as follows:

- **E-step:**
  In this step, the parameters υ are held fixed at the old values, υ^{old}, obtained from the previous iteration (or at their initial settings during the algorithm's initialization). Conditioned on the observations, the E-step then computes the probability density of the state variables $S_i, \forall i$ given the current model parameters and observation data, i.e.

  $$p(s_i \mid x_i, \upsilon^{old}) \propto p(x_i \mid s_i; \theta) p(s_i; \pi^{old}). \tag{2.41}$$

  In particular, we compute

$$\mathbb{P}((s_i)_k = 1 \mid x_i, \upsilon^{old}) = \frac{p(x_i \mid (s_i)_k = 1; \theta_k)\pi_k}{\sum_\ell p(x_i \mid (s_i)_\ell = 1; \theta_\ell)\pi_\ell}. \tag{2.42}$$

The likelihood terms $p(x_i \mid (s_i)_k = 1; \theta_k)$ are evaluated using the observation densities defined for each of the states.

- **M-step**:
  In this step, the hidden state probabilities are considered given and maximization is performed with respect to the parameters:

$$\upsilon^{new} = \arg\max_{\upsilon} \mathcal{L}(\upsilon). \tag{2.43}$$

This results in the following update equations for the parameters in the probability distributions:

$$\mu_k = \frac{1}{N} \frac{\sum_{i=1}^{N} (\gamma_i)_k x_i}{\sum_{i=1}^{N} (\gamma_i)_k} \tag{2.44}$$

$$\sigma_k^2 = \frac{1}{N} \frac{\sum_{i=1}^{N} (\gamma_i)_k \, (x_i - \mu_k)^2}{\sum_{i=1}^{N} (\gamma_i)_k}, \ \forall k \in \{1, \ldots, K\}, \tag{2.45}$$

where $(\gamma_i)_k := \mathbb{E}[(s_i)_k \mid x_i]$ are the *responsibilities*—conditional expectations which measure how strongly a data point, x_i, "belongs" to each component, k, of the mixture model.

The number of components needed to model the data depends on the data and can be determined by a Kolmogorov–Smirnoff test or based on entropy criteria. Heavy tailed data required at least two light tailed components to compensate. More components require larger sample sizes to ensure adequate fitting. In the extreme case there may be insufficient data available to calibrate a given mixture model with a certain degree of accuracy. In summary, while GMMs are flexible they may not be the most appropriate model. If more is known about the data distribution, such as its behavior in the tails, incorporation of this knowledge can only help improve the model.

? Multiple Choice Question 3

Which of the following statements are true:

1. Mixture models assume that the data is multi-modal and drawn from a linear combination of uni-modal distributions.
2. The expectation–maximization (EM) algorithm is a type of iterative unsupervised learning algorithm which alternates between updating the probability density of the state variables, based on model parameters (E-step) and updating the parameters by maximum likelihood estimation (M-step).

3. The EM algorithm automatically determines the modality of the distribution and hence the number of components.
4. A mixture model is only appropriate for use in finance if the modeler specifies which component is the most relevant for each observation.

9 Summary

Probabilistic modeling is an important class of models in financial data, which is often noisy and incomplete. Additionally much of finance rests on being able to make financial decisions under uncertainty, a task perfectly suited to probabilistic modeling. In this chapter we have identified and demonstrated how probabilistic modeling is used for financial modeling. In particular we have:

– Applied Bayesian inference to data using simple probabilistic models;
– Show how linear regression with probabilistic weights can be viewed as a simple probabilistic graphical model; and
– Developed more versatile representations of complex data with probabilistic graphical models such as mixture models and hidden Markov models.

10 Exercises

Exercise 2.1: Applied Bayes' Theorem
An accountant is 95 percent effective in detecting fraudulent accounting when it is, in fact, present. However, the audit also yields a "false positive" result for one percent of the non-fraudulent companies audited. If 0.1 percent of the companies are actually fraudulent, what is the probability that a company is fraudulent given that the audit revealed fraudulent accounting?

Exercise 2.2*: FX and Equity
A currency strategist has estimated that JPY will strengthen against USD with probability 60% if S&P 500 continues to rise. JPY will strengthen against USD with probability 95% if S&P 500 falls or stays flat. We are in an upward trending market at the moment, and we believe that the probability that S&P 500 will rise is 70%. We then learn that JPY has actually strengthened against USD. Taking this new information into account, what is the probability that S&P 500 will rise? Hint: Recall Bayes' rule: $P(A \mid B) = \frac{P(B \mid A)}{P(B)} P(A)$.

Exercise 2.3: Bayesian Inference in Trading**
Suppose there are n conditionally independent, but not identical, Bernoulli trials $G_1, \ldots, G_n$ generated from the map $P(G_i = 1 \mid X = x_i) = g_1(x_i \mid \theta)$ with $\theta \in [0, 1]$. Show that the likelihood of $G \mid X$ is given by

$$p(G \mid X, \theta) = \prod_{i=1}^{n} (g_1(x_i \mid \theta))^{G_i} \cdot (g_0(x_i \mid \theta))^{1-G_i} \tag{2.46}$$

and the log-likelihood of $G \mid X$ is given by

$$ln\ p(G \mid X, \theta) = \sum_{i=1}^{n} G_i ln(g_1(x_i \mid \theta)) + (1 - G_i) ln(g_0(x_i \mid \theta)). \tag{2.47}$$

Using Bayes' rule, write the condition probability density function of θ (the "posterior") given the data (X, G) in terms of the above likelihood function.

From the previous example, suppose that $G = 1$ corresponds to JPY strengthening against the dollar and X are the S&P 500 daily returns and now

$$g_1(x \mid \theta) = \theta \mathbb{1}_{x>0} + (\theta + 35)\mathbb{1}_{x \leq 0}. \tag{2.48}$$

Starting with a neutral view on the parameter θ (i.e., $\theta \in [0, 1]$), learn the distribution of the parameter θ given that JPY strengthens against the dollar for two of the three days and S&P 500 is observed to rise for 3 consecutive days. Hint: You can use the Beta density function with a scaling constant $\Gamma(\alpha, \beta)$

$$p(\theta|\alpha, \beta) = \frac{(\alpha + \beta - 1)!}{(\alpha - 1)!(\beta - 1)!} \theta^{\alpha-1}(1 - \theta)^{\beta-1} = \Gamma(\alpha, \beta)\theta^{\alpha-1}(1 - \theta)^{\beta-1} \tag{2.49}$$

to evaluate the integral in the marginal density function.
If θ represents the currency analyst's opinion of JPY strengthening against the dollar, what is the probability that the model overestimates the analyst's estimate?

Exercise 2.4*: Bayesian Inference in Trading
Suppose that you observe the following daily sequence of directional changes in the JPY/USD exchange rate (U (up), D(down or stays flat)):

```
U, D, U, U, D
```

and the corresponding daily sequence of S&P 500 returns is

```
-0.05, 0.01, -0.01, -0.02, 0.03
```

You propose the following probability model to explain the behavior of JPY against USD given the directional changes in S&P 500 returns: Let G denote a Bernoulli R.V., where $G = 1$ corresponds to JPY strengthening against the dollar and r are the S&P 500 daily returns. All observations of G are conditionally independent (but *not* identical) so that the likelihood is

$$p(G \mid r, \theta) = \prod_{i=1}^{n} p(G = G_i \mid r = r_i, \theta)$$

where

$$p(G_i = 1 \mid r = r_i, \theta) = \begin{cases} \theta_u, & r_i > 0 \\ \theta_d, & r_i \leq 0. \end{cases}$$

Compute the full expression for the likelihood that the data was generated by this model.

Exercise 2.5: Model Comparison
Suppose you observe the following daily sequence of direction changes in the stock market (U (up), D(down)):

```
U, D, U, U, D, D, D, D, U, U, U, U, U, U, U, D, U, D, U, D,
U, D, D, D, D, U, U, D, U, D, U, U, U, D, U, D, D, D, U, U,
D, D, D, U, D, U, D, U, D, D
```

You compare two models for explaining its behavior. The first model, $\mathcal{M}_1$, assumes that the probability of an upward movement is fixed to 0.5 and the data is i.i.d.

The second model, $\mathcal{M}_2$, also assumes the data is i.i.d. but that the probability of an upward movement is set to an unknown $\theta \in \Theta = (0, 1)$ with a uniform prior on $\theta : p(\theta|\mathcal{M}_2) = 1$. For simplicity, we additionally choose a uniform model prior $p(\mathcal{M}_1) = p(\mathcal{M}_2)$.

Compute the model evidence for each model.

Compute the Bayes' factor and indicate which model should we prefer in light of this data?

Exercise 2.6: Bayesian Prediction and Updating
Using Bayesian prediction, predict the probability of an upward movement given the best model and data in Exercise 2.5.

Suppose now that you observe the following new daily sequence of direction changes in the stock market (U (up), D(down)):

```
D, U, D, D, D, D, U, D, U, D, U, D, D, D, U, U, D, U, D, D, D,
U, U, D, D, D, U, D, U, D, U, D, D, D, U, D, U, D, U, D, D, D,
D, U, U, D, U, D, U, U
```

Using the best model from Exercise 2.5, compute the new posterior distribution function based on the new data and the data in the previous question and predict the probability of an upward price movement given all data. State all modeling assumptions clearly.

Exercise 2.7: Logistic Regression Is Naive Bayes
Suppose that G and $X \in \{0, 1\}^p$ are Bernoulli random variables and the X_is are mutually independent given G—that is, $\mathbb{P}[X \mid G] = \prod_{i=1}^{p} \mathbb{P}[X_i \mid G]$. Given a naive Bayes' classifier $\mathbb{P}[G \mid X]$, show that the following logistic regression model produces equivalent output if the weights are

$$w_0 = \log \frac{\mathbb{P}[G]}{\mathbb{P}[G^c]} + \sum_{i=1}^{p} \log \frac{\mathbb{P}[X_i = 0 \in G]}{\mathbb{P}[X_i = 0 \in G^c]}$$

$$w_i = \log \frac{\mathbb{P}[X_i = 1 \in G]}{\mathbb{P}[X_i = 1 \in G^c]} \cdot \frac{\mathbb{P}[X_i = 0 \in G^c]}{\mathbb{P}[X_i = 0 \in G]}, \quad i = 1, \ldots, p.$$

Exercise 2.8: Restricted Boltzmann Machines**
Consider a probabilistic model with two types of binary variables: visible binary stochastic units $\mathbf{v} \in \{0, 1\}^D$ and hidden binary stochastic units $\mathbf{h} \in \{0, 1\}^F$, where D and F are the number of visible and hidden units, respectively. The joint probability density to observe their values is given by the exponential distribution

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp\left(-E(\mathbf{v}, \mathbf{h})\right), \quad Z = \sum_{\mathbf{v},\mathbf{h}} \exp\left(-E(\mathbf{v}, \mathbf{h})\right)$$

and where the energy $E(\mathbf{v}, \mathbf{h})$ of the state $\{\mathbf{v}, \mathbf{h}\}$ is

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{v}^T W \mathbf{h} - \mathbf{b}^T \mathbf{v} - \mathbf{a}^T \mathbf{h} = -\sum_{i=1}^{D} \sum_{j=1}^{F} W_{ij} v_i h_j - \sum_{i=1}^{D} b_i v_i - \sum_{j=1}^{F} a_j h_j,$$

with model parameters $\mathbf{a}$, $\mathbf{b}$, W. This probabilistic model is called the restricted Boltzmann machine. Show that conditional probabilities for visible and hidden nodes are given by the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$:

$$\mathbb{P}[v_i = 1 \mid \mathbf{h}] = \sigma\left(\sum_j W_{ij} h_j + b_i\right), \quad \mathbb{P}[h_i = 1 \mid \mathbf{v}] = \sigma\left(\sum_j W_{ij} v_j + a_i\right).$$

Appendix

Answers to Multiple Choice Questions

Question 1
Answer: 1,3,4.

Question 2
Answer: 1,3,5.

Question 3
Answer: 1,2. Mixture models assume that the data is multi-modal—the data is drawn from a linear combination of uni-modal distributions. The expectation–maximization (EM) algorithm is a type of iterative, self-consistent, unsupervised

learning algorithm which alternates between updating the probability density of the state variables, based on model parameters (E-step) and updating the parameters by maximum likelihood estimation (M-step). The EM algorithm does not automatically determine the modality of the data distribution, although there are statistical tests to determine this. A mixture model assigns a probabilistic weight for every component that each observation might belong to. The component with the highest weight is chosen.

References

Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science 16*(3), 199–231.

Duembgen, M., & Rogers, L. C. G. (2014). Estimate nothing. https://arxiv.org/abs/1401.5666.

Finger, C. (1997). *A methodology to stress correlations*, fourth Quarter. RiskMetrics Monitor.

Rasmussen, C. E., & Ghahramani, Z. (2001). Occam's razor. In *In Advances in Neural Information Processing Systems 13*, (pp. 294–300). MIT Press.

Chapter 3
Bayesian Regression and Gaussian Processes

This chapter introduces Bayesian regression and shows how it extends many of the concepts in the previous chapter. We develop kernel based machine learning methods—specifically Gaussian process regression, an important class of Bayesian machine learning methods—and demonstrate their application to "surrogate" models of derivative prices. This chapter also provides a natural starting point from which to develop intuition for the role and functional form of regularization in a frequentist setting—the subject of subsequent chapters.

1 Introduction

In general, it is difficult to develop intuition about how the distribution of weights in a parametric regression model represents the data. Rather than induce distributions over variables, as we have seen in the previous chapter, we could instead induce distributions over functions. Specifically, we can express those intuitions using a "covariance kernel ."

We start by exploring Bayesian regression in a more general setup that enables us to easily move from a toy regression model to a more complex non-parametric Bayesian regression model, such as Gaussian process regression. By introducing Bayesian regression in more depth, we show how it extends many of the concepts in the previous chapter. We develop kernel based machine learning methods (specifically Gaussian process regression), and demonstrate their application to "surrogate" models of derivative prices.[1]

[1] Surrogate models learn the output of an existing mathematical or statistical model as a function of input data.

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_3

Chapter Objectives

The key learning points of this chapter are:

- Formulate a Bayesian linear regression model;
- Derive the posterior distribution and the predictive distribution;
- Describe the role of the prior as an equivalent form of regularization in maximum likelihood estimation; and
- Formulate and implement Gaussian Processes for kernel based probabilistic modeling, with programming examples involving derivative modeling.

2 Bayesian Inference with Linear Regression

Consider the following linear regression model which is affine in $x \in \mathbb{R}$:

$$y = f(x) = \theta_0 + \theta_1 x,\ \theta_0, \theta_1 \sim \mathcal{N}(0, 1),\ x \in \mathbb{R}, \tag{3.1}$$

and suppose that we observe the value of the function over the inputs $\mathbf{x} := [x_1, \ldots, x_n]$. The random parameter vector $\boldsymbol{\theta} := [\theta_0, \theta_1]$ is unknown. This setup is referred to as "noise-free," since we assume that $\mathbf{y}$ is strictly given by the function $f(x)$ without noise.

The graphical model representation of this model is given in Fig. 3.1 and clearly specifies that the ith model output only depends on x_i. Note that the graphical model also holds in the case when there is noise.

In the noise-free setting, the expectation of the function under known data is

$$\mathbb{E}_{\boldsymbol{\theta}}[f(x_i)|x_i] = \mathbb{E}_{\boldsymbol{\theta}}[\theta_0] + \mathbb{E}_{\boldsymbol{\theta}}[\theta_1]x_i = 0, \forall i, \tag{3.2}$$

where the expectation operator is w.r.t. the prior density of θ

$$\mathbb{E}_{\boldsymbol{\theta}}[\cdot] = \int (\cdot) p(\boldsymbol{\theta}) d\boldsymbol{\theta}. \tag{3.3}$$

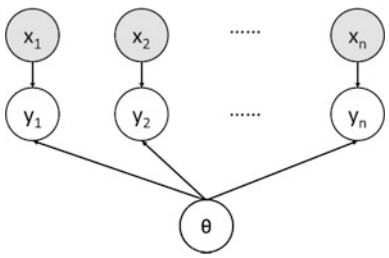


Fig. 3.1 This graphical model represents Bayesian linear regression. The features $\mathbf{x} := \{x_i\}_{i=1}^n$ and responses $\mathbf{y} := \{y_i\}_{i=1}^n$ are known and the random parameter vector $\boldsymbol{\theta}$ is unknown. The ith model output only depends on x_i

Then the covariance of the function values between any two points, x_i and x_j is

$$\mathbb{E}_{\boldsymbol{\theta}}[f(x_i)f(x_j)|x_i, x_j] = \mathbb{E}_{\boldsymbol{\theta}}[\theta_0^2 + \theta_0\theta_1(x_i + x_j) + \theta_1^2 x_i x_j] \tag{3.4}$$

$$= \mathbb{E}_{\boldsymbol{\theta}}[\theta_0^2] + \mathbb{E}_{\boldsymbol{\theta}}[\theta_0^2]x_i x_j + \mathbb{E}_{\boldsymbol{\theta}}[\theta_0\theta_1](x_i + x_j), \tag{3.5}$$

$$= 1 + x_i x_j, \tag{3.6}$$

where the last term is zero because of the independence of θ_0 and θ_1. Then any collection of function values $[f(x_1), \ldots, f(x_n)]$ with given data has a joint Gaussian distribution with covariance matrix $K_{ij} := \mathbb{E}_{\boldsymbol{\theta}}[f(x_i)f(x_j)|x_i, x_j] = 1 + x_i x_j$. Such a probabilistic model is the simplest example of a more general, non-linear, Bayesian kernel learning method referred to as "Gaussian Process Regression" or simply "Gaussian Processes" (GPs) and is the subject of the later material in this chapter.

Noisy Data

The above example is in a noise-free setting where the function values $[f(x_1), \ldots, f(x_n)]$ are observed. In practice, we do not observe these function values, but rather some target values $\mathbf{y} = [y_1, \ldots, y_n]$ which depend on $\mathbf{x}$ by the function, $f(\mathbf{x})$, and some zero-mean Gaussian i.i.d. additive noise with known variance σ_n^2:

$$y_i = f(x_i) + \epsilon_i, \ \epsilon_i \sim \mathcal{N}(0, \sigma_n^2). \tag{3.7}$$

Hence the observed i.i.d. data is $\mathcal{D} := (\mathbf{x}, \mathbf{y})$. Following Rasmussen and Williams (2006), under this noise assumption and the linear model we can write down the likelihood function of the data:

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \prod_{i=1}^{n} p(y_i|x_i, \boldsymbol{\theta})$$

$$= \frac{1}{\sqrt{2\pi}\sigma_n} \exp\{-(y_i - x_i\theta_1 - \theta_0)^2/(2\sigma_n^2)\}$$

and hence $\mathbf{y}|\mathbf{x}, \boldsymbol{\theta} \sim \mathcal{N}(\theta_0 + \theta_1\mathbf{x}, \sigma_n^2 I)$.

Bayesian inference of the parameters in this linear regression model is based on the posterior distribution over the weights:

$$p(\theta_i|\mathbf{y}, \mathbf{x}) = \frac{p(\mathbf{y}|\mathbf{x}, \theta_i)\, p(\theta_i)}{p(\mathbf{y}|\mathbf{x})}, i \in \{0, 1\}, \tag{3.8}$$

where the marginal likelihood in the denominator is given by integrating over the parameters as

$$p(\mathbf{y}|\mathbf{x}) = \int p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) d\boldsymbol{\theta}. \tag{3.9}$$

If we define the matrix X, where $[X]_i := [1, x_i]$, and under more general conjugate priors, we have

$$\boldsymbol{\theta} \sim \mathcal{N}(\mu, \Sigma),$$
$$\mathbf{y}|X, \boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\theta}^T X, \sigma_n^2 I),$$

and the product of Gaussian densities is also Gaussian, we can simply use standard results of moments of affine transformations to give

$$\mathbb{E}[\mathbf{y}|X] = \mathbb{E}[\boldsymbol{\theta}^T X + \epsilon] = \mathbb{E}[\boldsymbol{\theta}^T] X = \mu^T X. \tag{3.10}$$

The conditional covariance is

$$Cov(\mathbf{y}|X) = Cov(\boldsymbol{\theta}^T X) + \sigma_n^2 I = X Cov(\boldsymbol{\theta}) X^T + \sigma_n^2 I = X \Sigma X^T + \sigma_n^2 I. \tag{3.11}$$

To derive the posterior of $\boldsymbol{\theta}$, it is convenient to transform the prior density function from a moment parameterization to a natural parameterization by completing the square. This is useful for multiplying normal density functions such as normalized likelihoods and conjugate priors. The quadratic form for the prior transforms to

$$p(\boldsymbol{\theta}) \propto \exp\{-\frac{1}{2}(\boldsymbol{\theta} - \mu)^T \Sigma^{-1} (\boldsymbol{\theta} - \mu)\}, \tag{3.12}$$

$$\propto \exp\{\mu^T \Sigma^{-1} \boldsymbol{\theta} - \frac{1}{2} \boldsymbol{\theta}^T \Sigma^{-1} \boldsymbol{\theta}\}, \tag{3.13}$$

where the $\frac{1}{2} \mu^T \Sigma^{-1} \mu$ term is absorbed in the normalizing term as it is independent of $\boldsymbol{\theta}$. Using this transformation, the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ is proportional to:

$$p(\mathbf{y}|X, \boldsymbol{\theta}) p(\boldsymbol{\theta}) \propto \exp\{-\frac{1}{2\sigma_n^2}(\mathbf{y} - \boldsymbol{\theta}^T X)^T (\mathbf{y} - \boldsymbol{\theta}^T X)\} \exp\{\mu^T \Sigma^{-1} - \frac{1}{2} \boldsymbol{\theta}^T \Sigma^{-1} \boldsymbol{\theta}\} \tag{3.14}$$

$$\propto \exp\{-\frac{1}{2\sigma_n^2}(-2\mathbf{y}\boldsymbol{\theta}^T X + \boldsymbol{\theta}^T X X^T \boldsymbol{\theta})\} \exp\{\mu^T \Sigma^{-1} - \frac{1}{2} \boldsymbol{\theta}^T \Sigma^{-1} \boldsymbol{\theta}\} \tag{3.15}$$

$$= \exp\{(\Sigma^{-1}\mu + \frac{1}{\sigma_n^2} \mathbf{y}^T X)^T \boldsymbol{\theta}^T - \frac{1}{2} \boldsymbol{\theta}^T (\Sigma^{-1} + \frac{1}{\sigma_n^2} X X^T) \boldsymbol{\theta}\} \tag{3.16}$$

$$= \exp\{a^T \boldsymbol{\theta} - \frac{1}{2} \boldsymbol{\theta}^T A \boldsymbol{\theta}\}. \tag{3.17}$$

The posterior follows the distribution

$$\boldsymbol{\theta} \mid \mathcal{D} \sim \mathcal{N}(\mu', \Sigma'), \tag{3.18}$$

where the moments of the posterior are

$$\mu' = \Sigma' a = (\Sigma^{-1} + \frac{1}{\sigma_n^2} X X^T)^{-1} (\Sigma^{-1} \mu + \frac{1}{\sigma_n^2} \mathbf{y}^T X) \tag{3.19}$$

$$\Sigma' = A^{-1} = (\Sigma^{-1} + \frac{1}{\sigma_n^2} X X^T)^{-1} \tag{3.20}$$

and we use the inverse of transformation above, from natural back to moment parameterization to write

$$p(\boldsymbol{\theta}|\mathcal{D}) \propto \exp\{-\frac{1}{2}(\boldsymbol{\theta} - \mu')^T (\Sigma')^{-1} (\boldsymbol{\theta} - \mu')\}. \tag{3.21}$$

Σ^{-1}, the inverse of a covariance matrix, is referred to as the *precision matrix*. The mean of this distribution is the maximum a posteriori (MAP) estimate of the weights—it is the mode of the posterior distribution. We will show shortly that it corresponds to the penalized maximum likelihood estimate of the weights, with a L_2 (ridge) penalty term given by the log prior.

Figure 3.2 demonstrates Bayesian learning of the posterior distribution of the weights. A bi-variate Gaussian prior is initially chosen for the prior distribution and there are an infinite number of possible lines that could be drawn in the data space $[-1, 1] \times [-1, 1]$. The data is generated under the model $f(x) = 0.3 + 0.5x$ with a small amount of additive i.i.d. Gaussian noise. As the number of points that the likelihood function is evaluated over increases, the posterior distribution sharpens and eventually contracts to a point. See the Bayesian Linear regression Python notebook for details of the implementation.

? Multiple Choice Question 1

Which of the following statements are true:

1. Bayesian regression treats the regression weights as random variables.
2. In Bayesian regression the data function $f(x)$ is assumed to always be observed.
3. The posterior distribution of the parameters is always Gaussian if the prior is Gaussian.
4. The posterior distribution of the regression weights will typically contract with increasing data.
5. The mean of the posterior distribution depends on both the mean and covariance of the prior if it is Gaussian.

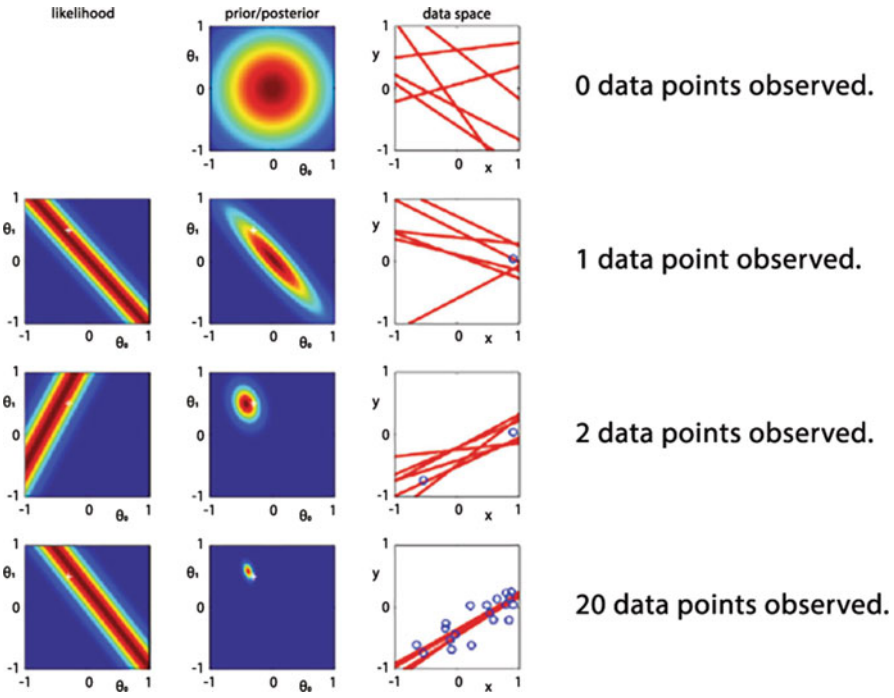


Fig. 3.2 This figure demonstrates Bayesian inference for the linear model. The data has been generated from the function $f(x) = 0.3 + 0.5x$ with a small amount of additive white noise. Source: Bishop (2006)

2.1 Maximum Likelihood Estimation

Let us briefly revisit parameter estimation in a frequentist setting to solidify our understanding of Bayesian inference. Assuming that σ_n^2 is a known parameter, we can easily derive the maximum likelihood estimate of the parameter vector, $\hat{\boldsymbol{\theta}}$. The gradient of the negative log-likelihood function (a.k.a. loss function) w.r.t. $\boldsymbol{\theta}$ is

$$\begin{aligned}\frac{d}{d\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}) &:= -\frac{d}{d\boldsymbol{\theta}}\left(\sum_{i=1}^{n} log\ p(y_i|x_i,\boldsymbol{\theta})\right), \\ &= \frac{1}{2\sigma_n^2}\frac{d}{d\boldsymbol{\theta}}\left(||\mathbf{y}-\boldsymbol{\theta}^T X||_2^2 + c\right) \\ &= \frac{1}{\sigma_n^2}(-\mathbf{y}^T X + \boldsymbol{\theta}^T X^T X),\end{aligned}$$

where the constant $c := -\frac{n}{2}(log(2\pi) + log(\sigma_n^2))$. Setting this gradient to zero gives the orthogonal projection of $\mathbf{y}$ on to the subspace spanned by X:

$$\hat{\boldsymbol{\theta}} = (X^T X)^{-1} X^T \mathbf{y}, \tag{3.22}$$

where $\hat{\boldsymbol{\theta}}$ is the vector in the subspace spanned by X which is closest to $\mathbf{y}$. This result states that the maximum likelihood estimate of an unpenalized loss function (i.e., without including the prior) is the OLS estimate when the noise variance is known. If the noise variance is unknown then the loss function is

$$\mathcal{L}(\boldsymbol{\theta}, \sigma_n^2) = \frac{n}{2} log(\sigma_n^2) + \frac{1}{2\sigma_n^2} ||\mathbf{y} - \boldsymbol{\theta}^T X||_2^2 + c, \tag{3.23}$$

where now $c = \frac{n}{2} log(2\pi)$. Taking the partial derivative

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta}, \sigma_n^2)}{\partial \sigma_n^2} = \frac{n}{2\sigma_n^2} - \frac{1}{2\sigma_n^4} ||\mathbf{y} - \boldsymbol{\theta}^T X||_2^2, \tag{3.24}$$

and setting it to zero gives[2] $\hat{\sigma}_n^2 = \frac{1}{n} ||\mathbf{y} - \boldsymbol{\theta}^T X||_2^2$.

Maximum likelihood estimation is prone to overfitting and therefore should be avoided. We instead maximize the posterior distribution to arrive at the MAP estimate, $\hat{\boldsymbol{\theta}}_{MAP}$. Returning to the above computation under known noise:

$$\begin{aligned}
\frac{d}{d\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) &:= -\frac{d}{d\boldsymbol{\theta}} \left(\sum_{i=1}^{n} log\ p(y_i | x_i, \boldsymbol{\theta}) + log\ p(\boldsymbol{\theta}) \right), \\
&= \frac{d}{d\boldsymbol{\theta}} \left(\frac{1}{2\sigma_n^2} ||\mathbf{y} - \boldsymbol{\theta}^T X||_2^2 + \frac{1}{2} (\boldsymbol{\theta} - \mu)^T \Sigma^{-1} (\boldsymbol{\theta} - \mu) + c \right) \\
&= \frac{1}{\sigma_n^2} (-\mathbf{y}^T X + \boldsymbol{\theta}^T X X^T) + (\boldsymbol{\theta} - \mu)^T \Sigma^{-1}.
\end{aligned}$$

Setting this derivative to zero gives

$$\frac{1}{\sigma_n^2} (\mathbf{y}^T X - \boldsymbol{\theta}^T X X^T) = (\boldsymbol{\theta} - \mu)^T \Sigma^{-1}, \tag{3.25}$$

and after some rearrangement we obtain

$$\hat{\boldsymbol{\theta}}_{MAP} = (X X^T + \sigma_n^2 \Sigma^{-1})^{-1} (\sigma_n^2 \Sigma^{-1} \mu + X^T \mathbf{y}) = A^{-1} (\Sigma^{-1} \mu + \sigma_n^{-2} X^T \mathbf{y}), \tag{3.26}$$

which is equal to the mean of the posterior derived in Eq. 3.19. Of course, this is to be expected since the mean of a Gaussian distribution is also its mode. The difference between $\hat{\boldsymbol{\theta}}_{MAP}$ and $\hat{\boldsymbol{\theta}}$ are the $\sigma_n^2 \Sigma^{-1}$ terms. This term has the effect of

[2] Note that the factor of 2 in the denominator of the second term does not cancel out because the derivative is w.r.t. σ_n^2 and not σ_n.

reducing the condition number of $X^T X$. Forgetting the mean of the prior, the linear system $(X^T X)\boldsymbol{\theta} = X^T \mathbf{y}$ becomes the regularized linear system: $A\boldsymbol{\theta} = \sigma_n^{-2} X^T \mathbf{y}$.

Note that choosing the isotropic Gaussian prior $p(\boldsymbol{\theta}) = \mathcal{N}(0, \frac{1}{2\lambda} I)$ gives the ridge penalty term in the loss function: $\lambda ||\boldsymbol{\theta}||_2^2$, i.e. the negative log Gaussian prior matches the ridge penalty term up to a constant. In the limit, $\lambda \to 0$ recovers maximum likelihood estimation—this corresponds to using the uninformative prior.

Of course, in Bayesian inference, we do not perform point-estimation of the parameters, however it was a useful exercise to confirm that the mean of the posterior in Eq. 3.19 did indeed match the MAP estimate. Furthermore, we have made explicit the interpretation of the prior as a regularization term used in ridge regression.

2.2 Bayesian Prediction

Recall from Chap. 2 that Bayesian prediction requires evaluating the density of $f_* := f(x_*)$ w.r.t. a new data point x_* and the training data $\mathcal{D}$.

In general, we predict the model output at a new point, f_*, by averaging the model output over all possible weights, with the weight density function given by the posterior. That is we seek to find the marginal density $p(f_*|x_*, \mathcal{D}) = \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[p(f_*|x_*, \boldsymbol{\theta})]$, where the dependency on $\boldsymbol{\theta}$ has been integrated out. This conditional density is Gaussian

$$f_*|x_*, \mathcal{D} \sim \mathcal{N}(\mu_*, \Sigma_*), \tag{3.27}$$

with moments

$$\begin{aligned}
\mu_* &= \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[f_*|x_*, \mathcal{D}] = x_*^T \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[\boldsymbol{\theta}|x_*, \mathcal{D}] = x_*^T \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[\boldsymbol{\theta}|\mathcal{D}] = x_*^T \mu' \\
\Sigma_* &= \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[(f_* - \mu_*)(f_* - \mu_*)^T|x_*, \mathcal{D}] \\
&= x_*^T \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[(\boldsymbol{\theta} - \mu')(\boldsymbol{\theta} - \mu')|x_*, \mathcal{D}]x_* \\
&= x_*^T \mathbb{E}_{\boldsymbol{\theta}|\mathcal{D}}[(\boldsymbol{\theta} - \mu')(\boldsymbol{\theta} - \mu')|\mathcal{D}]x_* = x_*^T \Sigma' x_*,
\end{aligned}$$

where we have avoided taking the expectation of the entire density function $p(f_*|x_*, \boldsymbol{\theta})$, but rather just the moments because we know that f_* is Gaussian.

? Multiple Choice Question 2

Which of the following statements are true:

1. Prediction under a Bayesian linear model requires first estimating the posterior distribution of the parameters;
2. The predictive distribution is Gaussian only if the posterior and likelihood distributions are Gaussian;

3. The predictive distribution depends on the weights in the models;
4. The variance of the predictive distribution typically contracts with increasing training data.

2.3 Schur Identity

There is another approach to deriving the predictive distribution from the conditional distribution of the model output which relies on properties of inverse matrices. We can write the joint density between Gaussian random variables X and Y in terms of the *partitioned covariance matrix*:

$$\begin{bmatrix} X \\ Y \end{bmatrix} = \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}\right),$$

where $\Sigma_{xx} = \mathbb{V}(X)$, $\Sigma_{xy} = Cov(XY)$ and $\Sigma_{yy} = \mathbb{V}(Y)$, how can we find the conditional density $p(y|x)$?

In order to express the moments in terms of the partitioned covariance matrix we shall use the following Schur identity:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} M & -MBD^{-1} \\ -D^{-1}CM & D^{-1} + D^{-1}CMBD^{-1} \end{bmatrix}.$$

where the Schur complement w.r.t. the submatrix D is $M := (A - BD^{-1}C)^{-1}$. Applying the Schur identity to the partitioned precision matrix A gives

$$\begin{bmatrix} \Sigma_{yy} & \Sigma_{yx} \\ \Sigma_{xy} & \Sigma_{xx} \end{bmatrix}^{-1} = \begin{bmatrix} A_{yy} & A_{yx} \\ A_{xy} & A_{xx} \end{bmatrix}, \tag{3.28}$$

where

$$A_{yy} = (\Sigma_{yy} - \Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy})^{-1} \tag{3.29}$$

$$A_{yx} = -(\Sigma_{yy} - \Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy})^{-1}\Sigma_{yx}\Sigma_{xx}^{-1}, \tag{3.30}$$

and thus the moments of the Gaussian distribution $p(y|x)$ are

$$\mu_{y|x} = \mu_y + \Sigma_{yx}\Sigma_{xx}^{-1}(x - \mu_x), \tag{3.31}$$

$$\Sigma_{y|x} = \Sigma_{yy} - \Sigma_{yx}\Sigma_{xx}^{-1}\Sigma_{xy}. \tag{3.32}$$

Hence the density of the condition distribution $Y|X$ can alternatively be derived by using the Schur identity. In the special case when the joint density $p(x, y)$ is bi-Gaussian, the expression for the moments simplify to

$$\mu_{y|x} = \mu_y + \frac{\sigma_{yx}}{\sigma_x^2}(x - \mu_x), \tag{3.33}$$

$$\Sigma_{y|x} = \sigma_y - \frac{\sigma_{yx}^2}{\sigma_x^2}, \tag{3.34}$$

where σ_{xy} is the covariance between X and Y.

Now returning to the predictive distribution, the joint density between $\mathbf{y}$ and f_* is

$$\begin{bmatrix} \mathbf{y} \\ f_* \end{bmatrix} = \mathcal{N}\left(\begin{bmatrix} \mu_y \\ \mu_{f_*} \end{bmatrix}, \begin{bmatrix} \Sigma_{yy} & \Sigma_{yf_*} \\ \Sigma_{f_*y} & \Sigma_{f_*f_*} \end{bmatrix} \right). \tag{3.35}$$

We can immediately write down the moments of the condition distribution

$$\mu_{f_*|X,y,x_*} = \mu_{f_*} + \Sigma_{f_*y}\Sigma_{yy}^{-1}(y - \mu_y), \tag{3.36}$$

$$\Sigma_{f_*|X,y,x_*} = \Sigma_{f_*f_*} - \Sigma_{f_*y}\Sigma_{yy}^{-1}\Sigma_{yf_*}. \tag{3.37}$$

Since we know the form of the function $f(x)$, we can simplify this expression by writing that

$$\Sigma_{yy} = K_{X,X} + \sigma_n^2 I, \tag{3.38}$$

where $K_{X,X}$ is the covariance of $f(X)$, which for linear regression takes the form

$$K_{X,X} = \mathbb{E}[\theta_1^2(X - \mu_x)^2],$$

$\Sigma_{f_*f_*} = K_{x_*,x_*}$ and $\Sigma_{yf_*} = K_{X,x_*}$. Now we can write the moments of the predictive distribution as

$$\mu_{f_*|X,y,x_*} = \mu_{f_*} + K_{x_*,X}K_{X,X}^{-1}(y - \mu_y), \tag{3.39}$$

$$K_{f_*|X,y,x_*} = K_{x_*,x_*} - K_{x_*,X}K_{X,X}^{-1}K_{X,x_*}. \tag{3.40}$$

Discussion

Note that we have assumed that the functional form of the map, $f(x)$ is known and parameterized. Here we assumed that the map is linear in the parameters and affine in the features. Hence our approximation of the map is in the data space and, for prediction, we can subsequently forget about the map and work with its moments. The moments of the prior on the weights also no longer need to be specified.

If we do not know the form of the map but want to specify structure on the covariance of the map (i.e., the kernel), then we are said to be approximating in the kernel space rather than in the data space. If the kernels are given by continuous functions of X, then such an approximation corresponds to learning a posterior distribution over an infinite dimensional function space rather than a finite dimensional vector space. Put differently, we perform non-parametric regression rather than parametric regression. This is the remaining topic of this chapter and is precisely how Gaussian process regression models data.

3 Gaussian Process Regression

Whereas, statistical inference involves learning a latent function $Y = f(X)$ of the training data, $(X, Y) := \{(\mathbf{x}_i, \mathbf{y}_i) \mid i = 1, \ldots, n\}$, the idea of GPs is to, without parameterizing[3] $f(X)$, place a probabilistic prior directly on the space of functions (MacKay 1998). Restated, the GP is hence a Bayesian non-parametric model that generalizes the Gaussian distributions from finite dimensional vector spaces to infinite dimensional function spaces. GPs do not provide some parameterized map, $\hat{Y} = f_\theta(X)$, but rather the posterior distribution of the latent function given the training data.

The basic theory of prediction with GPs dates back to at least as far as the time series work of Kolmogorov or Wiener in the 1940s (see (Whittle and Sargent 1983)). GPs are an example of a more general class of supervised machine learning techniques referred to as "kernel learning," which model the covariance matrix from a set of parameterized kernels over the input. GPs extend and put in a Bayesian framework spline or kernel interpolators, and Tikhonov regularization (see (Rasmussen and Williams 2006) and (Alvarez et al. 2012)). On the other hand, (Neal 1996) observed that certain neural networks with one hidden layer converge to a Gaussian process in the limit of an infinite number of hidden units.

We refer to the reader to (Rasmussen and Williams 2006) for an excellent introduction to GPs. In addition to a number of favorable statistical and mathematical properties, such as universality (Micchelli et al. 2006), the implementation support infrastructure is mature—provided by `GpyTorch`, `scikit-learn`, `Edward`, `STAN`, and other open-source machine learning packages.

In this section we restrict ourselves to the simpler case of single-output GPs where f is real-valued. Multi-output GPs are considered in the next section.

[3]This is in contrast to non-linear regressions commonly used in finance, which attempt to parameterize a non-linear function with a set of weights.

3.1 Gaussian Processes in Finance

The adoption of GPs in financial derivative modeling is more recent and sometimes under the name of "kriging" (see, e.g., (Cousin et al. 2016) or (Ludkovski 2018)). Examples of applying GPs to financial time series prediction are presented in (Roberts et al. 2013). These authors helpfully note that AR(p) processes are discrete-time equivalents of GP models with a certain class of covariance functions, known as Matérn covariance functions. Hence, GPs can be viewed as a Bayesian non-parametric generalization of well-known econometrics techniques. da Barrosa et al. (2016) present a GP method for optimizing financial asset portfolios. Other examples of GPs include metamodeling for expected shortfall computations (Liu and Staum 2010), where GPs are used to infer portfolio values in a scenario based on inner-level simulation of nearby scenarios, and Crépey and Dixon (2020), where multiple GPs infer derivative prices in a portfolio for market and credit risk modeling. The approach of Liu and Staum (2010) significantly reduces the required computational effort by avoiding inner-level simulation in every scenario and naturally takes account of the variance that arises from inner-level simulation. The caveat is that the portfolio remains fixed. The approach of Crépey and Dixon (2020), on the other hand, allows for the composition of the portfolio to be changed, which is especially useful for portfolio sensitivity analysis, risk attribution and stress testing.

Derivative Pricing, Greeking, and Hedging

In the general context of derivative pricing, Spiegeleer et al. (2018) noted that many of the calculations required for pricing a wide array of complex instruments, are often similar. The market conditions affecting OTC derivatives may often only slightly vary between observations by a few variables, such as interest rates. Accordingly, for fast derivative pricing, greeking, and hedging, Spiegeleer et al. (2018) propose offline learning the pricing function, through Gaussian Process regression. Specifically, the authors configure the training set over a grid and then use the GP to interpolate at the test points. We emphasize that such GP estimates depend on option pricing models, rather than just market data - somewhat counter the motivation for adopting machine learning, but also the case in other computational finance applications such as Hernandez (2017), Weinan et al. (2017), or Hans Bühler et al. (2018).

Spiegeleer et al. (2018) demonstrate the speed up of GPs relative to Monte-Carlo methods and tolerable accuracy loss applied to pricing and Greek estimation with a Heston model, in addition to approximating the implied volatility surface. The increased expressibility of GPs compared to cubic spline interpolation, a popular numerical approximation techniques useful for fast point estimation, is also demonstrated. However, the applications shown in (Spiegeleer et al. 2018) are limited to single instrument pricing and do not consider risk modeling aspects. In particular, their study is limited to single-output GPs, without consideration

of multi-output GPs (respectively referred to as single- vs. multi-GPs for brevity hereafter).

By contrast, multi-GPs directly model the uncertainty in the prediction of a vector of derivative prices (responses) with spatial covariance matrices specified by kernel functions. Thus the amount of error in a portfolio value prediction, at any point in space and time, can only be adequately modeled using multi-GPs (which, however, do not provide any methodology improvement in estimation of the mean with respect to single-GPs). See Crépey and Dixon (2020) for further details of how multi-GPs can be applied to estimate market and credit risk.

The need for uncertainty quantification in the prediction is certainly a practical motivation for using GPs, as opposed to frequentist machine learning techniques such as neural networks, etc., which only provide point estimates. A high uncertainty in a prediction might result in a GP model estimate being rejected in favor of either retraining the model or even using full derivative model repricing. Another motivation for using GPs, as we will see, is the availability of a scalable training method for the model hyperparameters.

3.2 Gaussian Processes Regression and Prediction

We say that a random function $f : \mathbb{R}^p \mapsto \mathbb{R}$ is drawn from a GP with a mean function μ and a covariance function, called kernel, k, i.e. $f \sim \mathcal{GP}(\mu, k)$, if for any input points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ in $\mathbb{R}^p$, the corresponding vector of function values is Gaussian:

$$[f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_n)] \sim \mathcal{N}(\boldsymbol{\mu}, K_{X,X}),$$

for some mean vector $\boldsymbol{\mu}$, such that $\boldsymbol{\mu}_i = \mu(\mathbf{x}_i)$, and covariance matrix $K_{X,X}$ that satisfies $(K_{X,X})_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. We follow the convention[4] in the literature of assuming $\boldsymbol{\mu} = \mathbf{0}$.

Kernels k can be any symmetric positive semidefinite function, which is the infinite dimensional analogue of the notion of a symmetric positive semidefinite (i.e., covariance) matrix, i.e. such that

$$\sum_{i,j=1}^{n} k(\mathbf{x}_i, \mathbf{x}_j)\xi_i\xi_j \geq 0, \text{ for any points } \mathbf{x}_k \in \mathbb{R}^p \text{ and reals } \xi_k.$$

Radial basis functions (RBF) are kernels that only depend on $||\mathbf{x} - \mathbf{x}'||$, such as the squared exponential (SE) kernel

[4]This choice is not a real limitation in practice (since it is for the prior) and does not prevent the mean of the predictor from being nonzero.

$$k(\mathbf{x}, \mathbf{x}') = \exp\{-\frac{1}{2\ell^2}||\mathbf{x} - \mathbf{x}'||^2\}, \tag{3.41}$$

where the length-scale parameter ℓ can be interpreted as "how far you need to move in input space for the function values to become uncorrelated," or the Matern (MA) kernel

$$k(\mathbf{x}, \mathbf{x}') = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\sqrt{2\nu} \frac{||\mathbf{x} - \mathbf{x}'||}{\ell} \right)^{\nu} K_{\nu} \left(\sqrt{2\nu} \frac{||\mathbf{x} - \mathbf{x}'||}{\ell} \right) \tag{3.42}$$

(which converges to (3.41) in the limit where ν goes to infinity), where Γ is the gamma function, K_{ν} is the modified Bessel function of the second kind, and ℓ and ν are non-negative parameters.

GPs can be seen as distributions over the reproducing kernel Hilbert space (RKHS) of functions which is uniquely defined by the kernel function, k (Scholkopf and Smola 2001). GPs with RBF kernels are known to be universal approximators with prior support to within an arbitrarily small epsilon band of any continuous function (Micchelli et al. 2006).

Assuming additive Gaussian noise, $y \mid \mathbf{x} \sim \mathcal{N}(f(\mathbf{x}), \sigma_n^2)$, and a GP prior on $f(\mathbf{x})$, given training inputs $\mathbf{x} \in X$ and training targets $\mathbf{y} \in Y$, the predictive distribution of the GP evaluated at an arbitrary test point $\mathbf{x}_* \in X_*$ is:

$$\mathbf{f}_* \mid X, Y, \mathbf{x}_* \sim \mathcal{N}(\mathbb{E}[\mathbf{f}_* | X, Y, \mathbf{x}_*], \text{var}[\mathbf{f}_* | X, Y, \mathbf{x}_*]), \tag{3.43}$$

where the moments of the posterior over X_* are

$$\begin{aligned} \mathbb{E}[\mathbf{f}_* | X, Y, X_*] &= \boldsymbol{\mu}_{X_*} + K_{X_*,X}[K_{X,X} + \sigma_n^2 I]^{-1} Y, \\ \text{var}[\mathbf{f}_* | X, Y, X_*] &= K_{X_*,X_*} - K_{X_*,X}[K_{X,X} + \sigma_n^2 I]^{-1} K_{X,X_*}. \end{aligned} \tag{3.44}$$

Here, $K_{X_*,X}$, K_{X,X_*}, $K_{X,X}$, and K_{X_*,X_*} are matrices that consist of the kernel, $k : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$, evaluated at the corresponding points, X and X_*, and $\boldsymbol{\mu}_{X_*}$ is the mean function evaluated on the test inputs X_*.

One key advantage of GPs over interpolation methods is their expressibility. In particular, one can combine the kernels, using convolution, to generalize the base kernels (c.f. "multi-kernel" GPs (Melkumyan and Ramos 2011)).

3.3 Hyperparameter Tuning

GPs are fit to the data by optimizing *the evidence*-the marginal probability of the data given the model with respect to the learned kernel hyperparameters.

The evidence has the form (see, e.g., (Murphy 2012, Section 15.2.4, p. 523)):

$$\log p(Y \mid X, \lambda) = -\left[Y^\top (K_{X,X} + \sigma_n^2 I)^{-1} Y + \log\det(K_{X,X} + \sigma_n^2 I)\right] - \frac{n}{2}\log 2\pi, \tag{3.45}$$

where $K_{X,X}$ implicitly depends on the kernel hyperparameters λ (e.g., $[\ell, \sigma]$, assuming an SE kernel as per (3.41) or an MA kernel for some exogenously fixed value of ν in (3.42)).

The first and second term in the $[\cdots]$ in (3.45) can be interpreted as a *model fit* and a *complexity penalty* term (see (Rasmussen and Williams 2006, Section 5.4.1)). Maximizing the evidence with respect to the kernel hyperparameters, i.e. computing $\lambda^* = \text{argmax}_\lambda \log p(\mathbf{y} \mid \mathbf{x}, \lambda)$, results in an automatic Occam's razor (see (Alvarez et al. 2012, Section 2.3) and (Rasmussen and Ghahramani 2001)), through which we effectively learn the structure of the space of functional relationships between the inputs and the targets. In practice, the negative evidence is minimized by stochastic gradient descent (SGD). The gradient of the evidence is given analytically by

$$\partial_\lambda \log p(\mathbf{y} \mid \mathbf{x}, \lambda) = tr\left((\boldsymbol{\alpha}\boldsymbol{\alpha}^T - (K + \sigma_n^2 I)^{-1})\partial_\lambda (K + \sigma_n^2 I)^{-1}\right), \tag{3.46}$$

where $\boldsymbol{\alpha} := (K + \sigma_n^2 I)^{-1}\mathbf{y}$ and

$$\partial_\ell (K + \sigma_n^2 I)^{-1} = -(K + \sigma_n^2 I)^{-2}\partial_\ell K, \tag{3.47}$$

$$\partial_\sigma (K + \sigma_n^2 I)^{-1} = -2\sigma (K + \sigma_n^2 I)^{-2}, \tag{3.48}$$

with

$$\partial_\ell k(\mathbf{x}, \mathbf{x}') = \ell^{-3}||\mathbf{x} - \mathbf{x}'||^2 k(\mathbf{x}, \mathbf{x}'). \tag{3.49}$$

? Multiple Choice Question 3

Which of the following statements are true:

1. Gaussian Processes are a Bayesian modeling approach which assumes that the data is Gaussian distributed.
2. Gaussian Processes place a probabilistic prior directly on the space of functions.
3. Gaussian Processes model the posterior of the predictor using a parameterized kernel representation of the covariance matrix.
4. Gaussian Processes can be fitted to data by maximizing the evidence for the kernel parameters.
5. During evidence maximization, different kernels are evaluated, and the optimal kernel is chosen.

3.4 Computational Properties

If uniform grids are used (as opposed to a mesh-free GP as described in Sect. 5.2), we have $n = \prod_{k=1}^{p} n_k$, where n_k are the number of grid points per variable.

However, although each kernel matrix $K_{X,X}$ is $n \times n$, we only store the n-vector α in (3.46), which brings reduced memory requirements.

Training time, required for maximizing (3.45) numerically, scales poorly with the number of observations n. This complexity stems from the need to solve linear systems and compute log determinants involving an $n \times n$ symmetric positive definite covariance matrix K. This task is commonly performed by computing the Cholesky decomposition of K incurring $O(n^3)$ complexity. Prediction, however, is faster and can be performed in $O(n^2)$ with a matrix–vector multiplication for each test point, and hence the primary motivation for using GPs is real-time risk estimation performance.

Online Learning

If the option pricing model is recalibrated intra-day, then the corresponding GP model should be retrained. Online learning techniques permit performing this incrementally (Pillonetto et al. 2010). To enable online learning, the training data should be augmented with the constant model parameters. Each time the parameters are updated, a new observation $(\mathbf{x}', \mathbf{y}')$ is generated from the option model prices under the new parameterization. The posterior at test point $\mathbf{x}_*$ is then updated with the new training point following

$$p(\mathbf{f}_*|X, Y, \mathbf{x}', \mathbf{y}', \mathbf{x}_*) = \frac{p(\mathbf{x}', \mathbf{y}'|\mathbf{f}_*)\, p(\mathbf{f}_*|X, Y, \mathbf{x}_*)}{\int_{\mathbf{f}_*} p(\mathbf{x}', \mathbf{y}'|\mathbf{z})\, p(\mathbf{z}|X, Y, \mathbf{x}_*) d\mathbf{z}}, \tag{3.50}$$

where the previous posterior $p(\mathbf{f}_*|X, Y, \mathbf{x}_*)$ becomes the prior in the update. Hence the GP learns over time as model parameters (which are an input to the GP) are updated through pricing model recalibration.

4 Massively Scalable Gaussian Processes

Massively scalable Gaussian processes (MSGP) are a significant extension of the basic kernel interpolation framework described above. The core idea of the framework, which is detailed in (Gardner et al. 2018), is to improve scalability by combining GPs with "inducing point methods." The basic setup is as follows; Using structured kernel interpolation (SKI), a small set of m inducing points are carefully selected from the original training points. The covariance matrix has a Kronecker and Toeplitz structure, which is exploited by the Fast Fourier Transform (FFT). Finally, output over the original input points is interpolated from the output at the

inducing points. The interpolation complexity scales linearly with dimensionality p of the input data by expressing the kernel interpolation as a product of 1D kernels. Overall, SKI gives $O(pn + pmlogm)$ training complexity and $O(1)$ prediction time per test point.

4.1 Structured Kernel Interpolation (SKI)

Given a set of m inducing points, the $n \times m$ cross-covariance matrix, $K_{X,U}$, between the training inputs, X, and the inducing points, $\mathbf{U}$, can be approximated as $\tilde{K}_{X,U} = W_X K_{U,U}$ using a (potentially sparse) $n \times m$ matrix of interpolation weights, W_X. This allows to approximate $K_{\mathbf{X},\mathbf{Z}}$ for an arbitrary set of inputs $\mathbf{Z}$ as $K_{\mathbf{X},\mathbf{Z}} \approx \tilde{K}_{X,U} W_{\mathbf{Z}}^{\top}$. For any given kernel function, K, and a set of inducing points, $\mathbf{U}$, *structured kernel interpolation* (SKI) procedure (Gardner et al. 2018) gives rise to the following approximate kernel:

$$K_{\text{SKI}}(\mathbf{x}, \mathbf{z}) = W_X K_{U,U} W_z^{\top}, \tag{3.51}$$

which allows to approximate $K_{X,X} \approx W_X K_{U,U} W_X^{\top}$. Gardner et al. (2018) note that standard inducing point approaches, such as subset of regression (SoR) or fully independent training conditional (FITC), can be reinterpreted from the SKI perspective. Importantly, the efficiency of SKI-based MSGP methods comes from, first, a clever choice of a set of inducing points which exploit the algebraic structure of $K_{U,U}$, and second, from using very sparse local interpolation matrices. In practice, local cubic interpolation is used.

4.2 Kernel Approximations

If inducing points, U, form a regularly spaced P-dimensional grid, and we use a stationary product kernel (e.g., the RBF kernel), then $K_{U,U}$ decomposes as a Kronecker product of Toeplitz matrices:

$$K_{U,U} = \mathbf{T}_1 \otimes \mathbf{T}_2 \otimes \cdots \otimes \mathbf{T}_P. \tag{3.52}$$

The Kronecker structure allows one to compute the eigendecomposition of $K_{U,U}$ by separately decomposing $\mathbf{T}_1, \ldots, \mathbf{T}_P$, each of which is much smaller than $K_{U,U}$. Further, a Toeplitz matrix can be approximated by a circulant matrix[5] which eigendecomposes by simply applying a discrete Fourier transform (DFT) to its

[5] Gardner et al. (2018) explored 5 different approximation methods known in the numerical analysis literature.

first column. Therefore, an approximate eigendecomposition of each $\mathbf{T}_1, \ldots, \mathbf{T}_P$ is computed via the FFT in only $O(m \log m)$ time.

4.2.1 Structure Exploiting Inference

To perform inference, we need to solve $(K_{\text{SKI}} + \sigma_n^2 I)^{-1}\mathbf{y}$; kernel learning requires evaluating $\log\det(K_{\text{SKI}} + \sigma_n^2 I)$. The first task can be accomplished by using an iterative scheme—linear conjugate gradients—which depends only on matrix vector multiplications with $(K_{\text{SKI}} + \sigma_n^2 I)$. The second is performed by exploiting the Kronecker and Toeplitz structure of $K_{U,U}$ for computing an approximate eigendecomposition, as described above.

In this chapter, we primarily use the basic interpolation approach for simplicity. However for completeness, Sect. 5.3 shows the scaling of the time taken to train and predict with MSGPs.

5 Example: Pricing and Greeking with Single-GPs

In the following example, the portfolio holds a long position in both a European call and a put option struck on the same underlying, with $K = 100$. We assume that the underlying follows Heston dynamics:

$$\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_t^1, \tag{3.53}$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^2, \tag{3.54}$$

$$d\langle W^1, W^2\rangle_t = \rho dt, \tag{3.55}$$

where the notation and fixed parameter values used for experiments are given in Table 3.1 under $\mu = r_0$. We use a Fourier Cosine method (Fang and Oosterlee 2008) to generate the European Heston option price training and testing data for the GP. We also use this method to compare the GP Greeks, obtained by differentiating the kernel function.

Table 3.1 lists the values of the parameters for the Heston dynamics and terms of the European Call and Put option contract used in our numerical experiments. Table 3.2 shows the values for the Euler time stepper used for simulating Heston dynamics and the credit risk model.

For each pricing time t_i, we simultaneously fit a multi-GP to both gridded call and put prices over stock price S and volatility $\sqrt{V}$, keeping time to maturity fixed. Figure 3.3 shows the gridded call (top) and put (bottom) price surfaces at various time to maturities, together with the GP estimate. Within each column in the figure, the same GP model has been simultaneously fitted to both the call and put price

Table 3.1 This table shows the values of the parameters for the Heston dynamics and terms of the European Call and Put option contracts

Parameter description	Symbol	Value
Mean reversion rate	κ	0.1
Mean reversion level	θ	0.15
Vol. of Vol.	σ	0.1
Risk-free rate	r_0	0.002
Strike	K	100
Maturity	T	2.0
Correlation	ρ	-0.9

Table 3.2 This table shows the values for the Euler time stepper used for market risk factor simulation

Parameter description	Symbol	Value
Number of simulation	M	1000
Number of time steps	n_s	100
Initial stock price	S_0	100
Initial variance	V_0	0.1

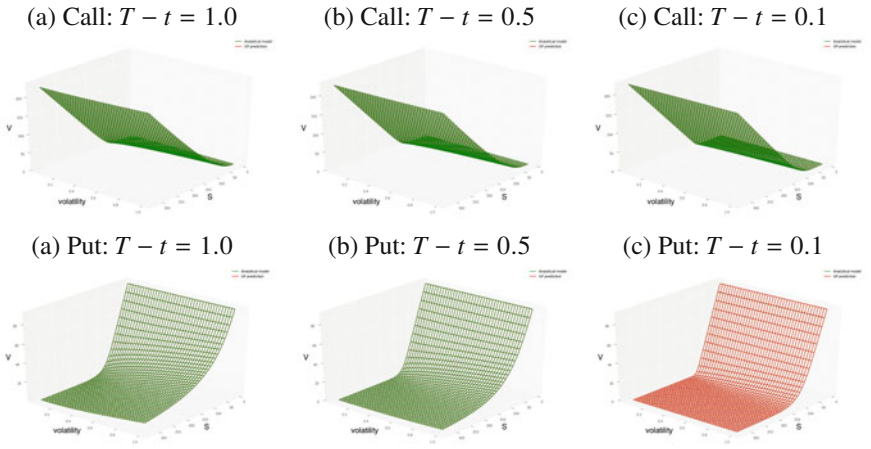


Fig. 3.3 This figure compares the gridded Heston model call (top) and put (bottom) price surfaces at various time to maturities, with the GP estimate. The GP estimate is observed to be practically identical (slightly below in the first five panels and slightly above in the last one). Within each column in the figure, the same GP model has been simultaneously fitted to both the Heston model call and put price surfaces over a 30×30 grid of prices and volatilities, fixing the time to maturity. Across each column, corresponding to different time to maturities, a different GP model has been fitted. The GP is then evaluated out-of-sample over a 40×40 grid, so that many of the test samples are new to the model. This is repeated over various time to maturities

surfaces over a 30×30 grid $\Omega_h \subset \Omega := [0, 1] \times [0, 1]$ of prices and volatilities,[6] fixing the time to maturity. The scaling to the unit domain is not essential. However, we observed superior numerical stability when scaling.

[6] Note that the plot uses the original coordinates and not the re-scaled coordinates.

Across each column, corresponding to different time to maturities, a different GP model has been fitted. The GP is then evaluated out-of-sample over a 40×40 grid $\Omega_{h'} \subset \Omega$, so that many of the test samples are new to the model. This is repeated over various time to maturities.[7]

Extrapolation

One instance where kernel combination is useful in derivative modeling is for extrapolation—the appropriate mixture or combination of kernels can be chosen so that the GP is able to predict outside the domain of the training set. Noting that the payoff is linear when a call or put option is respectively deeply in and out-of-the money, we can configure a GP as a combination of a linear kernel and, say, a SE kernel. The linear kernel is included to ensure that prediction outside the domain preserves the linear property, whereas the SE kernel captures non-linearity. Figure 3.4 shows the results of using this combination of kernels to extrapolate the prices of a call struck at 110 and a put struck at 90. The linear property of the payoff function is preserved by the GP prediction and the uncertainty increases as the test point is further from the training set.

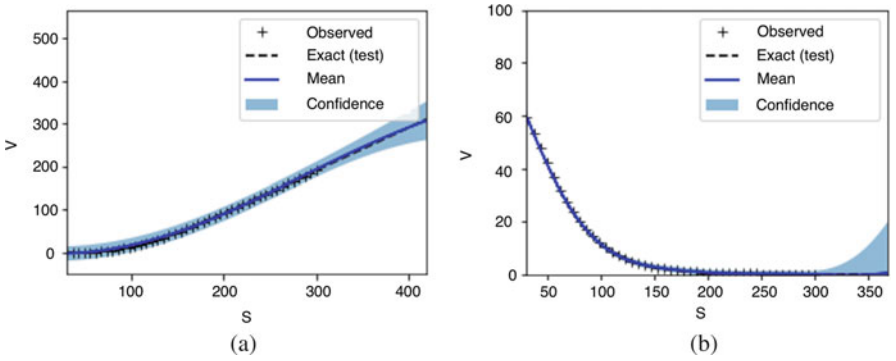


Fig. 3.4 This figure assesses the GP option price prediction in the setup of a Black–Scholes model. The GP with a mixture of a linear and SE kernel is trained on $n = 50$ X, Y pairs, where $X \in \Omega^h \subset (0, 300]$ is the gridded underlying of the option prices and Y is a vector of call or put prices. These training points are shown by the black "+" symbols. The exact result using the Black–Scholes pricing formula is given by the black line. The predicted mean (blue solid line) and variance of the posterior are estimated from Eq. 3.44 over $m = 100$ gridded test points, $X_* \in \Omega^h_* \subset [300, 400]$, for the (left) call option struck at 110 and (center) put option struck at 90. The shaded envelope represents the 95% confidence interval about the mean of the posterior. This confidence interval is observed to increase the further the test point is from the training set. The time to maturity of the options are fixed to two years. (**a**) Call price. (**b**) Put price

[7] Such maturities might correspond to exposure evaluation times in CVA simulation as in Crépey and Dixon (2020). The option model versus GP model are observed to produce very similar values.

5.1 Greeking

The GP provides analytic derivatives with respect to the input variables

$$\partial_{X_*}\mathbb{E}[\mathbf{f}_*|X, Y, X_*] = \partial_{X_*}\boldsymbol{\mu}_{X_*} + \partial_{X_*}K_{X_*,X}\alpha, \tag{3.56}$$

where $\partial_{X_*}K_{X_*,X} = \frac{1}{\ell^2}(X - X_*)K_{X_*,X}$ and we recall from Sect. (3.46) that $\alpha = [K_{X,X} + \sigma_n^2 I]^{-1}\mathbf{y}$ (and in the numerical experiments we set $\boldsymbol{\mu} = 0$). Second-order sensitivities are obtained by differentiating once more with respect to X_*.

Note that α is already calculated at training time (for pricing) by Cholesky matrix factorization of $[K_{X,X} + \sigma_n^2 I]$ with $O(n^3)$ complexity, so there is no significant computational overhead from Greeking. Once the GP has learned the derivative prices, Eq. 3.56 is used to evaluate the first order MtM Greeks with respect to the input variables over the test set. Example source code illustrating the implementation of this calculation is presented in the notebook `Example-2-GP-BS-Derivatives.ipynb`.

Figure 3.5 shows (left) the GP estimate of a call option's delta $\Delta := \frac{\partial C}{\partial S}$ and (right) vega $\nu := \frac{\partial C}{\partial \sigma}$, having trained on the underlying, respectively implied volatility, and on the BS option model prices. For avoidance of doubt, the model is not trained on the BS Greeks. For comparison in the figure, the BS delta and vega are also shown. In each case, the two graphs are practically indistinguishable, with one graph superimposed over the other.

5.2 Mesh-Free GPs

The above numerical examples have trained and tested GPs on uniform grids. This approach suffers from the curse of dimensionality, as the number of training points grows exponentially with the dimensionality of the data. This is why, in order to estimate the MtM cube, we advocate divide-and-conquer, i.e. the use of numerous

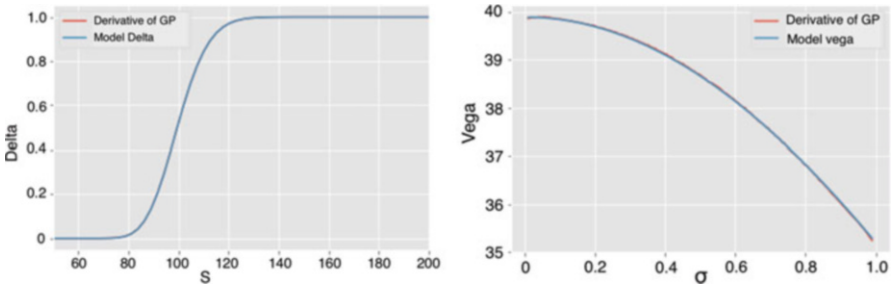


Fig. 3.5 This figure shows (left) the GP estimate of the call option's delta $\Delta := \frac{\partial C}{\partial S}$ and (right) vega $\nu := \frac{\partial C}{\partial \sigma}$, having trained on the underlying, respectively implied volatility, and on the BS option model prices

low input dimensional space, p, GPs run in parallel on specific asset classes. However, use of fixed grids is by no means necessary. We show here how GPs can show favorable approximation properties with a relatively few number of simulated reference points (cf. also (Gramacy and Apley 2015)).

Figure 3.6 shows the predicted Heston call prices using (left) 50 and (right) 100 simulated training points, indicated by "+"s, drawn from a uniform random distribution. The Heston call option is struck at $K = 100$ with a maturity of $T = 2$ years.

Figure 3.7 (left) shows the convergence of the GP MSE of the prediction, based on the number of Heston simulated training points. Fixing the number of simulated points to 100, but increasing the input space dimensionality, p, of each observation point (to include varying Heston parameters, Fig. 3.7 (right) shows the wall-clock time for training a GP with SKI (see Sect. 3.4), Note that the number of SGD iterations has been fixed to 1000.

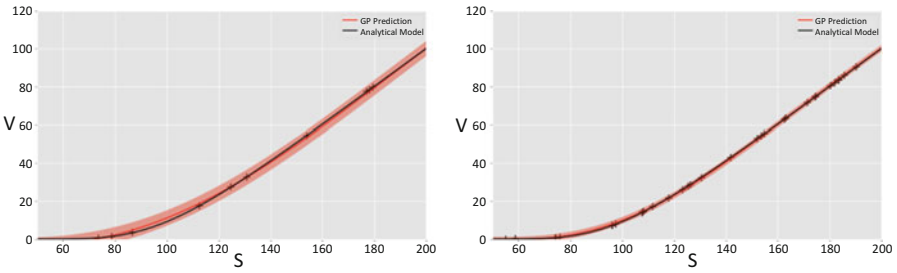


Fig. 3.6 Predicted Heston Call prices using (left) 50 and (right) 100 simulated training points, indicated by "+"s, drawn from a uniform random distribution

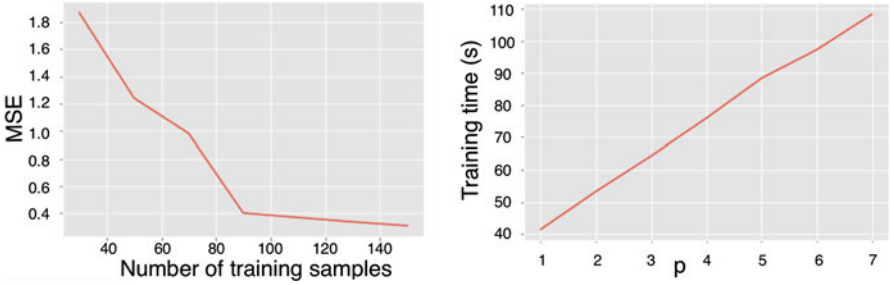


Fig. 3.7 (Left) The convergence of the GP MSE of the prediction is shown based on the number of simulated Heston training points. (Right) Fixing the number of simulated points to 100, but increasing the dimensionality p of each observation point (to include varying Heston parameters), the figure shows the wall-clock time for training a GP with SKI

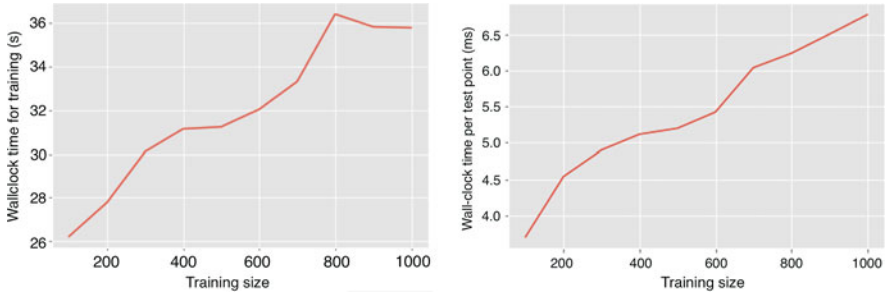


Fig. 3.8 (Left) The elapsed wall-clock time is shown for training against the number of training points generated by a Black–Scholes model. (Right) The elapsed wall-clock time for prediction of a single point is shown against the number of testing points. The reason that the prediction time increases (whereas the theory reviewed in Sect. 3.4 says it should be constant) is due to memory latency in our implementation—each point prediction involves loading a new test point into memory

5.3 Massively Scalable GPs

Figure 3.8 shows the increase of MSGP training time and prediction time against the number of training points n from a Black Scholes model. Fixing the number of inducing points to $m = 30$ (see Sect. 3.4), we increase the number of observations, n, in the $p = 1$ dimensional training set.

Setting the number of SGD iterations to 1000, we observe an approximate 1.4x increase in training time for a 10x increase in the training sample. We observe an approximate 2x increase in prediction time for a 10x increase in the training sample. The reason that the prediction time does not scale independently of n is due to memory latency in our implementation—each point prediction involves loading a new test point into memory. Fast caching approaches can be used to reduce this memory latency, but are beyond the scope of this section.

Note that training and testing times could be improved with CUDA on a GPU, but are not evaluated here.

6 Multi-response Gaussian Processes

A multi-output Gaussian process is a collection of random vectors, any finite number of which have a matrix-variate Gaussian distribution. We borrow from Chen et al. (2017) the following formulation of a separable multi-output kernel specification as per (Alvarez et al. 2012, Eq. (9)):

Definition (MGP) $\mathbf{f}$ is a d variate Gaussian process on $\mathbb{R}^p$ with vector-valued mean function $\boldsymbol{\mu} : \mathbb{R}^p \mapsto \mathbb{R}^d$, kernel $k : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$, and positive semi-definite

parameter covariance matrix $\Omega \in \mathbb{R}^{d\times d}$, if the vectorization of any finite collection of vectors $\mathbf{f}(\mathbf{x}_1), \ldots, \mathbf{f}(\mathbf{x}_n)$ have a joint multivariate Gaussian distribution,

$$\text{vec}([\mathbf{f}(\mathbf{x}_1), \ldots, \mathbf{f}(\mathbf{x}_n)]) \sim \mathcal{N}(\text{vec}(M), \Sigma \otimes \Omega),$$

where $\mathbf{f}(\mathbf{x}_i) \in \mathbb{R}^d$ is a column vector whose components are the functions $\mathbf{f}_l(\mathbf{x}_i)\}_{l=1}^d$, M is a matrix in $\mathbb{R}^{d\times n}$ with $M_{li} = \mu_l(\mathbf{x}_i)$, Σ is a matrix in $\mathbb{R}^{n\times n}$ with $\Sigma_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, and $\Sigma \otimes \Omega$ is the Kronecker product

$$\begin{pmatrix} \Sigma_{11}\Omega & \cdots & \Sigma_{1n}\Omega \\ \vdots & \ddots & \vdots \\ \Sigma_{m1}\Omega & \cdots & \Sigma_{mn}\Omega \end{pmatrix}.$$

Sometimes Σ is called the column covariance matrix while Ω is the row (or task) covariance matrix. We denote $\mathbf{f} \sim \mathcal{MGP}(\mathbf{m}\mu, k, \Omega)$. As explained after Eq. (10) in (Alvarez et al. 2012), the matrices Σ and Ω encode dependencies among the inputs, respectively outputs.

6.1 Multi-Output Gaussian Process Regression and Prediction

Given n pairs of observations $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$, $\mathbf{x}_i \in \mathbb{R}^p$, $\mathbf{y}_i \in \mathbb{R}^d$, we assume the model $\mathbf{y}_i = \mathbf{f}(\mathbf{x}_i)$, $i \in \{1, \ldots, n\}$, where $\mathbf{f} \sim \mathcal{MGP}(\boldsymbol{\mu}, k', \Omega)$ with $k' = k(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij}\sigma_n^2$, in which σ_n^2 is the variance of the additive Gaussian noise. That is, the vectorization of the collection of functions $[\mathbf{f}(\mathbf{x}_1), \ldots, \mathbf{f}(\mathbf{x}_n)]$ follows a multivariate Gaussian distribution

$$\text{vec}([\mathbf{f}(\mathbf{x}_1), \ldots, \mathbf{f}(\mathbf{x}_n)]) \sim \mathcal{N}(\mathbf{0}, K' \otimes \Omega),$$

where K' is the $n \times n$ covariance matrix of which the (i, j)-th element $[K']_{ij} = k'(\mathbf{x}_i, \mathbf{x}_j)$.

To predict a new variable $\mathbf{f}_* = [\mathbf{f}_{*1}, \ldots, \mathbf{f}_{*m}]$ at the test locations $X_* = [\mathbf{x}_{n+1}, \ldots, \mathbf{x}_{n+m}]$, the joint distribution of the training observations $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$ and the predictive targets $\mathbf{f}_*$ are given by

$$\begin{bmatrix} Y \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{MN}\left(\mathbf{0}, \begin{bmatrix} K'(X, X) & K'(X_*, X)^T \\ K'(X_*, X) & K'(X_*, X_*) \end{bmatrix}, \Omega\right), \tag{3.57}$$

where $K'(X, X)$ is an $n \times n$ matrix of which the (i, j)-th element $[K'(X, X)]_{ij} = k'(x_i, x_j)$, $K'(X_*, X)$ is an $m \times n$ matrix of which the (i, j)-th element

$[K'(X_*, X)]_{ij} = k'(x_{n+i}, x_j)$, and $K'(X_*, X_*)$ is an $m \times m$ matrix with the (i, j)-th element $[K'(X_*, X_*)]_{ij} = k'(x_{n+i}, x_{n+j})$. Thus, taking advantage of conditional distribution of multivariate Gaussian process, the predictive distribution is:

$$p(\text{vec}(\mathbf{f}_*)|X, Y, X_*) = \mathcal{N}(\text{vec}(\hat{M}), \hat{\Sigma} \otimes \hat{\Omega}), \tag{3.58}$$

where

$$\hat{M} = K'(X_*, X)^T K'(X, X)^{-1} Y, \tag{3.59}$$

$$\hat{\Sigma} = K'(X_*, X_*) - K'(X_*, X)^{\mathrm{T}} K'(X, X)^{-1} K'(X_*, X), \tag{3.60}$$

$$\hat{\Omega} = \Omega. \tag{3.61}$$

The hyperparameters and elements of the covariance matrix Ω are found by minimizing the negative log marginal likelihood of observations:

$$\mathcal{L}(Y|X, \lambda, \Omega) = \frac{nd}{2} \ln(2\pi) + \frac{d}{2} \ln|K'| + \frac{n}{2} \ln|\Omega| + \frac{1}{2} tr((K')^{-1} Y \Omega^{-1} Y^T). \tag{3.62}$$

Further details of the multi-GP are given in (Bonilla et al. 2007; Alvarez et al. 2012; Chen et al. 2017). The computational remarks made in Sect. 3.4 also apply here, with the additional comment that the training and prediction time also scale linearly (proportionally) with the number of dimensions d. Note that the task covariance matrix Ω is estimated via a d-vector factor $\mathbf{b}$ by $\Omega = \mathbf{b}\mathbf{b}^T + \sigma_\Omega^2 I$ (where the σ_Ω^2 component corresponds to a standard white noise term). An alternative computational approach, which exploits separability of the kernel, is the one described in Section 6.1 of (Alvarez et al. 2012), with complexity $O(d^3 + n^3)$.

7 Summary

In this chapter we have introduced Bayesian regression and shown how it extends many of the concepts in the previous chapter. We develop kernel based machine learning methods, known as Gaussian processes, and demonstrate their application to surrogate models of derivative prices. The key learning points of this chapter are:

- Introduced Bayesian linear regression;
- Derived the posterior distribution and the predictive distribution;
- Described the role of the prior as an equivalent form of regularization in maximum likelihood estimation; and
- Developed Gaussian Processes for kernel based probabilistic modeling, with programming examples in derivative modeling.

8 Exercises

Exercise 3.1: Posterior Distribution of Bayesian Linear Regression
Consider the Bayesian linear regression model

$$y_i = \boldsymbol{\theta}^T X + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma_n^2), \boldsymbol{\theta} \sim \mathcal{N}(\mu, \Sigma).$$

Show that the posterior over data $\mathcal{D}$ is given by the distribution

$$\boldsymbol{\theta}|\mathcal{D} \sim \mathcal{N}(\mu', \Sigma'),$$

with moments:

$$\mu' = \Sigma' a = (\Sigma^{-1} + \frac{1}{\sigma_n^2} X X^T)^{-1}(\Sigma^{-1}\mu + \frac{1}{\sigma_n^2}\mathbf{y}^T X)$$
$$\Sigma' = A^{-1} = (\Sigma^{-1} + \frac{1}{\sigma_n^2} X X^T)^{-1}.$$

Exercise 3.2: Normal Conjugate Distributions
Suppose that the prior is $p(\theta) = \phi(\theta;\ \mu_0, \sigma_0^2)$ and the likelihood is given by

$$p(x_{1:n} \mid \theta) = \prod_{i=1}^{n} \phi(x_i;\ \theta, \sigma^2),$$

where σ^2 is assumed to be known. Show that the posterior is also normal, $p(\theta \mid x_{1:n}) = \phi(\theta;\ \mu_{\text{post}}, \sigma_{\text{post}}^2)$, where

$$\mu_{\text{post}} = \frac{\sigma_0^2}{\frac{\sigma^2}{n} + \sigma_0^2}\bar{x} + \frac{\sigma^2}{\frac{\sigma^2}{n} + \sigma_0^2}\mu_0,$$

$$\sigma_{\text{post}}^2 = \frac{1}{\frac{1}{\sigma_0^2} + \frac{n}{\sigma^2}},$$

where $\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i$.

Exercise 3.3: Prediction with GPs
Show that the predictive distribution for a Gaussian Process, with model output over a test point, $\mathbf{f}_*$, and assumed Gaussian noise with variance σ_n^2, is given by

$$\mathbf{f}_* \mid \mathcal{D}, \mathbf{x}_* \sim \mathcal{N}(\mathbb{E}[\mathbf{f}_*|\mathcal{D}, \mathbf{x}_*], \text{var}[\mathbf{f}_*|\mathcal{D}, \mathbf{x}_*]),$$

where the moments of the posterior over X_* are

$$\mathbb{E}[\mathbf{f}_*|\mathcal{D}, X_*] = \boldsymbol{\mu}_{X_*} + K_{X_*,X}[K_{X,X} + \sigma_n^2 I]^{-1} Y,$$
$$\text{var}[\mathbf{f}_*|\mathcal{D}, X_*] = K_{X_*,X_*} - K_{X_*,X}[K_{X,X} + \sigma_n^2 I]^{-1} K_{X,X_*}.$$

*8.1 Programming Related Questions**

Exercise 3.4: Derivative Modeling with GPs
Using the notebook `Example-1-GP-BS-Pricing.ipynb`, investigate the effectiveness of a Gaussian process with RBF kernels for learning the shape of a European derivative (call) pricing function $V_t = f_t(S_t)$ where S_t is the underlying stock's spot price. The risk free rate is $r = 0.001$, the strike of the call is $K_C = 130$, the volatility of the underlying is $\sigma = 0.1$ and the time to maturity $\tau = 1.0$.

Your answer should plot the variance of the predictive distribution against the stock price, $S_t = s$, over a dataset consisting of $n \in \{10, 50, 100, 200\}$ gridded values of the stock price $s \in \Omega^h := \{i\Delta s \mid i \in \{0, \ldots, n-1\}, \Delta s = 200/(n-1)\} \subseteq [0, 200]$ and the corresponding gridded derivative prices $V(s)$. Each observation of the dataset, $(s_i, v_i = f_t(s_i))$ is a gridded (stock, call price) pair at time t.

Appendix

Answers to Multiple Choice Questions

Question 1
Answer: 1,4,5.

Parametric Bayesian regression always treats the regression weights as random variables.

In Bayesian regression the data function $f(x)$ is only observed if the data is assumed to be noise-free. Otherwise, the function is not directly observed.

The posterior distribution of the parameters will only be Gaussian if both the prior and the likelihood function are Gaussian. The distribution of the likelihood function depends on the assumed error distribution.

The posterior distribution of the regression weights will typically contract with increasing data. The precision matrix grows with decreasing variance and hence the variance of the posterior shrinks with increasing data. There are exceptions if, for example, there are outliers in the data.

The mean of the posterior distribution depends on both the mean and covariance of the prior if it is Gaussian. We can see this from Eq. 3.19.

Question 2
Answer: 1, 2, 4. Prediction under a Bayesian linear model requires first estimating the moments of the posterior distribution of the parameters. This is because the prediction is the expected likelihood of the new data under the posterior distribution.

The predictive distribution is Gaussian only if the posterior and likelihood distributions are Gaussian. The product of Gaussian density functions is also Gaussian.

The predictive distribution does not depend on the weights in the models - it is marginalized out under the expectation w.r.t. the posterior distribution. The variance of the predictive distribution typically contracts with increasing training data because the variance of the posterior and the likelihood typically decreases with increasing training data.

Question 3

Answer: 2, 3, 4.

Gaussian Process regression is a Bayesian modeling approach but they do not assume that the data is Gaussian distributed, neither do they make such an assumption about the error.

Gaussian Processes place a probabilistic prior directly on the space of functions and model the posterior of the predictor using a parameterized kernel representation of the covariance matrix. Gaussian Processes are fitted to data by maximizing the evidence for the kernel parameters. However, it is not necessarily the case that the choice of kernel is effectively a hyperparameter that can be optimized. While this could be achieved in an ad hoc way, there are other considerations which dictate the choice of kernel concerning smoothness and ability to extrapolate.

Python Notebooks

A number of notebooks are provided in the accompanying source code repository, beyond the two described in this chapter. These notebooks demonstrate the use of Multi-GPs and application to CVA modeling (see Crépey and Dixon (2020) for details of these models). Further details of the notebooks are included in the `README.md` file.

References

Alvarez, M., Rosasco, L., & Lawrence, N. (2012). Kernels for vector-valued functions: A review. *Foundations and Trends in Machine Learning, 4*(3), 195–266.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Berlin, Heidelberg: Springer-Verlag.

Bonilla, E. V., Chai, K. M. A., & Williams, C. K. I. (2007). Multi-task Gaussian process prediction. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, USA (pp. 153–160). Curran Associates Inc.

Chen, Z., Wang, B., & Gorban, A. N. (2017, March). Multivariate Gaussian and student$-t$ process regression for multi-output prediction. *ArXiv e-prints*.

Cousin, A., Maatouk, H., & Rullière, D. (2016). Kriging of financial term structures. *European Journal of Operational Research, 255*, 631–648.

Crépey, S., & M. Dixon (2020). Gaussian process regression for derivative portfolio modeling and application to CVA computations. *Computational Finance*.

da Barrosa, M. R., Salles, A. V., & de Oliveira Ribeiro, C. (2016). Portfolio optimization through kriging methods. *Applied Economics, 48*(50), 4894–4905.

Fang, F., & Oosterlee, C. W. (2008). A novel pricing method for European options based on Fourier-cosine series expansions. *SIAM J. SCI. COMPUT*.

Gardner, J., Pleiss, G., Wu, R., Weinberger, K., & Wilson, A. (2018). Product kernel interpolation for scalable Gaussian processes. In *International Conference on Artificial Intelligence and Statistics* (pp. 1407–1416).

Gramacy, R., & D. Apley (2015). Local Gaussian process approximation for large computer experiments. *Journal of Computational and Graphical Statistics, 24*(2), 561–578.

Hans Bühler, H., Gonon, L., Teichmann, J., & Wood, B. (2018). Deep hedging. *Quantitative Finance*. Forthcoming (preprint version available as arXiv:1802.03042).

Hernandez, A. (2017). Model calibration with neural networks. *Risk Magazine* (June 1–5). Preprint version available at SSRN.2812140, code available at https://github.com/Andres-Hernandez/CalibrationNN.

Liu, M., & Staum, J. (2010). Stochastic kriging for efficient nested simulation of expected shortfall. *Journal of Risk, 12*(3), 3–27.

Ludkovski, M. (2018). Kriging metamodels and experimental design for Bermudan option pricing. *Journal of Computational Finance, 22*(1), 37–77.

MacKay, D. J. (1998). Introduction to Gaussian processes. In C. M. Bishop (Ed.), *Neural networks and machine learning*. Springer-Verlag.

Melkumyan, A., & Ramos, F. (2011). Multi-kernel Gaussian processes. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Two*, IJCAI'11 (pp. 1408–1413). AAAI Press.

Micchelli, C. A., Xu, Y., & Zhang, H. (2006, December). Universal kernels. *J. Mach. Learn. Res., 7*, 2651–2667.

Murphy, K. (2012). *Machine learning: a probabilistic perspective*. The MIT Press.

Neal, R. M. (1996). *Bayesian learning for neural networks*, Volume 118 of *Lecture Notes in Statistics*. Springer.

Pillonetto, G., Dinuzzo, F., & Nicolao, G. D. (2010, Feb). Bayesian online multitask learning of Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 32*(2), 193–205.

Rasmussen, C. E., & Ghahramani, Z. (2001). Occam's razor. In *In Advances in Neural Information Processing Systems 13* (pp. 294–300). MIT Press.

Rasmussen, C. E., & Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press.

Roberts, S., Osborne, M., Ebden, M., Reece, S., Gibson, N., & Aigrain, S. (2013). Gaussian processes for time-series modelling. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 371*(1984).

Scholkopf, B., & Smola, A. J. (2001). *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Cambridge, MA, USA: MIT Press.

Spiegeleer, J. D., Madan, D. B., Reyners, S., & Schoutens, W. (2018). Machine learning for quantitative finance: fast derivative pricing, hedging and fitting. *Quantitative Finance, 0*(0), 1–9.

Weinan, E, Han, J., & Jentzen, A. (2017). Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. arXiv:1706.04702.

Whittle, P., & Sargent, T. J. (1983). *Prediction and regulation by linear least-square methods* (NED - New edition ed.). University of Minnesota Press.

Chapter 4
Feedforward Neural Networks

This chapter provides a more in-depth description of supervised learning, deep learning, and neural networks—presenting the foundational mathematical and statistical learning concepts and explaining how they relate to real-world examples in trading, risk management, and investment management. These applications present challenges for forecasting and model design and are presented as a reoccurring theme throughout the book. This chapter moves towards a more engineering style exposition of neural networks, applying concepts in the previous chapters to elucidate various model design choices.

1 Introduction

Artificial neural networks have a long history in financial and economic statistics. Building on the seminal work of (Gallant and White 1988; Andrews 1989; Hornik et al. 1989; Swanson and White 1995; Kuan and White 1994; Lo 1994; Hutchinson, Lo, and Poggio Hutchinson et al.; Baillie and Kapetanios 2007; Racine 2001) develop various studies in the finance, economics, and business literature. Most recently, the literature has been extended to include deep neural networks (Sirignano et al. 2016; Dixon et al. 2016; Feng et al. 2018; Heaton et al. 2017).

In this chapter we shall introduce some of the theory of function approximation and out-of-sample estimation with neural networks when the observation points are independent and typically also identically distributed. Such a case is not suitable for times series data and shall be the subject of later chapters. We shall restrict our attention to feedforward neural networks in order to explore some of the theoretical arguments which help us reason scientifically about architecture design and approximation error. Understanding these networks from a statistical, mathematical, and information-theoretic perspective is key to being able to successfully apply them in practice. While this chapter does present some simple financial examples to

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_4

highlight problematic conceptual issues, we defer the realistic financial applications to later chapters. Also, note that the emphasis of this chapter is how to build statistical models suitable for financial modeling, thus our emphasis is less on engineering considerations and more on how theory can guide the design of useful machine learning methods.

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Develop mathematical reasoning skills to guide the design of neural networks;
- Gain familiarity with the main theory supporting statistical inference with neural networks;
- Relate feedforward neural networks with other types of machine learning methods;
- Perform model selection with ridge and LASSO neural network regression;
- Learn how to train and test a neural network; and
- Gain familiarity with Bayesian neural networks.

Note that section headers ending with * are more mathematically advanced, often requiring some background in analysis and probability theory, and can be skipped by the less mathematically advanced reader.

2 Feedforward Architectures

2.1 Preliminaries

Feedforward neural networks are a form of supervised machine learning that use hierarchical layers of abstraction to represent high-dimensional non-linear predictors. The paradigm that deep learning provides for data analysis is very different from the traditional statistical modeling and testing framework. Traditional fit metrics, such as R^2, t-values, p-values, and the notion of *statistical significance* has been replaced in the machine learning literature by out-of-sample forecasting and understanding the *bias–variance tradeoff* ; that is the tradeoff between a more complex model versus over-fitting. Deep learning is data-driven and focuses on finding structure in large datasets. The main tools for variable or predictor selection are *regularization* and *dropout*.

There are a number of issues in any architecture design. How many layers? How many neurons N_l in each hidden layer? How to perform "variable selection?" Many of these problems can be solved by a stochastic search technique, called dropout

Srivastava et al. (2014), which we discuss in Sect. 5.2.2. Recall from Chap. 1 that a feedforward neural network model takes the general form of a parameterized map

$$Y = F_{W,b}(X) + \boldsymbol{\epsilon}, \tag{4.1}$$

where $F_{W,b}$ is a deep neural network with L layers (Fig. 4.1) and $\boldsymbol{\epsilon}$ is i.i.d. error. The deep neural network takes the form of a composition of simpler functions:

$$\hat{Y}(X) := F_{W,b}(X) = f^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}}(X), \tag{4.2}$$

where $W = (W^{(1)}, \ldots, W^{(L)})$ and $b = (b^{(1)}, \ldots, b^{(L)})$ are weight matrices and bias vectors. Any weight matrix $W^{(\ell)} \in \mathbb{R}^{m \times n}$ can be expressed as n column m-vectors $W^{(\ell)} = [\mathbf{w}^{(\ell)}_{,1}, \ldots, \mathbf{w}^{(\ell)}_{,n}]$. We denote each weight as $w^{(\ell)}_{ij} := \left[W^{(\ell)}\right]_{ij}$.

More formally and under additional restrictions, we can form this parameterized map in the class of compositions of *semi-affine functions*.

Semi-Affine Functions

Let $\sigma : \mathbb{R} \to B \subset \mathbb{R}$ denote a continuous, monotonically increasing function whose codomain is a bounded subset of the real line. A function $f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}$:

(continued)

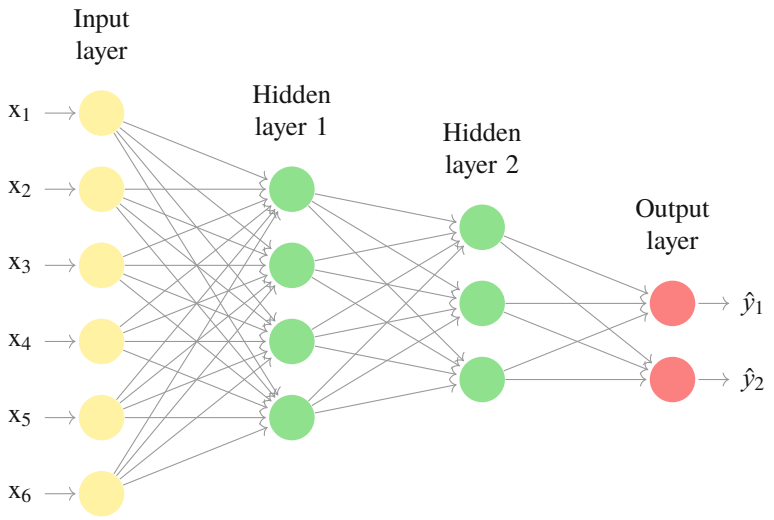


Fig. 4.1 An illustrative example of a feedforward neural network with two hidden layers, six features, and two outputs. Deep learning network classifiers typically have many more layers, use a large number of features and several outputs or classes. The goal of learning is to find the weight on every edge and the bias for every neuron (not illustrated) that minimizes the out-of-sample error

$\mathbb{R}^n \to \mathbb{R}^m$, given by $f(v) = W^{(\ell)}\sigma^{(\ell-1)}(v) + b^{(\ell)}$, $W^{(\ell)} \in \mathbb{R}^{m\times n}$ and $b^{(\ell)} \in \mathbb{R}^m$, is a semi-affine function in v, e.g. $f(v) = wtanh(v) + b$. $\sigma(\cdot)$ are the activation functions of the output from the previous layer.

If all the activation functions are linear, $F_{W,b}$ is just linear regression, regardless of the number of layers L and the hidden layers are redundant. For any such network we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation.[1] For example if there is one hidden layer and $\sigma^{(1)}$ is the identify function, then

$$\hat{Y}(X) = W^{(2)}(W^{(1)}X + b^{(1)}) + b^{(2)} = W^{(2)}W^{(1)}X + W^{(2)}b^{(1)} + b^{(2)} = \tilde{W}X + \tilde{b}. \tag{4.3}$$

Informally, the main effect of activation is to introduce non-linearity into the model, and in particular, interaction terms between the input. The geometric interpretation of the activation units will be discussed in the next section. We can view the special case when the network has one hidden layer and will see that the activation function introduces interaction terms $X_i X_j$. Consider the partial derivative

$$\partial_{X_j}\hat{Y} = \sum_i \mathbf{w}^{(2)}_{,i}\sigma'(I^{(1)}_i)w^{(1)}_{ij}, \tag{4.4}$$

where $\mathbf{w}^{(2)}_{,i}$ is the ith column vector of $W^{(2)}$, $I^{(\ell)}(X) := W^{(\ell)}X + b^{(\ell)}$, and differentiate again with respect to X_k, $k \neq i$ to give

$$\partial^2_{X_j,X_k}\hat{Y} = -2\sum_i \mathbf{w}^{(2)}_{,i}\sigma(I^{(1)}_i)\sigma'(I^{(1)}_i)w^{(1)}_{ij}w^{(1)}_{ik}, \tag{4.5}$$

which is not in general zero unless σ is the identity map.

2.2 Geometric Interpretation of Feedforward Networks

We begin by considering a simple feedforward binary classifier with only two features, as illustrated in Fig. 4.2. The simplest configuration we shall consider has just two inputs and one output unit—this is a multivariate regression model. More precisely, because we shall fit the model to binary responses, this network

[1] Note that there is a potential degeneracy in this case; There may exist "flat directions"—hypersurfaces in the parameter space that have exactly the same loss function.

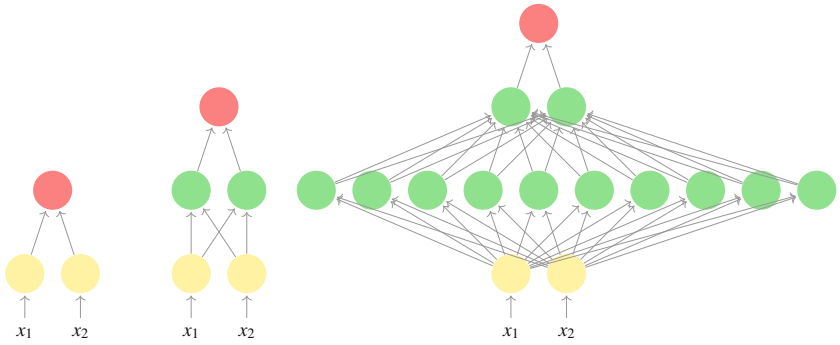


Fig. 4.2 Simple two variable feedforward networks with and without hidden layers. The yellow nodes denote input variables, the green nodes denote hidden units, and the red nodes are outputs. A feedforward network without hidden layers is a linear regressor. A feedforward network with one hidden layer is a *shallow learner* and a feedforward network with two or more hidden layers is a *deep learner*

is a logistic regression. Recall that only one output unit is required to represent the probability of a positive label, i.e. $\mathbb{P}[G = 1 \mid X]$. The next configuration we shall consider has one hidden layer—the number of hidden units shall be equal to the number of input neurons. This choice serves as a useful reference point as many hidden units are often needed for sufficient expressibility. The final configuration has substantially more hidden units. Note that the second layer has been introduced purely to visualize the output from the hidden layer. This set of simple configurations (a.k.a. architectures) is ample to illustrate how a neural network method works.

In Fig. 4.3 the data has been arranged so that no separating linear plane can perfectly separate the points in $[-1, 1] \times [-1, 1]$. The activation function is chosen to be $ReLU(x)$. The weight and biases of the network have been trained on this data. For each network, we can observe how the input space is transformed by the layers by viewing the top row of the figure. We can also view the linear regression in the original, input, space in the bottom row of the figure. The number of units in the first hidden layers is observed to significantly affect the classifier performance.[2]

Determining the weight and bias matrices, together with how many hidden units are needed for generalizable performance is the goal of parameter estimation and model selection. However, we emphasize that some conceptual understanding of neural networks is needed to derive interpretability, the topic of Chap. 5.

Partitioning

The partitioning of the input space is a distinguishing feature of neural networks compared to other machine learning methods. Each hidden unit defines a manifold

[2]There is some redundancy in the construction of the network and around 50 units are needed.

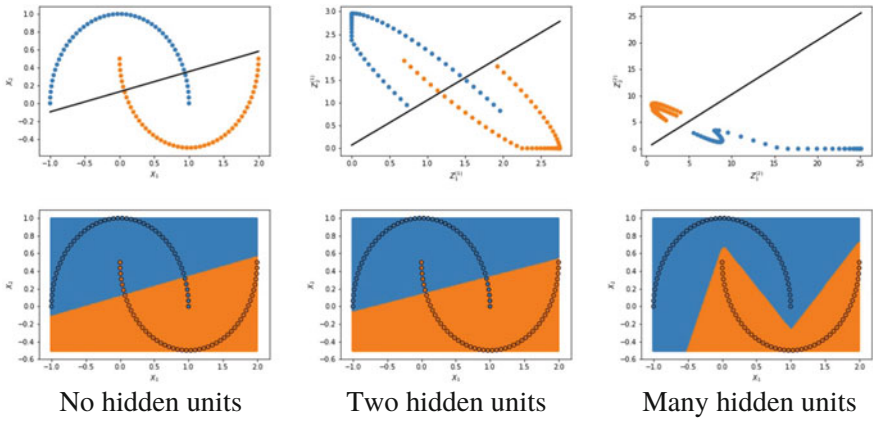


Fig. 4.3 This figure compares various feedforward neural network classifiers applied to a toy, non-linearly separable, binary classification dataset. Its purpose is to illustrate that increasing the number of hidden units in the first hidden layer provides substantial expressibility, even when the number of input variables is small. (Top) Each neural network classifier attempts to separate the labels with a hyperplane in the space of the output from the last hidden layer, $Z^{(L-1)}$. If the network has no hidden layers, then $Z^{(L-1)} = Z^{(0)} = X$. The features are shown in the space of $Z^{(L-1)}$. (Bottom) The separating hyperplane in the space of $Z^{(L-1)}$ is projected to the input space in order to visualize how the layers partition the input space. (Left) A feedforward classifier with no hidden layers is a logistic regression model—it partitions the input space with a plane. (Center) One hidden layer transforms the features by rotation, dilatation, and truncation. (Right) Two hidden layers with many hidden units perform an affine projection into high-dimensional space where points are more separable. See the Deep Classifiers notebook for an implementation of the classifiers and additional diagnostic tests (not shown here)

which divides the input space into convex regions. In other words, each unit in the hidden layer implements a half-space predictor. In the case of a ReLU activation function $f(x) = max(x, 0)$, each manifold is simply a hyperplane and the neuron gets activated when the observation is on the "best" side of this hyperplane, the activation amount is equal to how far from the boundary the given point is. The set of hyperplanes defines a *hyperplane arrangement* (Montúfar et al. 2014). In general, an arrangement of $n \geq p$ hyperplanes in $\mathbb{R}^p$ has at most $\sum_{j=0}^{p} \binom{n}{j}$ convex regions.

For example, in a two-dimensional input space, three neurons with ReLU activation functions will divide the space into no more than $\sum_{j=0}^{2} \binom{3}{j} = 7$ regions, as shown in Fig. 4.4.

Multiple Hidden Layers

We can easily extend this geometrical interpretation to three-layered perceptrons ($L = 3$). Clearly, the neurons in the first (hidden) layer partition the network input space by corresponding hyperplanes into various half-spaces. Hence, the number of these half-spaces equals the number of neurons in the first layer. Then, the neurons in the second layer can classify the intersections of some of these half-spaces, i.e.

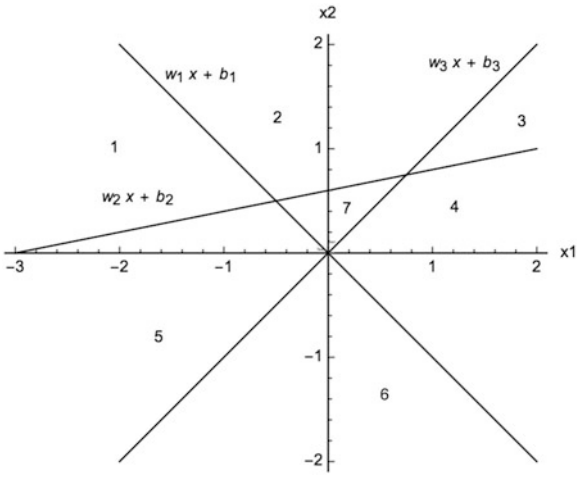


Fig. 4.4 Hyperplanes defined by three neurons in the hidden layer, each with ReLU activation functions, form a hyperplane arrangement. An arrangement of 3 hyperplanes in $\mathbb{R}^2$ has at most $\sum_{j=0}^{2} \binom{3}{j} = 7$ convex regions

they represent convex regions in the input space. This means that a neuron from the second layer is active if and only if the network input corresponds to a point in the input space that is located simultaneously in all half-spaces, which are classified by selected neurons from the first layer.

The maximal number of linear regions of the functions computed by a neural network with p input units and $L-1$ hidden layers, with equal width $n^{(\ell)} = n \geq p$ rectifiers at the ℓth layer, can compute functions that have $\Omega\left(\left[\frac{n}{p}\right]^{(L-2)p} n^p\right)$ linear regions (Montúfar et al. 2014). We see that the number of linear regions of deep models grows exponentially in L and polynomially in n. See Montúfar et al. (2014) for a more detailed exposition of how the additional layers partition the input space.

While this form of reasoning guides our intuition towards designing neural network architectures it falls short at explaining why projection into a higher dimensional space is complementary to how the networks partition the input space. To address this, we turn to some informal probabilistic reasoning to aid our understanding.

2.3 Probabilistic Reasoning

Data Dimensionality

First consider any two independent standard Gaussian random p-vectors $X, Y \sim \mathcal{N}(0, I)$ and define their distance in Euclidean space by the 2-norm

$$d(X, Y)^2 := ||X - Y||_2^2 = \sum_{i=1}^{p} (X_i - Y_i)^2. \tag{4.6}$$

Taking expectations gives

$$\mathbb{E}[d(X, Y)^2] = \sum_{i=1}^{p} \mathbb{E}[X_i^2] + \mathbb{E}[Y_i^2] = 2p. \tag{4.7}$$

Under these i.i.d. assumptions, the mean of the pairwise distance squared between any random points in $\mathbb{R}^p$ is increasingly linear with the dimensionality of the space. By Jensen's inequality for concave functions, such as $\sqrt{x}$

$$\mathbb{E}[d(X, Y)] = \mathbb{E}[\sqrt{d(X, Y)^2}] \leq \sqrt{\mathbb{E}[d(X, Y)^2]} = \sqrt{2p}, \tag{4.8}$$

and hence the expected distance is bounded above by a function which grows to the power of $p^{1/2}$. This simple observation supports the characterization of random points as being less concentrated as the dimensionality of the input space increases. In particular, this property suggests machine learning techniques which rely on concentration of points in the input space, such as linear kernel methods, may not scale well with dimensionality. More importantly, this notion of loss of concentration with dimensionality of the input space does not conflict with how the input space is partitioned—the model defines a convex polytope with a less stringent requirement for locality of data for approximation accuracy.

Size of Hidden Layer

A similar simple probabilistic reasoning can be applied to the output from a one-layer network to understand how concentration varies with the number of units in the hidden layer. Consider, as before two i.i.d. random vectors X and Y in $\mathbb{R}^p$. Suppose now that these vectors are projected by a bounded semi-affine function $g : \mathbb{R}^p \to \mathbb{R}^q$. Assume that the output vectors $g(X), g(Y) \in \mathbb{R}^q$ are i.i.d. with zero mean and variance $\sigma^2 I$. Defining the distance between the output vectors as the $2-$norm

$$d_g^2 := ||g(X) - g(Y)||_2^2 = \sum_{i=1}^{q} (g_i(X) - g_i(Y))^2. \tag{4.9}$$

Under expectations

$$\mathbb{E}[d_g^2] = \sum_{i=1}^{p} \mathbb{E}[g(X)_i^2] + \mathbb{E}[g(Y)_i^2] = 2q\sigma^2 \leq q(\bar{g} - \mathrm{g}) \tag{4.10}$$

and again by Jensen's inequality,

$$\mathbb{E}[d] \leq \sqrt{2}\sqrt{q}\sigma \leq \sqrt{q(\bar{g} - \mathrm{g})}, \tag{4.11}$$

we observe that the distance between the two output vectors, corresponding to the output of a hidden layer g under different inputs X and Y, can be less concentrated as the dimensionality of the output space increases. In other words, points in the codomain of g are on average more separate as q increases.

2.4 Function Approximation with Deep Learning*

While the above informal geometric and probabilistic reasoning provides some intuition for the need for multiple units in the hidden layer of a two-layer MLP, it does not address why deep networks are needed. The most fundamental mathematical concept in neural networks is the *universal representation* theorem. Simply put, this is a statement about the ability of a neural network to approximate any continuous, and unknown, function between input and output pairs with a simple, and known, functional representation. Hornik et al. (1989) show that a feedforward network with a single hidden layer can approximate any continuous function, regardless of the choice of activation function or data.

Formally, let $C^p := \{F : \mathbb{R}^p \to \mathbb{R} \mid F(x) \in C(\mathbb{R})\}$ be the set of continuous functions from $\mathbb{R}^p$ to $\mathbb{R}$. Denote $\Sigma^p(g)$ as the class of functions $\{F : \mathbb{R}^p \to \mathbb{R} : F(x) = W^{(2)}\sigma(W^{(1)}x + b^{(1)}) + b^{(2)}\}$. Consider $\Omega = (0, 1]$ and let C_0 be the collection of all open intervals in $(0, 1]$. Then $\sigma(C_0)$, the σ-algebra generated by C_0, is called the Borel σ-algebra. It is denoted by $\mathcal{B}((0, 1])$. An element of $\mathcal{B}((0, 1])$ is called a Borel set. A map $f : X \to Y$ between two topological spaces X, Y is called Borel measurable if $f^{-1}(A)$ is a Borel set for any open set A.

Let $M^p := \{F : \mathbb{R}^p \to \mathbb{R} \mid F(x) \in \mathcal{B}(\mathbb{R})\}$ be the set of all Borel measurable functions from $\mathbb{R}^p$ to $\mathbb{R}$. We denote the Borel σ-algebra of $\mathbb{R}^p$ as $\mathcal{B}^p$.

Universal Representation Theorem

(Hornik et al. (1989)) *For every monotonically increasing activation function σ, every input dimension size p, and every probability measure μ on $(\mathbb{R}^p, \mathcal{B}^p)$, $\Sigma^p(g)$ is uniformly dense on compacta in C^p and ρ_μ-dense in M^p.*

This theorem shows that standard feedforward networks with only a single hidden layer can approximate any continuous function uniformly on any compact set and any measurable function arbitrarily well in the ρ_μ metric, regardless of the activation function (provided it is measurable), regardless of the dimension of the input space, p, and regardless of the input space. In other words, by taking the number of hidden units, k, *large enough*, every continuous function over $\mathbb{R}^p$ can be approximated arbitrarily closely, uniformly over any bounded set by functions realized by neural networks with one hidden layer.

The universal approximation theorem is important because it characterizes feedforward networks with a single hidden layer as a class of approximate solutions. However, the theorem is not constructive—it does not specify how to configure a multilayer perceptron with the required approximation properties.

The theorem has some important limitations. It says nothing about the effect of adding more layers, other than to suggest they are redundant. It assumes that the optimal network weight vectors are reachable by gradient descent from the initial weight values, but this may not be possible in finite computations. Hence there are additional limitations introduced by the learning algorithm which are not apparent from a functional approximation perspective. The theorem cannot characterize the prediction error in any way, the result is purely based on approximation theory. An important concern is over-fitting and performance generalization on out-of-sample datasets, both of which it does not address. Moreover, it does not inform how MLPs can recover other approximation techniques, as a special case, such as polynomial spline interpolation. As such we shall turn to alternative theory in this section to assess the learnability of a neural network and to further understand it, beginning with a perceptron binary classifier. The reason why multiple hidden layers are needed is still an open problem, but various clues are provided in the next section and later in Sect. 2.7.

2.5 VC Dimension

In addition to expressive power, which determines the approximation error of the model, there is the notion of learnability, which determines the level of estimation error. The former measures the error introduced by an approximating function and the latter error measures the performance lost as a result of using a finite training sample.

One classical measure of the learnability of neural network classifiers is the Vapnik–Chervonenkis (VC) dimension. The VC dimension of a binary model $g = F_{W,b}(X)$ is the maximum number of points that can be arranged so that $F_{W,b}(X)$ *shatters* them, i.e. for all possible assignments of labels to those points, there exists a W, b such that $F_{W,b}$ makes no errors when classifying that set of data points. In the simplest case, a perceptron with n inputs units and a linear threshold activation $\sigma(x) := sgn(x)$ has a VC dimension of $n+1$. For example, if $n = 1$, then

Fig. 4.5 For the points $\{-0.5, 0.5\}$, there are weights and biases that activate only one of them ($W = 1, b = 0$ or $W = -1, b = 0$), none of them ($W = 1, b = -0.75$), and both of them ($W = 1, b = 0.75$)

only two distinct points can always be correctly classified under all possible binary label assignments.

As shown in Fig. 4.5, for the points $\{-0.5, 0.5\}$, there are weights and biases that *activate* both of them ($W = 1, b = 0.75$), only one of them ($W = 1, b = 0$ or $W = -1, b = 0$), and none of them ($W = 1, b = -0.75$). Every distinct pair of points is separable with the linear threshold perceptron. So every dataset of size 2 is shattered by the perceptron. However, this linear threshold perceptron is incapable of shattering triplets, for example, $X \in \{-0.5, 0, 0.5\}$ and $Y \in \{0, 1, 0\}$. In general, the VC dimension of the class of half-spaces in $\mathbb{R}^k$ is $k + 1$. For example, a 2d plane shatters any three points, but cannot shatter four points.

The VC dimension determines both the necessary and sufficient conditions for the consistency and rate of convergence of learning processes (i.e., the process of choosing an appropriate function from a given set of functions). If a class of functions has a finite VC dimension, then it is *learnable*. This measure of capacity is more robust than arbitrary measures such as the number of parameters. It is possible, for example, to find a simple set of functions that depends on only one parameter and that has infinite VC dimension.

? VC Dimension of an Indicator Function

Determine the VC dimension of the indicator function over $\Omega = [0, 1]$

$$\mathcal{F}(x) = \{f : \Omega \to \{0, 1\},\ f(x) = \mathbb{1}_{x\in[t_1,t_2)},\ \text{or}\ f(x) = 1-\mathbb{1}_{x\in[t_1,t_2)}, t_1 < t_2 \in \Omega\}. \tag{4.12}$$

Suppose there are three points x_1, x_2, and x_3 and assume $x_1 < x_2 < x_3$ without loss of generality. All possible labeling of the points is reachable; therefore, we assert that $VC(\mathcal{F}) \geq 3$. With four points x_1, x_2, x_3, and x_4 (assumed increasing as always), you cannot label x_1 and x_3 with the value 1 and x_2 and x_4 with the value 0, for example. Hence $VC(\mathcal{F}) = 3$.

Recently (Bartlett et al. 2017a) prove upper and lower bounds on the VC dimension of deep feedforward neural network classifiers with the piecewise linear activation function, such as ReLU activation functions. These bounds are tight for almost the entire range of parameters. Letting $|W|$ be the number of weights and L be the number of layers, they proved that the VC dimension is $O(|W|Llog(|W|))$.

They further showed the effect of network depth on VC dimension with different non-linearities: there is *no dependence* for piecewise constant, *linear dependence* for piecewise-linear, and *no more than quadratic dependence* for general piecewise-polynomials.

Vapnik (1998) formulated a method of VC dimension based inductive inference. This approach, known as *structural empirical risk minimization*, achieved the smallest bound on the test error by using the training errors and choosing the machine (i.e., the set or functions) with the smallest VC dimension. The minimization problem expresses the bias–variance tradeoff . On the one hand, to minimize the bias, one needs to choose a function from a wide set of functions, not necessarily with a low VC dimension. On the other hand, the difference between the training error and the test error (i.e., variance) increases with VC dimension (a.k.a. expressibility).

The *expected risk* is an out-of-sample measure of performance of the learned model and is based on the joint probability density function (pdf) $p(x, y)$:

$$R[\hat{F}] = \mathbb{E}[\mathcal{L}(\hat{F}(X), Y)] = \int \mathcal{L}(\hat{F}(\mathbf{x}), y) dp(x, y). \tag{4.13}$$

If one could choose $\hat{F}$ to minimize the expected risk, then one would have a definite measure of optimal learning. Unfortunately, the expected risk cannot be measured directly since this underlying pdf is unknown. Instead, we typically use the risk over the training set of N observations, also known as the *empirical risk measure* (ERM):

$$R_{emp}(\hat{F}) := \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\hat{F}(\mathbf{x}_i), \mathbf{y}_i). \tag{4.14}$$

Under i.i.d. data assumptions, the law of large numbers ensures that the empirical risk will asymptotically converge to the expected risk. However, for small samples, one cannot guarantee that ERM will also minimize the expected risk. A famous result from statistical learning theory (Vapnik 1998) is that the VC dimension provides bounds on the expected risk as a function of the ERM and the number of training observations N, which holds with probability $(1 - \eta)$:

$$R[\hat{F}] \leq R_{emp}(\hat{F}) + \sqrt{\frac{h\left(ln\left(\frac{2N}{h}\right) + 1\right) - ln\left(\frac{\eta}{4}\right)}{N}}, \tag{4.15}$$

where h is the VC dimension of $\hat{F}(X)$ and $N > h$. Figure 4.6 shows the tradeoff between VC dimension and the tightness of the bound. As the ratio N/h gets larger, i.e. for a fixed N, we decrease h, the VC confidence becomes smaller, and the actual risk becomes closer to the empirical risk. On the other hand, choosing a model with a higher VC dimension reduces the ERM at the expense of increasing the VC confidence.

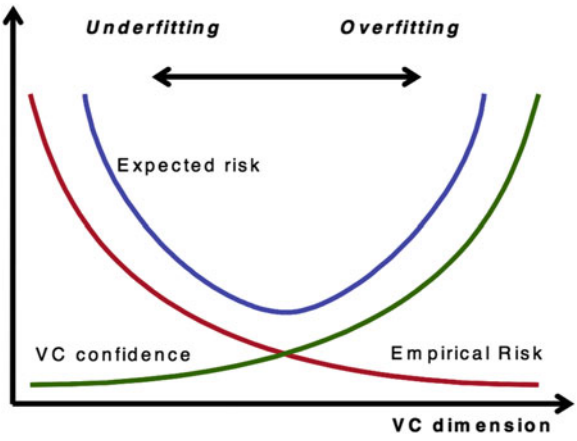


Fig. 4.6 This figure shows the tradeoff between VC dimension and the tightness of the bound. As the ratio N/h gets larger, i.e. for a fixed N, we decrease h, the VC confidence becomes smaller, and the actual risk becomes closer to the empirical risk. On the other hand, choosing a model with a higher VC dimension reduces the ERM at the expense of increasing the VC confidence

The VC dimension plays a more dominant role in small-scale learning problems, where i.i.d. training data is limited and optimization error, that is the error introduced by the optimizer, is negligible. Beyond a certain sample size, computing power and the optimization algorithm become more dominant and the VC dimension is limited as a measure of learnability. Several studies demonstrate that VC dimension based error bounds are too weak and its usage, while providing some intuitive notion of model complexity, have faded in favor of alternative theories. Perhaps most importantly for finance, the bound in Eq. 4.15 only holds for i.i.d. data and little is known in the case when the data is auto-correlated.

? Multiple Choice Question 1

Which of the following statements are true:

1. The hidden units of a shallow feedforward network partition, with n hidden units, partition the input space in $\mathbb{R}^p$ into no more than $\sum_{j=0}^{p} \binom{n}{j}$ convex regions.
2. The VC dimension of a Heaviside activated shallow feedforward network, with one hidden unit, and p features, is $p + 1$.
3. The bias–variance tradeoff is equivalently expressed through the VC confidence and the empirical risk measure.
4. The upper bound on the out-of-sample error of a feedforward network depends on its VC dimension and the number of training samples.
5. The VC dimension always grows linearly with the number of layers in a deep network.

2.6 When Is a Neural Network a Spline?*

Under certain choices of the activation function, we can construct MLPs which are a certain type of piecewise polynomial interpolants referred to as "splines." Let $f(x)$ be any function whose domain is Ω and the function values $f_k := f(x_k)$ are known only at grid points $\Omega^h := \{x_k \mid x_k = kh, k \in \{1, \ldots, K\}\} \subset \Omega \subset \mathbb{R}$ which are spaced by h. Note that the requirement that the data is gridded is for ease of exposition and is not necessary. We construct an orthogonal basis over Ω to give the interpolant

$$\hat{f}(x) = \sum_{i=1}^{K} \phi_k(x) f_k, \ x \in \Omega, \tag{4.16}$$

where the $\{\phi_k\}_{k=1}^K$ are orthogonal basis functions. Under additional restrictions of the function space of f, we can derive error bounds which are a function of h.

We can easily demonstrate how a MLP with hidden units activated by Heaviside functions (unit step functions) is a piecewise constant functional approximation. Let $f(x)$ be any function whose domain is $\Omega = [0, 1]$. Suppose that the function is Lipschitz continuous, that is,

$$\forall x, x' \in [0, 1), \ |f(x') - f(x)| \leq L|x' - x|,$$

for some constant $L \geq 0$. Using Heaviside functions to activate the hidden units

$$H(x) = \begin{cases} 1, & x \geq 0, \\ 0, & x < 0, \end{cases} \tag{4.17}$$

we construct a neural network with $K = \lfloor \frac{L}{2\epsilon} + 1 \rfloor$ units in a single hidden layer that approximates $f(x)$ within $\epsilon > 0$. That is, $\forall x \in [0, 1), \ |f(x) - \hat{f}(x)| \leq \epsilon$, where $\hat{f}(x)$ is the output of the neural network given input x. Let $\epsilon' = \frac{\epsilon}{L}$. We shall show that the neural network is a linear combination of indicator functions, ϕ_k, with compact support over $[x_k - \epsilon', x_k + \epsilon')$ and centered about x_k:

$$\phi_k(x) = \begin{cases} 1 & [x_k - \epsilon', x_k + \epsilon'), \\ 0 & \text{otherwise.} \end{cases} \tag{4.18}$$

The $\{\phi_k\}_{k=1}^K$ are piecewise constant basis functions, $\phi_i(x_j) = \delta_{ij}$, and the first few are illustrated in Fig. 4.7 below. The basis functions satisfy the partition of unity property $\sum_{k=1}^K \phi_k(x) = 1, \ \forall x \in \Omega$.

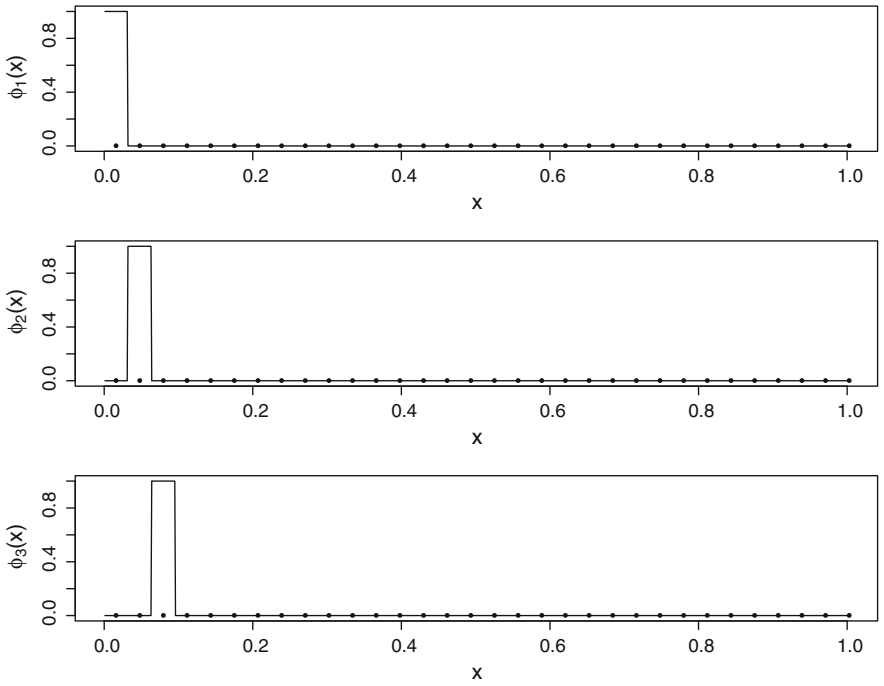


Fig. 4.7 The first three piecewise constant basis functions produced by the difference of neighboring step function activated units, $\phi_k(x) = H(x - (x_k - \epsilon')) - H(x - (x_k + \epsilon'))$

We shall construct such basis functions as the difference of Heaviside functions $\phi_k(x) = H(x - (x_k - \epsilon')) - H(x - (x_k + \epsilon'))$, $x_k = (2k - 1)\epsilon'$, by choosing the bias $b_k^{(1)} = -2(k - 1)\epsilon'$ and $W^{(1)} = \mathbf{1}$ so that the neural network, $\hat{f}(X) = W^{(2)}H(W^{(1)}X + b^{(1)})$ has values based on

$$H(W^{(1)}x + b^{(1)}) = \begin{bmatrix} H(x) \\ H(x - 2\epsilon') \\ \cdots \\ H(x - 2(k-1)\epsilon') \\ \cdots \\ H(x - (2K-1)\epsilon') \end{bmatrix}. \tag{4.19}$$

Then $W^{(2)}$ is set equal to exact function values and their differences:

$$W^{(2)} = [f(x_1), f(x_2) - f(x_1), \ldots, f(x_K) - f(x_{K-1})], \tag{4.20}$$

so that

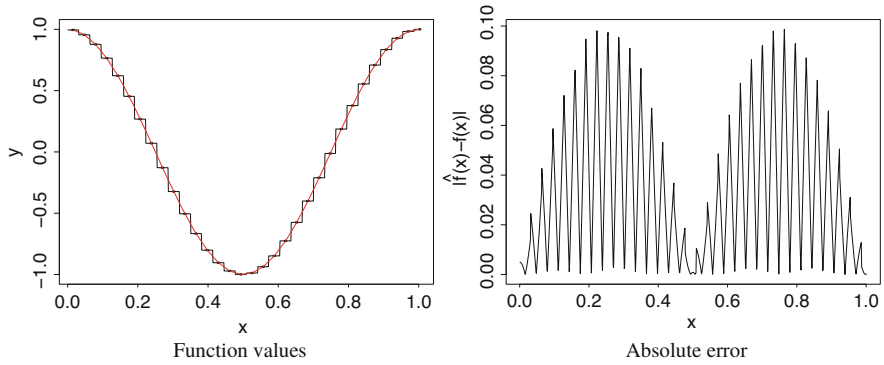


Fig. 4.8 The approximation of $\cos(2\pi x)$ using gridded input data and Heaviside activation functions. The error in approximation is at most ϵ with $K = \lfloor \frac{L}{2\epsilon} + 1 \rfloor$ hidden units

$$\hat{f} = \begin{cases} f(x_1), & x \leq 2\epsilon', \\ f(x_2), & 2\epsilon' < x \leq 4\epsilon', \\ \cdots & \cdots \\ f(x_k), & 2(k)\epsilon' < x \leq 2(k+1)\epsilon', \\ \cdots & \cdots \\ f(x_{K-1}), & 2(K-1)\epsilon' < x \leq 2K\epsilon'. \end{cases} \tag{4.21}$$

Figure 4.8 illustrates the function approximation for the case when $f(x) = \cos(2\pi x)$.

Since $x_k = (2k-1)\epsilon'$,we have that $\hat{Y} = f(x_k)$ over the interval $[x_k - \epsilon', x_k + \epsilon']$, which is the support of $\phi_k(x)$. By the Lipschitz continuity of $f(x)$, it follows that the worst-case error appears at the mid-point of any interval $[x_k, x_{k+1})$

$$|f(x_k + \epsilon') - \hat{f}(x + \epsilon')| = |f(x + \epsilon') - f(x_k)| \leq |f(x_k)| + L\epsilon' - |f(x_k)| = \epsilon. \tag{4.22}$$

This example is a special case of a more general representation permitted by MLPs. If we relax that the points need to be gridded, but instead just assume there are K data points in $\mathbb{R}^p$, then the region boundaries created by the K hidden units define a Voronoi diagram. Informally, a Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. The set of points are referred to as "seeds." For each seed there is a corresponding region consisting of all points closer to that seed than to any other. The discussion of Voronoi diagrams is beyond the scope of this chapter, but suffice to say that the representation of MLPs as splines extends to higher dimensional input spaces and higher degree splines.

Hence, under a special configuration of the weights and biases, with the hidden units defining Voronoi cells for each observation, we can show that a neural network is a univariate spline. This result generalizes to higher dimensional and higher order splines. Such a result enables us to view splines as a special case of a neural network which is consistent with our reasoning of neural networks as generalized approximation and regression techniques. The formulation of neural networks as splines allows approximation theory to guide the design of the network. Unfortunately, equating neural networks with splines says little about why and when multiples layers are needed.

2.7 Why Deep Networks?

The extension to deep neural networks is in fact well motivated on statistical and information-theoretical grounds (Tishby and Zaslavsky 2015; Poggio 2016; Mhaskar et al. 2016; Martin and Mahoney 2018; Bartlett et al. 2017a). Poggio (2016) shows that deep networks can achieve superior performance versus linear additive models, such as linear regression, while avoiding the curse of dimensionality. There are additionally many recent theoretical developments which characterize the approximation behavior as a function of network depth, width, and sparsity level (Polson and Rockova 2018). Recently (Bartlett et al. 2017b) prove upper and lower bounds on the expressibility of deep feedforward neural network classifiers with the piecewise linear activation function, such as ReLU activation functions. These bounds are tight for almost the entire range of parameters. Letting n denote the total number of weights, they prove that the VC dimension is $O(nLlog(n))$.

> VC Dimension Theorem

Theorem (Bartlett et al. (2017b)) *There exists a universal constant C such that the following holds. Given any W, L with $W > CL > C^2$, there exists a ReLU network with $\leq L$ layers and $\leq W$ parameters with VC dimension $\geq WLlog(W/L)/C$.* □

They further showed the effect of network depth on VC dimension with different non-linearities: there is no dependence for piecewise constant, linear dependence for piecewise-linear, and no more than quadratic dependence for general piecewise-polynomial. Thus the relationship between expressibility and depth is determined by the degree of the activation function. There is further ample theoretical evidence to

suggest that shallow networks cannot approximate the class of non-linear functions represented by deep ReLU networks without blow-up. Telgarsky (2016) shows that there is a ReLU network with L layers such that any network approximating it with only $O(L^{1/3})$ layers must have $\Omega(2^{L^{1/3}})$ units. Mhaskar et al. (2016) discuss the differences between composition versus additive models and show that it is possible to approximate higher polynomials much more efficiently with several hidden layers than a single hidden layer.

Martin and Mahoney (2018) shows that deep networks are implicitly self-regularizing behaving like Tikhonov regularization. Tishby and Zaslavsky (2015) characterizes the layers as "statistically decoupling" the input variables.

2.7.1 Approximation with Compositions of Functions

To gain some intuition as to why function composition can lead to successively more accurate function representation with each layer, consider the following example of a binary expansion of a decimal x.

Example 4.1 Binary Expansion of a Decimal

For each integer $n \geq 1$ and $x \in [0, 1]$, define $f_n(x) = x_n$, where x_n is the nth binary digit of x. The binary expansion of $x = \sum_{n=1}^{\infty} \frac{x_n}{2^n}$, where x_n is 1 or 0 depends on whether $X_{n-1} \geq \frac{1}{2^n}$ or otherwise, respectively, and $X_n := x - \sum_{i=1}^{n} \frac{x_i}{2^i}$. For example, we can find the first binary digit, x_1 as either 1 or 0 depending on whether $x_0 = x \geq \frac{1}{2}$. Now consider $X_1 = x - x_1/2$ and set $x_2 = 1$ if $X_1 \geq \frac{1}{2^2}$ or $x_2 = 0$ otherwise.

Example 4.2 Neural Network for Binary Expansion of a Decimal

A deep feedforward network for such a binary expansion of a decimal uses two neurons in each layer with different activations—Heaviside and identity functions. The input weight matrix, $W^{(1)}$, is the identity matrix, the other weight matrices, $\{W^{(\ell)}\}_{\ell>1}$ are

$$W^{(\ell)} = \begin{bmatrix} -\frac{1}{2^{\ell-1}} & 1 \\ -\frac{1}{2^{\ell-1}} & 1 \end{bmatrix},$$

and $\sigma_1^{(\ell)}(x) = H(x, \frac{1}{2^\ell})$ and $\sigma_2^{(\ell)}(x) = id(x) = x$. There are no bias terms. The output after ℓ hidden layers is the error, $X_\ell \leq \frac{1}{2^\ell}$.

While the example of converting a decimal in binary format using a binary expansion is simple, the approach can be readily generalized to the binary expansion of polynomials.

Theorem 4.2 (Liang and Srikant (2016)) *For the pth order polynomial* $f(x) = \sum_{i=0}^{p} a_i x^i$, $x \in [0, 1]$ *and* $\sum_{i=1}^{p} |a_i| \leq 1$, *there exists a multilayer neural network* $\hat{f}(x)$ *with* $O\left(p + \log \frac{p}{\varepsilon}\right)$ *layers,* $O\left(\log \frac{p}{\varepsilon}\right)$ *Heaviside units, and* $O\left(p \log \frac{p}{\varepsilon}\right)$ *rectifier linear units such that* $|f(x) - \hat{f}(x)| \leq \varepsilon$, $\forall x \in [0, 1]$.

Proof The sketch of the proof is as follows. Liang and Srikant (2016) use the deep structure shown in Fig. 4.9 to find the n-step binary expansion $\sum_{i=0}^{n} a_i x^i$ of x. Then they construct a multilayer network to approximate polynomials $g_i(x) = x^i$, $i = 1, \ldots, p$. Finally, they analyze the approximation error which is

$$|f(x) - \hat{f}(x)| \leq \frac{p}{2^{n-1}}.$$

See Appendix "Proof of Theorem 4.2" for the proof. □

2.7.2 Composition with ReLU Activation

An intuitive way to understand the importance of multiple network layers is to consider the effect of composing piecewise affine functions instead of adding them. It is easy to see that combinations of ReLU activated neurons give piecewise affine

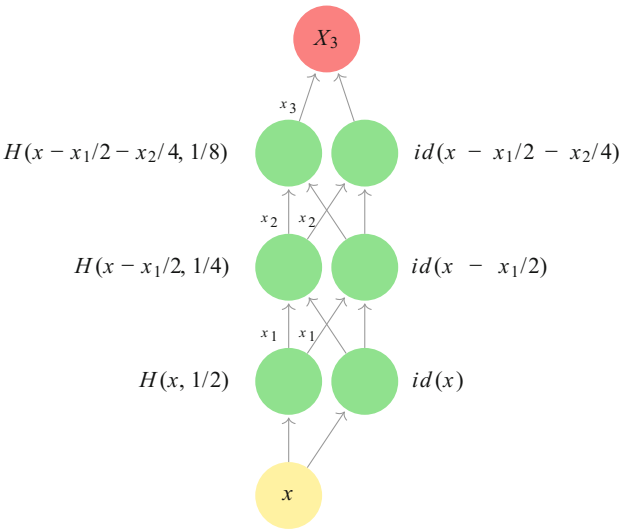


Fig. 4.9 An illustrative example of a deep feedforward neural network for binary expansion of a decimal. Each layer has two neurons with different activations—Heaviside and identity functions

approximations. For example, consider the shallow ReLU network with 2 and 4 perceptrons in Fig. 4.10:

$$F_{W,b} = W^{(2)}\sigma(W^{(1)}x + b^{(1)}),\ \sigma := \max(x, 0).$$

Let us start by defining $\sigma : \mathbb{R} \to \mathbb{R}$ to be *t-sawtooth* if it is piecewise affine with t pieces, meaning $\mathbb{R}$ is partitioned into t consecutive intervals, and σ is affine within each interval. Consequently, $ReLU(x)$ is 2-sawtooth, but this class also includes many other functions, for instance, the decision stumps used in boosting are 2-sawtooth, and decision trees with $t - 1$ nodes correspond to t-sawtooths. The following lemma serves to build intuition about the effect of adding versus composing sawtooth functions which is illustrated in Fig. 4.11.

Lemma 4.1 *Let $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$ be, respectively, k- and l-sawtooth. Then $f + g$ is $(k + l)$-sawtooth, and $f \circ g$ is kl-sawtooth.* □

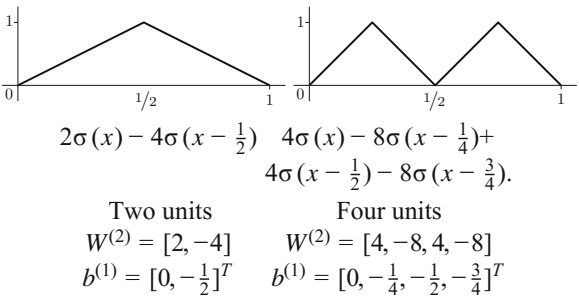


Fig. 4.10 A Shallow ReLU network with (left) two perceptrons and (right) four perceptrons

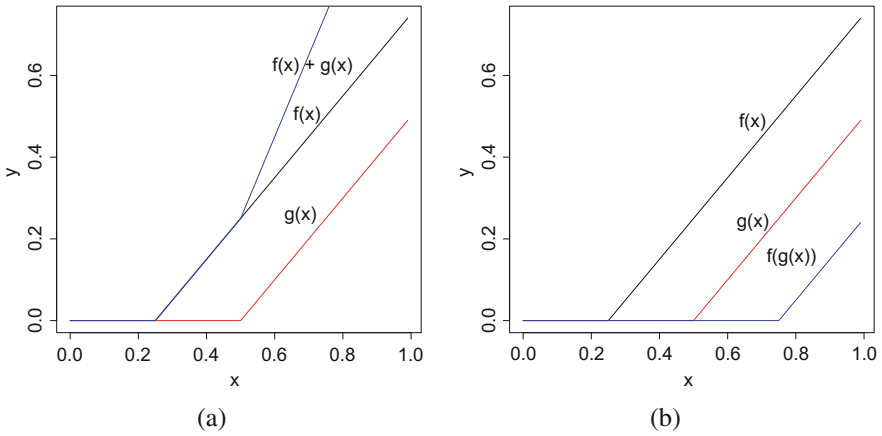


Fig. 4.11 Adding versus composing 2-sawtooth functions. (**a**) Adding 2-sawtooths. (**b**) Composing 2-sawtooths

Let us now build on this result by considering the *mirror map* $f_{\mathrm{m}} : \mathbb{R} \to \mathbb{R}$, which is shown in Fig. 4.12, and defined as

$$f_{\mathrm{m}}(x) := \begin{cases} 2x & \text{when } 0 \leq x \leq 1/2, \\ 2(1-x) & \text{when } 1/2 < x \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

Note that f_{m} can be represented by a two-layer ReLU activated network with two neurons; For instance, $f_{\mathrm{m}}(x) = (2\sigma(x) - 4\sigma(x - 1/2))$. Hence f_{m}^k is the composition of k (identical) ReLU sub-networks. A key observation is that fewer hidden units are needed to shatter a set of points when the network is deep versus shallow.

Consider, for example, the sequence of $n = 2^k$ points with alternating labels, referred to as the n-ap, and illustrated in Fig. 4.13 for the case when $k = 3$. As the x values pass from left to right, the labels change as often as possible and provide the most challenging arrangement for shattering n points.

There are many ways to measure the representation power of a network, but we will consider the classification error here. Suppose that we have a σ activated network with m units per layer and l layers. Given a function $f : \mathbb{R}^p \to \mathbb{R}$ let $\tilde{f} : \mathbb{R}^p \to \{0, 1\}$ denote the corresponding classifier $\tilde{f}(x) := \mathbb{1}_{f(x) \geq 1/2}$, and additionally given a sequence of points $((x_i, y_i))_{i=1}^n$ with $x_i \in \mathbb{R}^p$ and $y_i \in \{0, 1\}$, define the classification error as $\mathcal{E}(f) := \frac{1}{n} \sum_i \mathbb{1}_{\tilde{f}(x_i) \neq y_i}$.

Given a sawtooth function, its classification error on the n-ap may be lower bounded as follows.

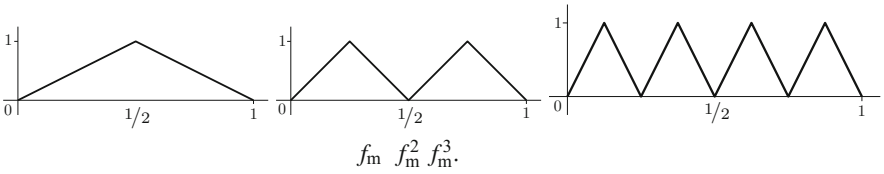


Fig. 4.12 The mirror map composed with itself

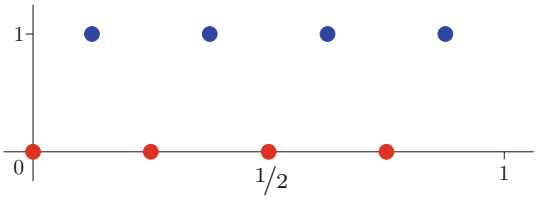


Fig. 4.13 The n-ap consists of n uniformly spaced points with alternating labels over the interval $[0, 1 - 2^{-n}]$. That is, the points $((x_i, y_i))_{i=1}^n$ with $x_i = i2^{-n}$ and $y_i = 0$ when i is even, and otherwise $y_i = 1$

Lemma 4.2 *Let $((x_i, y_i))_{i=1}^n$ be given according to the n-ap. Then every t-sawtooth function $f : \mathbb{R} \to \mathbb{R}$ satisfies $\mathcal{E}(f) \geq (n - 4t)/(3n)$.* □

The proof in the appendix relies on a simple counting argument for the number of crossings of 1/2. If there are m t-saw-tooth functions, then by Lemma 4.1, the resultant is a piecewise affine function over mt intervals. The main theorem now directly follows from Lemma 4.2.

Theorem 4.3 *Let positive integer k, number of layers l, and number of nodes per layer m be given. Given a t-sawtooth $\sigma : \mathbb{R} \to \mathbb{R}$ and $n := 2^k$ points as specified by the n-ap, then*

$$\min_{W,b} \mathcal{E}(f) \geq \frac{n - 4(tm)^l}{3n}.$$

From this result one can say, for example, that on the n-ap one needs $m = 2^{k-3}$ many units when classifying with a ReLU activated shallow network versus only $m = 2^{(1/l(k-2)-1)}$ units per layer for a $l \geq 2$ deep network.

Research on deep learning is very active and there are still many questions that need to be addressed before deep learning is fully understood. However, the purpose of these examples is to build intuition and motivate the need for many hidden layers in addition to the effect of increasing the number of neurons in each hidden layer.

In the remaining part of this chapter we turn towards the practical application of neural networks and consider some of the primary challenges in the context of financial modeling. We shall begin by considering how to preserve the shape of functions being approximated and, indeed, how to train and evaluate a network.

3 Convexity and Inequality Constraints

It may be necessary to restrict the range of $\hat{f}(x)$ or impose certain properties which are known about the shape of the function $f(x)$ being approximated. For example, $V = f(S)$ might be an option price and S the value of the underlying asset and convexity and non-negativity of $\hat{f}(S)$ are necessary. Consider the following feedforward network architecture $F_{W,b}(X) : \mathbb{R}^p \to \mathbb{R}^d$:

$$\hat{Y} = F_{W,b}(X) = f^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ f^{(1)}_{W^{(1)},b^{(1)}}(X), \tag{4.23}$$

where

$$f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}(x) = \sigma(W^{(\ell)}x + b^{(\ell)}), \ \forall \ell \in \{1, \ldots, L\}. \tag{4.24}$$

Convexity

For convexity of $\hat{Y}$ w.r.t. x, the activation function, $\sigma(x)$, must be a convex function of x. For avoidance of doubt, this convexity constraint should not be confused with convexity of the loss function w.r.t. the weights as in, for example, Bengio et al. (2006).

Examples[3] include $\text{ReLU}(x) := \max(x, 0)$ and $\text{softplus}(x; t) := \frac{1}{t} ln(1 + \exp\{tx\})$. For this class of activation functions, the semi-affine function $f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}(x) = \sigma(W^{(\ell)}x + b^{(\ell)})$ must also be convex in x since a convex function of a linear combination of x is also convex in x. The composition, $f^{(\ell+1)}_{W^{(\ell+1)},b^{(\ell+1)}} \circ f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}(x)$, is convex if and only if $f^{(\ell+1)}_{W^{(\ell+1)},b^{(\ell+1)}}(x)$ is non-decreasing convex and $f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}(x)$ is convex. The proof is left to the reader as a straightforward exercise. Hence, for convexity of $\hat{f}(x) = F_{W,b}(x)$ w.r.t. x we require that the weights in all but the first layer be positive:

$$w^{(\ell)}_{ij} \geq 0, \forall i, j, \forall \ell \in \{2, \ldots, L\}. \tag{4.25}$$

The constraints on the weights needed to enforce convexity guarantee non-negative output if the bias $b^{(L)}_i \geq 0, \forall i \in \{1, \ldots, d\}$ and $\sigma(x) \geq 0, \forall x$. Since $w^{(L)}_{ij} \geq 0, \forall i, j$ it follows that

$$w^{(L)}_{ij}\sigma(I^{(L-1)}_i) \geq 0, \tag{4.26}$$

and with non-negative bias terms, $\hat{f}_i$ is non-negative.

We now separately consider bounding the network output independently of imposing convexity on $\hat{f}_i(x)$ w.r.t. x. If we choose a bounded activation function $\sigma \in [\underline{\sigma}, \bar{\sigma}]$, then we can easily impose linear inequality constraints to ensure that $\hat{f}_i \in [c_i, d_i]$

$$c_i \leq \hat{f}_i(x) \leq d_i, d_i > c_i, \; i \in \{1, \ldots, d\}, \tag{4.27}$$

by setting

$$b^{(L)}_i = c_i - \sum_{j=1}^{n^{(L-1)}} \min(s_{ij}|\underline{\sigma}|, s_{ij}|\bar{\sigma}|)|w^{(L)}_{ij}|, \; s_{ij} := \text{sign}(w^{(L)}_{ij}). \tag{4.28}$$

[3]The parameterized softplus function $\sigma(x; t) = \frac{1}{t} ln(1+\exp\{tx\})$, with a model parameter $t >> 1$, converges to the ReLU function in the limit $t \to \infty$.

Note that the expression inside the min function can be simplified further to $\min(s_{ij}|\underline{\sigma}|, s_{ij}|\bar{\sigma}|)|w_{ij}| = \min(w_{ij}|\underline{\sigma}|, w_{ij}|\bar{\sigma}|)$. Training of the weights and biases is a constrained optimization problem with the linear constraints

$$\sum_{j=1}^{n^{(L-1)}} \max(s_{ij}|\underline{\sigma}|, s_{ij}|\bar{\sigma}|)|w_{ij}^{(L)}| \leq d_i - b_i^{(L)}, \tag{4.29}$$

which can be solved with the method of Lagrange multipliers or otherwise. If we require that $\hat{f}_i$ should be convex and bounded in the interval $[c_i, d_i]$, then the additional constraint, $w_{ij}^{(L)} \geq 0, \forall i, j$, is needed of course and the above simplifies to

$$b_i^{(L)} = c_i - \underline{\sigma} \sum_{j=1}^{n^{(L-1)}} w_{ij}^{(L)} \tag{4.30}$$

and solving the underdetermined system for $w_{ij}^{(L)}, \forall j$:

$$\sum_{j=1}^{n^{(L-1)}} \bar{\sigma}\, w_{ij}^{(L)} \leq d_i - b_i^{(L)} \tag{4.31}$$

$$\sum_{j=1}^{n^{(L-1)}} w_{ij}^{(L)} \leq \frac{d_i - c_i}{(\bar{\sigma} - \underline{\sigma})}. \tag{4.32}$$

The following toy examples provide simplified versions of constrained learning problems that arise in derivative modeling and calibration. The examples are intended only to illustrate the methodology introduced here. The first example is motivated by the need to learn an arbitrage free option price as a function of the underlying asset price. In particular, there are three scenarios where neural networks, and more broadly, supervised machine learning is useful for pricing. First, it provides a "model-free" framework, where no data generation process is assumed for the underlying dynamics. Second, machine learning can price complex derivatives where no analytic solution is known. Finally, machine learning does not suffer from the curse of dimensionality w.r.t. to the input space and can thus scale to basket options, options on many underlying assets. Each of these aspects merits further exploration and our example illustrates some of the challenges with learning pricing functions.

Perhaps the single largest defect of conventional derivative pricing models, however, is their calibration to data. Machine learning, too, provides an answer here—it provides a method for learning the relationship between market and contract variables and the model parameters.

Example 4.3 Approximating Option Prices

The payoff of a European call option at expiry time T is $V_T = \max(S_T - K, 0)$ and is convex with respect to S. Under the risk-neutral measure the option price at time t is the conditional expectation $V_t = \mathbb{E}_t[\exp\{-r(T-t)\}V_T]$. Since the conditional expectation is a linear operator, it preserves the convexity of the payoff function, so that the option price is always convex w.r.t. the underlying price. Thus, the second derivative, γ is always non-negative. Furthermore, the option price must always be non-negative.

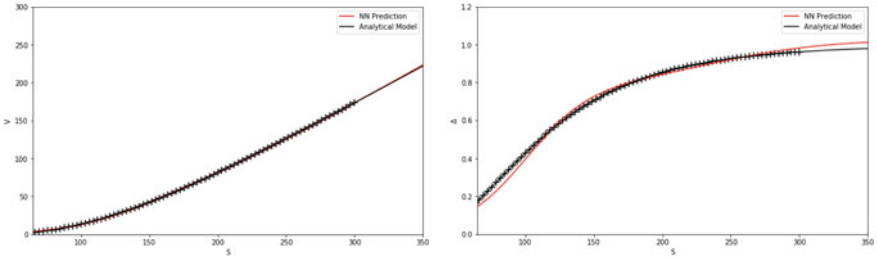
Let us approximate the surface of a European call option with strike K over all underlying values $S_t \in (0, \bar{S})$. The input variable $X \in \mathbb{R}^+$ are underlying asset prices and the outputs are call prices, so that the data is $\{S_i, V_i\}$. We use a neural network to learn the relation $V = f(S)$ and enforce the property that f is non-negative and convex w.r.t. S.

In the following example, we train the MLP over a uniform grid of 100 training points $S_i \in \Omega_h \subset [0.001, 300]$, and $V_i = f(S_i)$ generated by the Black–Scholes (BS) pricing formula. The risk-free rate $r = 0.01$, the strike is 130, the volatility is σ, and time to maturity is $T = 2.0$. The test data of 100 observations are on a different uniform gridded over a wider domain $[0.001, 600]$. The network uses one hidden layer ($L = 2$) with 100 units, a softplus activation function, and $w_{ij}^{(L)}, b_i^{(L)} \geq 0, \ \forall i, j$. Figure 4.14 compares the prediction with the BS model over the test set. $\hat{Y}$ is observed to be convex w.r.t. S because $w_{ij}^{(2)}$ is non-negative. Additionally, because $b_i^{(2)} \geq 0$ and $\underline{\sigma} = 0, \hat{Y} \geq 0$.

The figure also compares the Black–Scholes formula for the delta of the call option, $\Delta(X)$, the derivative of the price w.r.t. to S with the gradient of $\hat{Y}$:

$$\hat{\Delta}(X) = \partial_X \hat{Y} = (W^{(2)})^T D W^{(1)}, \ D_{ii} = \frac{1}{1 + \exp\{-\mathbf{w}_{i,}^{(1)} X - b_i^{(1)}\}}. \tag{4.33}$$

Under the BS model, the delta of a call option is in the interval $[0, 1]$. Note that the delta, although observed positive here, could be negative since there are no restrictions on $W^{(1)}$. Similarly, the delta approximation is observed to exceed unity. Thus, additional constraints are needed to bound the delta. For this architecture, imposing $w_{ij}^{(1)} \geq 0$ preserves the non-negativity of the delta and $\sum_j^{n^{(1)}} w_{ij}^{(2)} \mathbf{w}_{j,}^{(1)} \leq 1, \forall i$ bounds the delta at unity.



(a) Estimated call prices (b) Derived delta

Fig. 4.14 (**a**) The out-of-sample call prices are estimated using a single-layer neural network with constraints to ensure non-negativity and convexity of the price approximation w.r.t. the underlying price S. (**b**) The analytic derivative of $\hat{Y}$ is taken as the approximation of delta and compared over the test set with the Black–Scholes delta. We observe that additional constraints on the weights are needed to ensure that $\partial_X \hat{Y} \in [0, 1]$

Example 4.4 Calibrating Options

The goal is to learn the inverse of the Black–Scholes formula, as a function of moneyness, $M = S/K$. For simplicity, it considers the calibration of a chain of European in-the-money put or in-the-money equity call options with fixed time to maturity only. The input is moneyness for each option in the chain. The output of the neural network is the BS implied volatility—this is the implied volatility needed to calibrate the BS model to option price data corresponding to each moneyness.

The neural network preserves the positivity of the volatility and, in this example, imposes a convexity constraint on the surface w.r.t. to moneyness. The latter ensures consistency with liquid option markets, the implied volatility for both puts and calls typically monotonically increases as the strike price moves away from the current stock price—the so-called implied volatility smile. In markets, such as the equity markets, an implied volatility skew occurs because money managers usually prefer to write calls over puts.

The input variable $X \in \mathbb{R}^+$ is moneyness and the output is volatility so that the training data is $\{M_i, \sigma_i\}$. We use a neural network to learn the relation $\sigma = f(M)$ and enforce the property that f is non-negative and convex w.r.t. M. Note, in this example, that we do not directly learn the relationship between option prices and implied volatilities. Instead we learn how a BS root finder approximates the implied volatility as a function of the moneyness.

(continued)

Example 4.4 (continued)

In the following example, we train the MLP over a uniform grid of $n = 100$ training points $M_i \in \Omega_h \subset [0.5, 1 \times 10^4]$, and $\sigma_i = f(M_i)$ is generated by using a root finder for $V(\sigma; S, K_i, \tau, r) - \hat{V}_i = 0$, $\forall i = 1, \ldots, n$ and $\tau = 0.2$ years using the option price with strike K_i and time to maturity τ. The risk-free rate $r = 0.01$. The test data of 100 observations are on a different uniform gridded over a wider domain $[0.4166, 1 \times 10^4]$. The network uses one hidden layer ($L = 2$) with 100 units, a softplus activation function, and $w_{ij}^{(L)}, b_i^{(L)} \leq 0$, $\forall i, j$. Figure 4.15 compares the out-of-sample model output with the root finder for the BS model over the test set. $\hat{Y}$ is observed to be convex w.r.t. M because $w_{ij}^{(2)}$ is non-negative. Additionally, because $b_i^{(2)} \geq 0$ and $\underline{\sigma} = 0$, $\hat{Y} \geq 0$.

No-Arbitrage Pricing

The previous examples are simple enough to illustrate the application of constraints in neural networks. However, one would typically need to enforce more complex constraints for no-arbitrage pricing and calibration. Pricing approximations should be monotonically increasing w.r.t. to maturity and convex w.r.t. strike. Such constraints require that the neural network is fitted with more input variables K and T.

Accelerating Calibrations

One promising direction, which does not require neural network derivative pricing, is to simply learn a stochastic volatility based pricing model, such as the Heston model, as a function of underlying price, strike, and maturity, and then use the neural network pricing function to calibrate the pricing model. Such a calibration

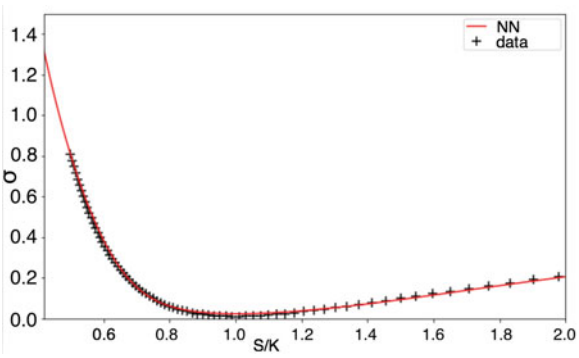


Fig. 4.15 The out-of-sample MLP estimation of implied volatility as a function of moneyness is compared with the true values

avoids fitting a few parameters to the chain of observed option prices or implied volatilities. Replacement of expensive pricing functions, which may require FFTs or Monte Carlo methods, with trained neural networks reduces calibration time considerably. See Horvath et al. (2019) for further details.

Dupire Local Volatility

Another challenge is how to price exotic options consistently with the market prices of their European counterpart. The former are typically traded over-the-counter, whereas the latter are often exchange traded and therefore "fully" observable. To fix ideas, let $C(K, T)$ denote an observed call price, for some fixed strike, K, maturity, T, and underlying price S_t. Modulo a short rate and dividend term, the unique "effective" volatility, σ_0^2, is given by the Dupire formula:

$$\sigma_0^2 = \frac{2\partial_T C(K, T)}{K^2 \partial_K^2 C(K, T)}. \tag{4.34}$$

The challenge arises when calibrating the local volatility model, extracting effective volatility from market option prices is an ill-posed inverse problem. Such a challenge has recently been addressed by Chataigner et al. (2020) in their paper on deep local volatility.

? Multiple Choice Question 2

Which of the following statements are true:

1. A feedforward architecture is always convex w.r.t. each input variables if every activation function is convex and the weights are constrained to be either all positive or all negative.
2. A feedforward architecture with positive weights is a monotonically increasing function of the input for any choice of monotonically increasing activation function.
3. The weights of a feedforward architecture must be constrained for the output of a feedforward network to be bounded.
4. The bias terms in a network simply shift the output and have no effect on the derivatives of the output w.r.t. to the input.

3.1 Similarity of MLPs with Other Supervised Learners

Under special circumstances, MLPs are functionally equivalent to a number of other machine learning techniques. As previously mentioned, when the network has no hidden layer, it is either a regression or logistic regression. Neural networks with

one hidden layer is essentially a projection pursuit regression (PPR), both project the input vector onto a hyperplane, apply a non-linear transformation into feature space, followed by an affine transformation. The mapping of input vectors to feature space by the hidden layer is conceptually similar to kernel methods, such as support vector machines (SVMs), which map to a kernel space, where classification and regression are subsequently performed. Boosted decision stumps, one level boosted decision trees, can even be expressed as a single-layer MLP. Caution must be exercised in over-stretching these conceptual similarities. Data generation assumptions aside, there are differences in the classes of non-linear functions and learning algorithms used. For example, the non-linear function being fitted in PPR can be different for each combination of input variables and is sequentially estimated before updating the weights. In contrast, neural networks fix these functions and estimate all the weights belonging to a single layer simultaneously. A summary of other machine learning approaches is given in Table 4.1 and we refer the reader to numerous excellent textbooks (Bishop 2006; Hastie et al. 2009) covering such methods.

Table 4.1 This table compares supervised machine learning algorithms (reproduced from Mullainathan and Spiess (2017))

Function class $\mathcal{F}$ (and its parameterization)	Regularizer $R(f)$
Global/parametric predictors	
Linear $\beta' x$ (and generalizations)	Subset selection $\|\beta\|_0 = \sum_{j=1}^{k} \mathbf{1}_{\beta_j \neq 0}$
	LASSO $\|\beta\|_1 = \sum_{j=1}^{k} \|\beta_j\|$
	Ridge $\|\beta\|_2^2 = \sum_{j=1}^{k} \beta_j^2$
	Elastic net $\alpha\|\beta\|_1 + (1-\alpha)\|\beta\|_2^2$
Local/non-parametric predictors	
Decision/regression trees	Depth, number of nodes/leaves, minimal leaf size, information gain at splits
Random forest (linear combination of trees)	Number of trees, number of variables used in each tree, size of bootstrap sample, complexity of trees (see above)
Nearest neighbors	Number of neighbors
Kernel regression	Kernel bandwidth
Mixed predictors	
Deep learning, neural nets, convolutional neural networks	Number of levels, number of neurons per level, connectivity between neurons
Splines	Number of knots, order
Combined predictors	
Bagging: unweighted average of predictors from bootstrap draws	Number of draws, size of bootstrap samples (and individual regularization parameters)
Boosting: linear combination of predictions of residual	Learning rate, number of iterations (and individual regularization parameters)
Ensemble: weighted combination of different predictors	Ensemble weights (and individual regularization parameters)

4 Training, Validation, and Testing

Deep learning is a data-driven approach which focuses on finding structure in large datasets. The main tools for variable or predictor selection are regularization and dropout. Out-of-sample predictive performance helps assess the optimal amount of regularization, the problem of finding the optimal hyperparameter selection. There is still a very Bayesian flavor to the modeling procedure and the modeler follows two key steps:

1. Training phase: pair the input with expected output, until a sufficiently close match has been found. Gauss' original least squares procedure is a common example.
2. Validation and test phase: assess how well the deep learner has been trained for out-of-sample prediction. This depends on the size of your data, the value you would like to predict, the input, etc., and various model properties including the mean-error for numeric predictors and classification errors for classifiers.

Often, the validation phase is split into two parts.

2.a First, estimate the out-of-sample accuracy of all approaches (a.k.a. validation).
2.b Second, compare the models and select the best performing approach based on the validation data (a.k.a. verification).

Step 2.b. can be skipped if there is no need to select an appropriate model from several rivaling approaches. The researcher then only needs to partition the dataset into a training and test set.

To construct and evaluate a learning machine, we start with training data of input–output pairs $D = \{Y^{(i)}, X^{(i)}\}_{i=1}^{N}$. The goal is to find the machine learner of $Y = F(X)$, where we have a loss function $\mathcal{L}(Y, \hat{Y})$ for a predictor, $\hat{Y}$, of the output signal, Y. In many cases, there is an underlying probability model, $p(Y \mid \hat{Y})$, then the loss function is the negative log probability $\mathcal{L}(Y, \hat{Y}) = -\log\ p(Y \mid \hat{Y})$. For example, under a Gaussian model $\mathcal{L}(Y, \hat{Y}) = ||Y - \hat{Y}||^2$ is a L^2 norm, for binary classification, $\mathcal{L}(Y, \hat{Y}) = -Y \log\ \hat{Y}$ is the negative cross-entropy. In its simplest form, we then solve an optimization problem

$$\underset{W,b}{\text{minimize}} f(W, b) + \lambda \phi(W, b)$$

$$f(W, b) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(Y^{(i)}, \hat{Y}(X^{(i)}))$$

with a regularization penalty, $\phi(W, b)$. The loss function is non-convex, possessing many local minima and is generally difficult to find a global minimum. An important assumption, which is often not explicitly stated, is that the errors are assumed to be "homoscedastic." Homoscedasticity is the assumption that the error has an identical distribution over each observation. This assumption can be relaxed by

weighting the observations differently. However, we shall regard such extensions as straightforward and compatible with the algorithms for solving the unweighted optimization problem. Here λ is a global regularization parameter which we tune using the out-of-sample predictive mean squared error (MSE) of the model. The regularization penalty, $\phi(W, b)$, introduces a bias–variance tradeoff . $\nabla\mathcal{L}$ is given in closed form by a chain rule and, through back-propagation, each layer's weights $\hat{W}^{(\ell)}$ are fitted with stochastic gradient descent.

Recall from Chap. 1 that a 1-of-K encoding is used for a categorical response, so that G is a K-binary vector $G \in [0, 1]^K$ and the value k is presented as $G_k = 1$ and $G_j = 0, \forall j \neq k$, where $||G_k||_1 = 1$. The predictor is given by $\hat{G}_k := g_k(X|(W, b))$, $||\hat{G}_k||_1 = 1$ and the loss function is the negative cross-entropy for discrete random variables

$$\mathcal{L}(G, \hat{G}(X)) = -G^T ln\hat{G}. \tag{4.35}$$

For example, if there are $K = 3$ classes, then $G = [0, 0, 1], G = [0, 1, 0]$, or $G = [1, 0, 0]$ to represent the three classes. When $K > 2$, the output layer has K neurons and the loss function is the negative cross-entropy

$$\mathcal{L}(G, \hat{G}(X)) = -\sum_{k=1}^{K} G_k ln\hat{G}_k. \tag{4.36}$$

For the case when $K = 2$, i.e. binary classification, there is only one neuron in the output layer and the loss function is

$$\mathcal{L}(G, \hat{G}(X)) = -Gln\hat{G} - (1 - G)ln(1 - \hat{G}), \tag{4.37}$$

where $\hat{G} = g_1(X|(W, b)) = \sigma(I^{(L-1)})$ and σ is a sigmoid function.

We observe that when there are no hidden layers, $I^{(1)} = W^{(1)}X + b^{(1)}$ and $g_1(X|(W, b))$ is a logistic regression. The *softmax* function, σ_s generalizes binary classifiers to multi-classifiers. $\sigma_s : \mathbb{R}^K \to [0, 1]^K$ is a continuous K-vector function given by

$$\sigma_s(\mathbf{x})_k = \frac{\exp(x_k)}{||\exp(\mathbf{x})||_1}, \ k \in \{1, \ldots, K\}, \tag{4.38}$$

where $||\sigma_s(\mathbf{x})||_1$=1. The *softmax* function is used to represent a probability distribution over K possible states:

$$\hat{G}_k = P(G = k \mid X) = \sigma_s(WX + b) = \frac{\exp((WX + b)_k)}{||\exp(WX + b)||_1}. \tag{4.39}$$

> Derivative of the Softmax Function

Using the quotient rule $f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2}$, the derivative $\sigma := \sigma_s(\mathbf{x})$ can be written as:

$$\frac{\partial \sigma_i}{\partial x_i} = \frac{\exp(x_i)||\exp(\mathbf{x})||_1 - \exp(x_i)\exp(x_i)}{||\exp(\mathbf{x})||_1^2} \tag{4.40}$$

$$= \frac{\exp(x_i)}{||\exp(\mathbf{x})||_1} \cdot \frac{||\exp(\mathbf{x})||_1 - \exp(x_i)}{||\exp(\mathbf{x})||_1} \tag{4.41}$$

$$= \sigma_i(1 - \sigma_i) \tag{4.42}$$

For the case $i \neq j$, the derivative of the sum is

$$\frac{\partial \sigma_i}{\partial x_j} = \frac{0 - \exp(x_i)\exp(x_j)}{||\exp(\mathbf{x})||_1^2} \tag{4.43}$$

$$= -\frac{\exp(x_j)}{||\exp(\mathbf{x})||_1} \cdot \frac{\exp(x_i)}{||\exp(\mathbf{x})||_1} \tag{4.44}$$

$$= -\sigma_j \sigma_i \tag{4.45}$$

This can be written compactly as $\frac{\partial \sigma_i}{\partial x_j} = \sigma_i(\delta_{ij} - \sigma_j)$, where δ_{ij} is the Kronecker delta function.

5 Stochastic Gradient Descent (SGD)

Stochastic gradient descent (SGD) method or its variation is typically used to find the deep learning model weights by minimizing the penalized loss function, $f(W, b)$. The method minimizes the function by taking a negative step along an estimate g^k of the gradient $\nabla f(W^k, b^k)$ at iteration k. The approximate gradient is calculated by

$$g^k = \frac{1}{b_k} \sum_{i \in E_k} \nabla \mathcal{L}_{W,b}(Y^{(i)}, \hat{Y}^k(X^{(i)})),$$

where $E_k \subset \{1, \ldots, N\}$ and $b_k = |E_k|$ is the number of elements in E_k (a.k.a. batch size). When $b_k > 1$ the algorithm is called batch SGD and simply SGD otherwise. A usual strategy to choose subset E is to go cyclically and pick consecutive elements of $\{1, \ldots, N\}$ and $E_{k+1} = [E_k \mod N]+1$, where modular arithmetic is applied to the set. The approximated direction g^k is calculated using a chain rule (a.k.a. back-propagation) for deep learning. It is an unbiased estimator of $\nabla f(W^k, b^k)$, and we have

$$\mathrm{E}(g^k) = \frac{1}{N} \sum_{i=1}^{N} \nabla \mathcal{L}_{W,b} \left(Y^{(i)}, \hat{Y}^k(X^{(i)}) \right) = \nabla f(W^k, b^k).$$

At each iteration, we update the solution $(W, b)^{k+1} = (W, b)^k - t_k g^k$.

Deep learning applications use a step size t_k (a.k.a. learning rate) as constant or a reduction strategy of the form, $t_k = a \exp\{-kt\}$. Appropriate learning rates or the hyperparameters of reduction schedule are usually found empirically from numerical experiments and observations of the loss function progression. In order to update the weights across the layers, *back-propagation* is needed and will now be explained.

? Multiple Choice Question 3

Which of the following statements are true:

1. The training of a neural network involves minimizing a loss function w.r.t. the weights and biases over the training data.
2. L_1 regularization is used during model selection to penalize models with too many parameters.
3. In deep learning, regularization can be applied to each layer of the network.
4. Back-propagation uses the chain rule to update the weights of the network but is not guaranteed to convergence to a unique minimum.
5. Stochastic gradient descent and back-propagation are two different optimization algorithms for minimizing the loss function and the user must choose the best one.

5.1 Back-Propagation

Staying with a multi-classifier, we can begin by informally motivating the need for a recursive approach to updating the weights and biases. Let us express $\hat{Y} \in [0, 1]^K$ as a function of the final weight matrix $W \in \mathbb{R}^{K \times M}$ and output bias $\mathbb{R}^K$ so that

$$\hat{Y}(W, b) = \sigma \circ I(W, b), \tag{4.46}$$

where the input function $I : \mathbb{R}^{K\times M} \times \mathbb{R}^K \to \mathbb{R}^K$ is of the form $I(W, b) := WX + b$ and $\sigma : \mathbb{R}^K \to \mathbb{R}^K$ is the softmax function. Applying the multivariate chain rule gives the Jacobian of $\hat{Y}(W, b)$:

$$\nabla \hat{Y}(W, b) = \nabla(\sigma \circ I)(W, b) \tag{4.47}$$

$$= \nabla\sigma(I(W, b)) \cdot \nabla I(W, b). \tag{4.48}$$

5.1.1 Updating the Weight Matrices

Recall that the loss function for a multi-classifier is the cross-entropy

$$\mathcal{L}(Y, \hat{Y}(X)) = -\sum_{k=1}^{K} Y_k ln \hat{Y}_k. \tag{4.49}$$

Since Y is a constant vector we can express the cross-entropy as a function of (W, b)

$$\mathcal{L}(W, b) = \mathcal{L} \circ \sigma(I(W, b)). \tag{4.50}$$

Applying the multivariate chain rule gives

$$\nabla\mathcal{L}(W, b) = \nabla(\mathcal{L} \circ \sigma)(I(W, b)) \tag{4.51}$$

$$= \nabla\mathcal{L}(\sigma(I(W, b))) \cdot \nabla\sigma(I(W, b)) \cdot \nabla I(W, b). \tag{4.52}$$

Stochastic gradient descent is used to find the minimum

$$(\hat{W}, \hat{b}) = \arg\min_{W,b} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{Y}^{W,b}(\mathbf{x}_i)). \tag{4.53}$$

Because of the compositional form of the model, the gradient must be derived using the chain rule for differentiation. This can be computed by a forward and then a backward sweep ("back-propagation") over the network, keeping track only of quantities local to each neuron.

Forward Pass

Set $Z^{(0)} = X$ and for $\ell \in \{1, \ldots, L\}$ set

$$Z^{(\ell)} = f^{(\ell)}_{W^{(\ell)},b^{(\ell)}}(Z^{(\ell-1)}) = \sigma^{(\ell)}(W^{(\ell)} Z^{(\ell-1)} + b^{(\ell)}). \tag{4.54}$$

On completion of the forward pass, the error $\hat{Y} - Y$ is evaluated using $\hat{Y} := Z^{(L)}$.

Back-Propagation

Define the back-propagation error $\delta^{(\ell)} := \nabla_{b^{(\ell)}}\mathcal{L}$, where given $\delta^{(L)} = \hat{Y} - Y$, and for $\ell = L-1, \ldots, 1$ the following recursion relation gives the updated back-propagation error and weight update for layer ℓ:

$$\delta^{(\ell)} = (\nabla_{I^{(\ell)}}\sigma^{(\ell)})W^{(\ell+1)T}\delta^{(\ell+1)}, \tag{4.55}$$

$$\nabla_{W^{(\ell)}}\mathcal{L} = \delta^{(\ell)} \otimes Z^{(\ell-1)}, \tag{4.56}$$

and $\otimes$ is the outer product of two vectors. See Appendix "Back-Propagation" for a derivation of Eqs. 4.55 and 4.56.

The weights and biases are updated for all $\ell \in \{1, \ldots, L\}$ according to the expression

$$\Delta W^{(\ell)} = -\gamma \nabla_{W^{(\ell)}}\mathcal{L} = -\gamma \delta^{(\ell)} \otimes Z^{(\ell-1)},$$
$$\Delta b^{(\ell)} = -\gamma \delta^{(\ell)},$$

where γ is a user defined learning rate parameter. Note the negative sign: this indicates that weight changes are in the direction of decrease in error. *Mini-batch* or off-line updates involve using many observations of X at the same time. The *batch size* refers to the number of observations of X used in each pass. An *epoch* refers to a round-trip (i.e., forward+backward pass) over all training samples.

Example 4.5 Back-Propagation with a Three-Layer Network

Suppose that a feedforward network classifier has two sigmoid activated hidden layers and a softmax activated output layer. After a forward pass, the values of $\{Z^{(\ell)}\}_{\ell=1}^{3}$ are stored and the error $\hat{Y} - Y$, where $\hat{Y} = Z^{(3)}$, is calculated for one observation of X. The back-propagation and weight updates in the final layer are evaluated:

$$\delta^{(3)} = \hat{Y} - Y$$
$$\nabla_{W^{(3)}}\mathcal{L} = \delta^{(3)} \otimes Z^{(2)}.$$

Now using Eqs. 4.55 and 4.56, we update the back-propagation error and weight updates for hidden layer 2

$$\delta^{(2)} = Z^{(2)}(1 - Z^{(2)})(W^{(3)})^T\delta^{(3)},$$
$$\nabla_{W^{(2)}}\mathcal{L} = \delta^{(2)} \otimes Z^{(1)}.$$

(continued)

Example 4.4 (continued)

Repeating for hidden layer 1

$$\delta^{(1)} = Z^{(1)}(1 - Z^{(1)})(W^{(2)})^T \delta^{(2)},$$
$$\nabla_{W^{(1)}} \mathcal{L} = \delta^{(1)} \otimes X.$$

We update the weights and biases using Eqs. 4.57 and 4.57, so that $b^{(3)} \rightarrow b^{(3)} - \gamma \delta^{(3)}$, $W^{(3)} \rightarrow W^{(3)} - \gamma \delta^{(3)} \otimes Z^{(2)}$ and repeat for the other weight-bias pairs, $\{(W^{(\ell)}, b^{(\ell)})\}_{\ell=1}^{2}$. See the back-propagation notebook for further details of a worked example in Python and then complete Exercise 4.12.

5.2 *Momentum*

One disadvantage of SGD is that the descent in f is not guaranteed or can be very slow at every iteration. Furthermore, the variance of the gradient estimate g^k is near zero as the iterates converge to a solution. To address those problems a coordinate descent (CD) and momentum-based modifications of SGD are used. Each CD step evaluates a single component E_k of the gradient ∇f at the current point and then updates the E_kth component of the variable vector in the negative gradient direction. The momentum-based versions of SGD or the so-called accelerated algorithms were originally proposed by Nesterov (2013).

The use of momentum in the choice of step in the search direction combines new gradient information with the previous search direction. These methods are also related to other classical techniques such as the heavy-ball method and conjugate gradient methods. Empirically momentum-based methods show a far better convergence for deep learning networks. The key idea is that the gradient only influences changes in the "velocity" of the update

$$v^{k+1} = \mu v^k - t_k g^k,$$
$$(W, b)^{k+1} = (W, b)^k + v^k.$$

The parameter μ controls the dumping effect on the rate of update of the variables. The physical analogy is the reduction in kinetic energy that allows "slow down" in the movements at the minima. This parameter is also chosen empirically using cross-validation.

Nesterov's momentum method (a.k.a. Nesterov acceleration) instead calculate gradient at the point predicted by the momentum. We can think of it as a look-ahead strategy. The resulting update equations are

$$
\begin{aligned}
v^{k+1} &= \mu v^k - t_k g((W,b)^k + v^k), \\
(W,b)^{k+1} &= (W,b)^k + v^k.
\end{aligned}
$$

Another popular modification to the SGD method is the AdaGrad method, which adaptively scales each of the learning parameters at each iteration

$$
\begin{aligned}
c^{k+1} &= c^k + g((W,b)^k)^2, \\
(W,b)^{k+1} &= (W,b)^k - t_k g(W,b)^k)/(\sqrt{c^{k+1}} - a),
\end{aligned}
$$

where a is usually a small number, e.g. $a = 10^{-6}$ that prevents dividing by zero. PRMSprop takes the AdaGrad idea further and places more weight on recent values of the gradient squared to scale the update direction, i.e. we have

$$
c^{k+1} = dc^k + (1-d)g((W,b)^k)^2.
$$

The Adam method combines both PRMSprop and momentum methods and leads to the following update equations:

$$
\begin{aligned}
v^{k+1} &= \mu v^k - (1-\mu) t_k g((W,b)^k + v^k), \\
c^{k+1} &= dc^k + (1-d)g((W,b)^k)^2, \\
(W,b)^{k+1} &= (W,b)^k - t_k v^{k+1}/(\sqrt{c^{k+1}} - a).
\end{aligned}
$$

Second-order methods solve the optimization problem by solving a system of non-linear equations $\nabla f(W,b) = 0$ with Newton's method

$$
(W,b)^+ = (W,b) - \{\nabla^2 f(W,b)\}^{-1} \nabla f(W,b).
$$

SGD simply approximates $\nabla^2 f(W,b)$ by $1/t$. The advantages of a second-order method include much faster convergence rates and insensitivity to the conditioning of the problem. In practice, second-order methods are rarely used for deep learning applications (Dean et al. 2012). The major disadvantage is the inability to train the model using batches of data as SGD does. Since typical deep learning models rely on large-scale datasets, second-order methods become memory and computationally prohibitive at even modest-sized training datasets.

5.2.1 Computational Considerations

Batching alone is not sufficient to scale SGD methods to large-scale problems on modern high-performance computers. Back-propagation through a chain rule creates an inherit sequential dependency in the weight updates which limits the

dataset dimensions for the deep learner. Polson et al. (2015) consider a proximal Newton method, a Bayesian optimization technique which provides an efficient solution for estimation and optimization of such models and for calculating a regularization path. The authors present a splitting approach, alternating direction method of multipliers (ADMM), which overcomes the inherent bottlenecks in back-propagation by providing a simultaneous block update of parameters at all layers. ADMM facilitates the use of large-scale computing.

A significant factor in the widespread adoption of deep learning has been the creation of `TensorFlow` (Abadi et al. 2016), an interface for easily expressing machine learning algorithms and mapping compute intensive operations onto a wide variety of different hardware platforms and in particular GPU cards. Recently, `TensorFlow` has been augmented by `Edward` (Tran et al. 2017) to combine concepts in Bayesian statistics and probabilistic programming with deep learning.

5.2.2 Model Averaging via Dropout

We close this section by briefly mentioning one final technique which has proved indispensable in preventing neural networks from over-fitting. Dropout is a computationally efficient technique to reduce model variance by considering many model configurations and then averaging the predictions. The layer input space $Z = (Z_1, \ldots, Z_n)$, where n is large, needs dimension reduction techniques which are designed to avoid over-fitting in the training process. Dropout works by removing layer inputs randomly with a given probability θ. The probability, θ, can be viewed as a further hyperparameter (like λ) which can be tuned via cross-validation. Heuristically, if there are 1000 variables, then a choice of $\theta = 0.1$ will result in a search for models with 100 variables. The dropout architecture with stochastic search for the predictors can be used

$$
\begin{aligned}
d_i^{(\ell)} &\sim \text{Ber}(\theta), \\
\tilde{Z}^{(\ell)} &= d^{(\ell)} \circ Z^{(\ell)}, \ 1 \leq \ell < L, \\
Z^{(\ell)} &= \sigma^{(\ell)}(W^{(\ell)} \tilde{Z}^{(\ell-1)} + b^{(\ell)}).
\end{aligned}
$$

Effectively, this replaces the layer input Z by $d \circ Z$, where $\circ$ denotes the element-wise product and d is a vector of independent Bernoulli, Ber(θ), distributed random variables. The overall objective function is closely related to ridge regression with a g-prior (Heaton et al. 2017).

6 Bayesian Neural Networks*

Bayesian deep learning (Neal 1990; Saul et al. 1996; Frey and Hinton 1999; Lawrence 2005; Adams et al. 2010; Mnih and Gregor 2014; Kingma and Welling 2013; Rezende et al. 2014) provides a powerful and general framework for statistical modeling. Such a framework allows for a completely new approach to data modeling and solves a number of problems that conventional models cannot address: (i) DLs (deep learners) permit complex dependencies between variables to be explicitly represented which are difficult, if not impossible, to model with copulas; (ii) they capture correlations between variables in high-dimensional datasets; and (iii) they characterize the degree of uncertainty in predicting large-scale effects from large datasets relevant for quantifying uncertainty.

Uncertainty refers to the statistically unmeasurable situation of Knightian uncertainty, where the event space is known but the probabilities are not (Chen et al. 2017). Oftentimes, a forecast may be shrouded in uncertainty arising from noisy data or model uncertainty, either through incorrect modeling assumptions or parameter error. It is desirable to characterize this uncertainty in the forecast. In conventional Bayesian modeling, uncertainty is used to learn from small amounts of low-dimensional data under parametric assumptions on the prior. The choice of the prior is typically the point of contention and chosen for solution tractability rather than modeling fidelity. Recently, deterministic deep learners have been shown to scale well to large, high-dimensional, datasets. However, the probability vector obtained from the network is often erroneously interpreted as model confidence (Gal 2016).

A typical approach to model uncertainty in neural network models is to assume that model parameters (weights and biases) are random variables (as illustrated in Fig. 4.16). Then ANN model approaches Gaussian process as the number of weights goes to infinity (Neal 2012; Williams 1997). In the case of finite number of weights, a network with random parameters is called a Bayesian neural network (MacKay 1992b). Recent advances in "variational inference" techniques and software that represent mathematical models as a computational graph (Blundell et al. 2015a) enable probabilistic deep learning models to be built, without having to worry about how to perform testing (forward propagation) or inference (gradient- based optimization, with back-propagation and automatic differentiation). *Variational inference* is an approximate technique which allows multi-modal likelihood functions to be extremized with standard stochastic gradient descent algorithm. An alternative to variational and MCMC algorithms was recently proposed by Gal (2016) and builds on efficient dropout regularization technique.

All of the current techniques rely on approximating the true posterior over the model parameters $p(w \mid X, Y)$ by another distribution $q_\theta(w)$ which can be evaluated in a computationally tractable way. Such a distribution is chosen to be as close as possible to the true posterior and is found by minimizing the Kullback–Leibler (KL) divergence

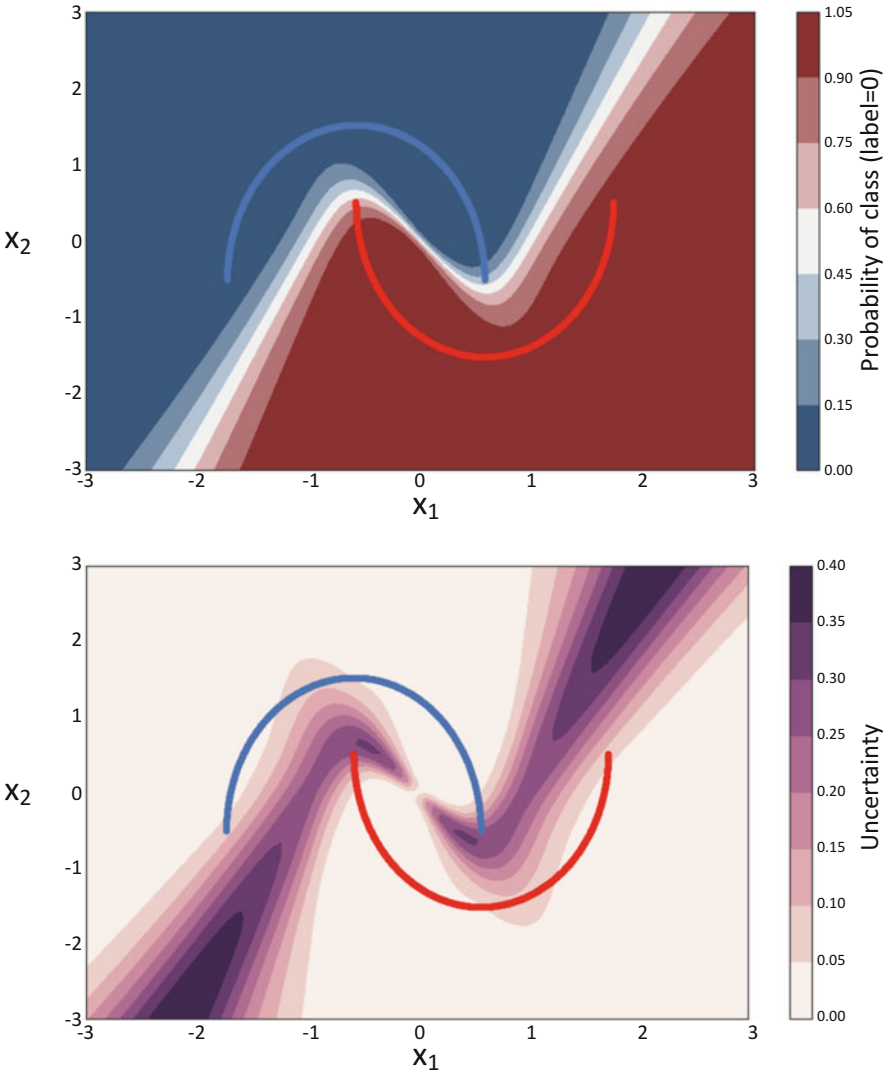


Fig. 4.16 Bayesian classification of the half-moon problem with neural networks. (top) The posterior mean and (bottom) the posterior std. dev.

$$\theta^* \in \arg\min_{\theta} \int q_\theta(w) \log \frac{q_\theta(w)}{p(w \mid X, Y)} dw.$$

There are numerous approaches to Bayesian deep learning for uncertainty quantification including MCMC (Markov chain Monte Carlo) methods. These are known to scale poorly with the number of observations and recent studies have

developed SG-MCMC (stochastic gradient MCMC) and related methods such as PX-MCMC (parameter expansion MCMC) to ease the computational burden. A Bayesian extension of feedforward network architectures has been considered by several authors (Neal 1990; Saul et al. 1996; Frey and Hinton 1999; Lawrence 2005; Adams et al. 2010; Mnih and Gregor 2014; Kingma and Welling 2013; Rezende et al. 2014). Recent results show how dropout regularization can be used to represent uncertainty in deep learning models. In particular, Gal (2015) shows that dropout provides uncertainty estimates for the predicted values. The predictions generated by the deep learning models with dropout are nothing but samples from the predictive posterior distribution.

A classical example of using neural networks to model a vector of binary variables is the Boltzmann machine (BM), with two layers. The first layer encodes latent variables and the second layer encodes the observed variables. Both conditional distributions $p(\text{data} \mid \text{latent variables})$ and $p(\text{latent variables} \mid \text{data})$ are specified using logistic functions parameterized by weights and offset vectors. The size of the joint distribution table grows exponentially with the number of variables and (Hinton and Sejnowski 1983) proposed using Gibbs sampler to calculate updates to model weights on each iteration. The multi-modal nature of the posterior distribution leads to prohibitive computational times required to learn models of a practical size. Tieleman (2008) proposed a variational approach that replaces the posterior $p(\text{latent variables} \mid \text{data})$ and approximates it with another easy to calculate distribution and was considered in Salakhutdinov (2008). Several extensions to the BMs have been proposed. Exponential family extensions have been considered by (Smolensky 1986; Salakhutdinov 2008; Salakhutdinov and Hinton 2009; Welling et al. 2005).

There have also been multiple approaches to building inference algorithms for deep learning models (MacKay 1992a; Hinton and Van Camp 1993; Neal 1992; Barber and Bishop 1998). Performing Bayesian inference on a neural network calculates the posterior distribution over the weights given the observations. In general, such a posterior cannot be calculated analytically, or even efficiently sampled from. However, several recently proposed approaches address the computational problem for some specific deep learning models (Graves 2011; Kingma and Welling 2013; Rezende et al. 2014; Blundell et al. 2015b; Hernández-Lobato and Adams 2015; Gal and Ghahramani 2016).

The recent successful approaches to develop efficient Bayesian inference algorithms for deep learning networks are based on the reparameterization techniques for calculating Monte Carlo gradients while performing variational inference. Such an approach has led to an explosive development in the application of stochastic variational inference. Given the data $\mathcal{D} = (X, Y)$, the variational inference relies on approximating the posterior $p(\theta \mid \mathcal{D})$ with a variation distribution $q(\theta \mid \mathcal{D}, \phi)$, where $\theta = (W, b)$. Then q is found by minimizing the Kullback–Leibler divergence between the approximate distribution and the posterior, namely

$$\text{KL}(q \mid\mid p) = \int q(\theta \mid \mathcal{D}, \phi) \log \frac{q(\theta \mid \mathcal{D}, \phi)}{p(\theta \mid \mathcal{D})} d\theta.$$

Since $p(\theta \mid \mathcal{D})$ is not necessarily tractable, we replace minimization of $\mathrm{KL}(q \mid\mid p)$ with maximization of the evidence lower bound (ELBO)

$$\mathrm{ELBO}(\phi) = \int q(\theta \mid \mathcal{D}, \phi) \log \frac{p(Y \mid X, \theta) p(\theta)}{q(\theta \mid \mathcal{D}, \phi)} d\theta$$

The *log* of the total probability (evidence) is then

$$\log p(D) = \mathrm{ELBO}(\phi) + \mathrm{KL}(q \mid\mid p).$$

The sum does not depend on ϕ, thus minimizing $\mathrm{KL}(q \mid\mid p)$ is the same as maximizing $\mathrm{ELBO}(q)$. Also, since $\mathrm{KL}(q \mid\mid p) \geq 0$, which follows from Jensen's inequality, we have $\log p(\mathcal{D}) \geq \mathrm{ELBO}(\phi)$. Thus, the evidence lower bound name. The resulting maximization problem $\mathrm{ELBO}(\phi) \rightarrow \max_\phi$ is solved using stochastic gradient descent.

To calculate the gradient, it is convenient to write the ELBO as

$$\mathrm{ELBO}(\phi) = \int q(\theta \mid \mathcal{D}, \phi) \log p(Y \mid X, \theta) d\theta - \int q(\theta \mid \mathcal{D}, \phi) \log \frac{q(\theta \mid \mathcal{D}, \phi)}{p(\theta)} d\theta$$

The gradient of the first term $\nabla_\phi \int q(\theta \mid \mathcal{D}, \phi) \log p(Y \mid X, \theta) d\theta = \nabla_\phi E_q \log p(Y \mid X, \theta)$ is not an expectation and thus cannot be calculated using Monte Carlo methods. The idea is to represent the gradient $\nabla_\phi E_q \log p(Y \mid X, \theta)$ as an expectation of some random variable, so that Monte Carlo techniques can be used to calculate it. There are two standard methods to do it. First, the log-derivative trick uses the following identity $\nabla_x f(x) = f(x) \nabla_x \log f(x)$ to obtain $\nabla_\phi E_q \log p(Y \mid \theta)$. Thus, if we select $q(\theta \mid \phi)$ so that it is easy to compute its derivative and generate samples from it, the gradient can be efficiently calculated using Monte Carlo techniques. Second, we can use the reparameterization trick by representing θ as a value of a deterministic function, $\theta = g(\epsilon, x, \phi)$, where $\epsilon \sim r(\epsilon)$ does not depend on ϕ. The derivative is given by

$$\begin{aligned} \nabla_\phi E_q \log p(Y \mid X, \theta) &= \int r(\epsilon) \nabla_\phi \log p(Y \mid X, g(\epsilon, x, \phi)) d\epsilon \\ &= E_\epsilon [\nabla_g \log p(Y \mid X, g(\epsilon, x, \phi)) \nabla_\phi g(\epsilon, x, \phi)]. \end{aligned}$$

The reparameterization is trivial when $q(\theta \mid \mathcal{D}, \phi) = \mathcal{N}(\theta \mid \mu(\mathcal{D}, \phi), \Sigma(D, \phi))$, and $\theta = \mu(\mathcal{D}, \phi) + \epsilon \Sigma(\mathcal{D}, \phi)$, $\epsilon \sim \mathcal{N}(0, I)$. Kingma and Welling (2013) propose using $\Sigma(\mathcal{D}, \phi) = I$ and representing $\mu(\mathcal{D}, \phi)$ and ϵ as outputs of a neural network (multilayer perceptron), the resulting approach was called a variational autoencoder. A generalized reparameterization has been proposed by Ruiz et al. (2016) and combines both log-derivative and reparameterization techniques by assuming that ϵ can depend on ϕ.

7 Summary

In this chapter we have introduced some of the theory of function approximation and out-of-sample estimation with neural networks when the observation points are i.i.d. Such a case is not suitable for times series data and shall be the subject of later chapters. We restricted our attention to feedforward neural networks in order to explore some of the theoretical arguments which help us reason scientifically about architecture design. We have seen that feedforward networks use hidden units, or perceptrons, to partition the input space into regions bounded with manifolds. In the case of ReLU activated units, each manifold is a hyperplane and the hidden units form a hyperplane arrangement. We have introduced various approaches to reason about the effect of the number of units in each layer in addition to reasoning about the effect of hidden layers. We also introduced various concepts and methods necessary for understanding and applying neural networks to i.i.d. data including

- Fat shattering, VC dimension, and the empirical risk measure (ERM) as the basis for characterizing the learnability of a class of MLPs;
- The construction of neural networks as splines and their pointwise approximation error bound;
- The reason for composing layers in deep learning;
- Stochastic gradient descent and back-propagation as techniques for training neural networks; and
- Imposing constraints on the network needed for approximating financial derivatives and other constrained optimization problems in finance.

8 Exercises

Exercise 4.1
Show that substituting

$$\nabla_{ij} I_k = \begin{cases} X_j, & i = k, \\ 0, & i \neq k, \end{cases}$$

into Eq. 4.47 gives

$$\nabla_{ij}\sigma_k \equiv \frac{\partial \sigma_k}{\partial w_{ij}} = \nabla_i \sigma_k X_j = \sigma_k(\delta_{ki} - \sigma_i)X_j.$$

Exercise 4.2
Show that substituting the derivative of the softmax function w.r.t. w_{ij} into Eq. 4.52 gives for the special case when the output is $Y_k = 1,\ k = i$, and $Y_k = 0,\ \forall k \neq i$:

$$\nabla_{ij}\mathcal{L}(W,b) := [\nabla_W \mathcal{L}(W,b)]_{ij} = \begin{cases} (\sigma_i - 1)X_j, & Y_i = 1, \\ 0, & Y_k = 0,\ \forall k \neq i. \end{cases}$$

Exercise 4.3
Consider feedforward neural networks constructed using the following two types of activation functions:

– Identity

$$Id(x) := x$$

– Step function (a.k.a. Heaviside function)

$$H(x) := \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

1. Consider a feedforward neural network with one input $x \in \mathbb{R}$, a single hidden layer with K units having step function activations, $H(x)$, and a single output with identity (a.k.a. linear) activation, $Id(x)$. The output can be written as

$$\hat{f}(x) = Id\left(b^{(2)} + \sum_{k=1}^{K} w_k^{(2)} H(b_k^{(1)} + w_k^{(1)} x)\right).$$

   Construct neural networks using these activation functions.

   a. Consider the step function

$$u(x;a) := yH(x-a) = \begin{cases} y, & \text{if } x \geq a, \\ 0, & \text{otherwise.} \end{cases}$$

   Construct a neural network with one input x and one hidden layer, whose response is $u(x;a)$. Draw the structure of the neural network, specify the activation function for each unit (either Id or H), and specify the values for all weights (in terms of a and y).

   b. Now consider the indicator function

$$\mathbf{1}_{[a,b)}(x) = \begin{cases} 1, & \text{if } x \in [a,b), \\ 0, & \text{otherwise.} \end{cases}$$

   Construct a neural network with one input x and one hidden layer, whose response is $y\mathbf{1}_{[a,b)}(x)$, for given real values y, a and b. Draw the structure of the neural network, specify the activation function for each unit (either Id or H), and specify the values for all weights (in terms of a, b and y).

Exercise 4.4
A neural network with a single hidden layer can provide an arbitrarily close approximation to any 1-dimensional bounded smooth function. This question will guide you through the proof. Let $f(x)$ be any function whose domain is $[C, D)$, for real values $C < D$. Suppose that the function is Lipschitz continuous, that is,

$$\forall x, x' \in [C, D),\ |f(x') - f(x)| \leq L|x' - x|,$$

for some constant $L \geq 0$. Use the building blocks constructed in the previous part to construct a neural network with one hidden layer that approximates this function within $\epsilon > 0$, that is, $\forall x \in [C, D),\ |f(x) - \hat{f}(x)| \leq \epsilon$, where $\hat{f}(x)$ is the output of your neural network given input x. Your network should use only the identity or the Heaviside activation functions. You need to specify the number K of hidden units, the activation function for each unit, and a formula for calculating each weight w_0, w_k, $w_0^{(k)}$, and $w_1^{(k)}$, for each $k \in \{1, \ldots, K\}$. These weights may be specified in terms of C, D, L, and ϵ, as well as the values of $f(x)$ evaluated at a finite number of x values of your choosing (you need to explicitly specify which x values you use). You do not need to explicitly write the $\hat{f}(x)$ function. Why does your network attain the given accuracy ϵ?

Exercise 4.5
Consider a shallow neural network regression model with n tanh activated units in the hidden layer and d outputs. The hidden-outer weight matrix $W_{ij}^{(2)} = \frac{1}{n}$ and the input-hidden weight matrix $W^{(1)} = 1$. The biases are zero. If the features, $X_1, \ldots, X_p$ are i.i.d. Gaussian random variables with mean $\mu = 0$, variance σ^2, show that

a. $\hat{Y} \in [-1, 1]$.
b. $\hat{Y}$ is independent of the number of hidden units, $n \geq 1$.
c. The expectation, $\mathbb{E}[\hat{Y}] = 0$, and the variance $\mathbb{V}[\hat{Y}] \leq 1$.

Exercise 4.6
Determine the VC dimension of the sum of indicator functions where $\Omega = [0, 1]$

$$F_k(x) = \{f : \Omega \to \{0, 1\},\ f(x) = \sum_{i=0}^{k} \mathbf{1}_{x \in [t_{2i}, t_{2i+1})}, 0 \leq t_0 < \cdots < t_{2k+1} \leq 1,\ k \geq 1\}.$$

Exercise 4.7
Show that a feedforward binary classifier with two Heaviside activated units shatters the data $\{0.25, 0.5, 0.75\}$.

Exercise 4.8
Compute the weight and bias updates of $W^{(2)}$ and $b^{(2)}$ given a shallow binary classifier (with one hidden layer) with unit weights, zero biases, and ReLU activation of two hidden units for the labeled observation $(x = 1, y = 1)$.

8.1 Programming Related Questions*

Exercise 4.9
Consider the following dataset (taken from Anscombe's quartet):

$$(x_1, y_1) = (10.0, 9.14), (x_2, y_2) = (8.0, 8.14), (x_3, y_3) = (13.0, 8.74),$$
$$(x_4, y_4) = (9.0, 8.77), (x_5, y_5) = (11.0, 9.26), (x_6, y_6) = (14.0, 8.10),$$
$$(x_7, y_7) = (6.0, 6.13), (x_8, y_8) = (4.0, 3.10), (x_9, y_9) = (12.0, 9.13),$$
$$(x_{10}, y_{10}) = (7.0, 7.26), (x_{11}, y_{11}) = (5.0, 4.74).$$

a. Use a neural network library of your choice to show that a feedforward network with one hidden layer consisting of one unit and a feedforward network with no hidden layers, each using only linear activation functions, do not outperform linear regression based on ordinary least squares (OLS).
b. Also demonstrate that a neural network with a hidden layer of three neurons using the tanh activation function and an output layer using the linear activation function captures the non-linearity and outperforms the linear regression.

Exercise 4.10
Review the Python notebook `deep_classifiers.ipynb`. This notebook uses Keras to build three simple feedforward networks applied to the half-moon problem: a logistic regression (with no hidden layer); a feedforward network with one hidden layer; and a feedforward architecture with two hidden layers. The half-moons problem is not linearly separable in the original coordinates. However you will observe—after plotting the fitted weights and biases—that a network with many hidden neurons gives a linearly separable representation of the classification problem in the coordinates of the output from the final hidden layer.

Complete the following questions in your own words.

a. Did we need more than one hidden layer to perfectly classify the half-moons dataset? If not, why might multiple hidden layers be useful for other datasets?
b. Why not use a very large number of neurons since it is clear that the classification accuracy improves with more degrees of freedom?
c. Repeat the plotting of the hyperplane, in Part 1b of the notebook, only without the ReLU function (i.e., activation="linear"). Describe qualitatively how the decision surface changes with increasing neurons. Why is a (non-linear) activation function needed? The use of figures to support your answer is expected.

Exercise 4.11
Using the EarlyStopping callback in Keras, modify the notebook `Deep_Classifiers.ipynb` to terminate training under the following stopping criterion $|L^{(k+1)} - L^{(k)}| \leq \delta$ with $\delta = 0.1$.

Exercise 4.12***
Consider a feedforward neural network with three inputs, two units in the first hidden layer, two units in the second hidden layer, and three units in the output layer. The activation function for hidden layer 1 is ReLU, for hidden layer 2 is sigmoid, and for the output layer is softmax.

The initial weights are given by the matrices

$$W^{(1)} = \begin{pmatrix} 0.1 & 0.3 & 0.7 \\ 0.9 & 0.4 & 0.4 \end{pmatrix}, W^{(2)} = \begin{pmatrix} 0.4 & 0.3 \\ 0.7 & 0.2 \end{pmatrix}, W^{(3)} = \begin{pmatrix} 0.5 & 0.6 \\ 0.6 & 0.7 \\ 0.3 & 0.2 \end{pmatrix},$$

and all the biases are unit vectors.

Assuming that the input $\begin{pmatrix} 0.1 & 0.7 & 0.3 \end{pmatrix}$ corresponds to the output $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$, manually compute the updated weights and biases after a single epoch (forward + backward pass), clearly stating all derivatives that you have used. You should use a learning rate of 1.

As a practical exercise, you should modify the implementation of a stochastic gradient descent routine in the back-propagation Python notebook.

Note that the notebook example corresponds to the example in Sect. 5, which uses sigmoid activated hidden layers only. Compare the weights and biases obtained by `TensorFlow` (or your ANN library of choice) with those obtained by your procedure after 200 epochs.

Appendix

Answers to Multiple Choice Questions

Question 1
Answer: 1, 2, 3, 4. All answers are found in the text.

Question 2
Answer: 1,2. A feedforward architecture is always convex w.r.t. each input variable if every activation function is convex and the weights are constrained to be either all positive or all negative. Simply using convex activation functions is not sufficient, since the composition of a convex function and the affine transformation of a convex function do not preserve the convexity. For example, if $\sigma(x) = x^2$, $w = -1$, and $b = 1$, then $\sigma(w\sigma(x) + b) = (-x^2 + 1)^2$ is not convex in x.

A feedforward architecture with positive weights is a monotonically increasing function of the input for any choice of monotonically increasing activation function.

The weights of a feedforward architecture need not be constrained for the output of a feedforward network to be bounded. For example, activating the output with a softmax function will bound the output. Only if the output is not activated, should the weights and bias in the final layer be bounded to ensure bounded output.

The bias terms in a network shift the output but also effect the derivatives of the output w.r.t. to the input when the layer is activated.

Question 3
Answer: 1,2,3,4. The training of a neural network involves minimizing a loss function w.r.t. the weights and biases over the training data. L_1 regularization is used during model selection to penalize models with too many parameters. The loss function is augmented with a Lagrange penalty for the number of weights. In deep learning, regularization can be applied to each layer of the network. Therefore each layer has an associated regularization parameter. Back-propagation uses the chain rule to update the weights of the network but is not guaranteed to convergence to a unique minimum. This is because the loss function is not convex w.r.t. the weights. Stochastic gradient descent is a type of optimization method which is implemented with back-propagation. There are variants of SGD, however, such as adding Nestov's momentum term, ADAM , or RMSProp.

Back-Propagation

Let us consider a feedforward architecture with an input layer, $L-1$ hidden layers, and one output layer, with K units in the output layer for classification of K categories. As a result, we have L sets of weights and biases $(W^{(\ell)}, \mathbf{b}^{(\ell)})$ for $\ell = 1, \ldots, L$, corresponding to the layer inputs $Z^{(\ell-1)}$ and outputs $Z^{(\ell)}$ for $\ell = 1, \ldots, L$. Recall that each layer is an activation of a semi-affine transformation, $I^{(\ell)}(Z^{(\ell-1)}) := W^{(L)}Z^{(\ell-1)} + b^{(L)}$. The corresponding activation functions are denoted as $\sigma^{(\ell)}$. The activation function for the output layer is a softmax function, $\sigma_s(x)$.

Here we use the cross-entropy as the loss function, which is defined as

$$\mathcal{L} := -\sum_{k=1}^{K} Y_k \log \hat{Y}_k.$$

The relationship between the layers, for $\ell \in \{1, \ldots, L\}$ are

$$\begin{aligned}
\hat{Y}(X) = Z^{(L)} &= \sigma_s(I^{(L)}) \in [0,1]^K, \\
Z^{(\ell)} &= \sigma^{(\ell)}\left(I^{(\ell)}\right),\ \ell = 1, \ldots, L-1, \\
Z^{(0)} &= X.
\end{aligned}$$

The update rules for the weights and biases are

$$\begin{aligned}
\Delta W^{(\ell)} &= -\gamma \nabla_{W^{(\ell)}} \mathcal{L}, \\
\Delta \mathbf{b}^{(\ell)} &= -\gamma \nabla_{\mathbf{b}^{(\ell)}} \mathcal{L}.
\end{aligned}$$

We now begin the back-propagation, tracking the intermediate calculations carefully using Einstein summation notation.

For the gradient of $\mathcal{L}$ w.r.t. $W^{(L)}$ we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{ij}^{(L)}} &= \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial Z_k^{(L)}} \frac{\partial Z_k^{(L)}}{\partial w_{ij}^{(L)}} \\ &= \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial Z_k^{(L)}} \sum_{m=1}^{K} \frac{\partial Z_k^{(L)}}{\partial I_m^{(L)}} \frac{\partial I_m^{(L)}}{\partial w_{ij}^{(L)}}\end{aligned}$$

But

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial Z_k^{(L)}} &= -\frac{Y_k}{Z_k^{(L)}} \\ \frac{\partial Z_k^{(L)}}{\partial I_m^{(L)}} &= \frac{\partial}{\partial I_m^{(L)}} [\sigma(I^{(L)})]_k \\ &= \frac{\partial}{\partial I_m^{(L)}} \frac{\exp[I_k^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} \\ &= \begin{cases} -\frac{\exp[I_k^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} \frac{\exp[I_m^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} & \text{if } k \neq m \\ \frac{\exp[I_k^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} - \frac{\exp[I_k^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} \frac{\exp[I_m^{(L)}]}{\sum_{n=1}^{K} \exp[I_n^{(L)}]} & \text{otherwise} \end{cases} \\ &= \begin{cases} -\sigma_k \sigma_m & \text{if } k \neq m \\ \sigma_k(1-\sigma_m) & \text{otherwise} \end{cases} \\ &= \sigma_k(\delta_{km} - \sigma_m) \quad \text{where } \delta_{km} \text{ is the Kronecker's Delta} \\ \frac{\partial I_m^{(L)}}{\partial w_{ij}^{(L)}} &= \delta_{mi} Z_j^{(L-1)} \\ \implies \frac{\partial \mathcal{L}}{\partial w_{ij}^{(L)}} &= -\sum_{k=1}^{K} \frac{Y_k}{Z_k^{(L)}} \sum_{m=1}^{K} Z_m^{(L)}(\delta_{km} - Z_m^{(L)}) \delta_{mi} Z_j^{(L-1)} \\ &= -Z_j^{(L-1)} \sum_{k=1}^{K} Y_k(\delta_{ki} - Z_i^{(L)}) \\ &= Z_j^{(L-1)}(Z_i^{(L)} - Y_i),\end{aligned}$$

where we have used the fact that $\sum_{k=1}^{K} Y_k = 1$ in the last equality. Similarly for $\mathbf{b}^{(L)}$, we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial b_i^{(L)}} &= \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial Z_k^{(L)}} \sum_{m=1}^{K} \frac{\partial Z_k^{(L)}}{\partial I_m^{(L)}} \frac{\partial I_m^{(L)}}{\partial b_i^{(L)}} \\ &= Z_i^{(L)} - Y_i\end{aligned}$$

It follows that

$$\begin{aligned}\nabla_{\mathbf{b}^{(L)}} \mathcal{L} &= Z^{(L)} - Y \\ \nabla_{W^{(L)}} \mathcal{L} &= \nabla_{\mathbf{b}^{(L)}} \mathcal{L} \otimes Z^{(L-1)},\end{aligned}$$

where $\otimes$ denotes the outer product.

Now for the gradient of $\mathcal{L}$ w.r.t. $W^{(L-1)}$ we have

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{ij}^{(L-1)}} &= \sum_{k=1}^{K} \frac{\partial L}{\partial Z_k^{(L)}} \frac{\partial Z_k^{(L)}}{\partial w_{ij}^{(L-1)}} \\ &= \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\partial Z_k^{(L)}} \sum_{m=1}^{K} \frac{\partial Z_k^{(L)}}{\partial I_m^{(L)}} \sum_{n=1}^{n^{(L-1)}} \frac{\partial I_m^{(L)}}{\partial Z_n^{(L-1)}} \sum_{p=1}^{n^{(L-1)}} \frac{\partial Z_n^{(L-1)}}{\partial I_p^{(L-1)}} \frac{\partial I_p^{(L-1)}}{\partial w_{ij}^{(L-1)}}\end{aligned}$$

If we assume that $\sigma^{(\ell)}(x) = \text{sigmoid}(x),\ \ell \in \{1, \ldots, L-1\}$, then

$$\begin{aligned}\frac{\partial I_m^{(L)}}{\partial Z_n^{(L-1)}} &= w_{mn}^{(L)} \\ \frac{\partial Z_n^{(L-1)}}{\partial I_p^{(L-1)}} &= \frac{\partial}{\partial I_p^{(L-1)}} \left(\frac{1}{1 + \exp(-I_n^{(L-1)})} \right) \\ &= \frac{1}{1 + \exp(-I_n^{(L-1)})} \frac{\exp(-I_n^{(L-1)})}{1 + \exp(-I_n^{(L-1)})} \delta_{np} \\ &= Z_n^{(L-1)}(1 - Z_n^{(L-1)})\, \delta_{np} = \sigma_n^{(L-1)}(1 - \sigma_n^{(L-1)}) \delta_{np} \\ \frac{\partial I_p^{(L-1)}}{\partial w_{ij}^{(L-1)}} &= \delta_{pi} Z_j^{(L-2)}\end{aligned}$$

$$\implies \frac{\partial L}{\partial w_{ij}^{(L)}} = -\sum_{k=1}^{K} \frac{Y_k}{Z_k^{(L)}} \sum_{m=1}^{K} Z_k^{(L)} (\delta_{km} - Z_m^{(L)})$$

$$\sum_{n=1}^{n^{(L-1)}} w_{mn}^{(L)} \sum_{p=1}^{n^{(L-1)}} Z_n^{(L-1)}(1 - Z_n^{(L-1)})\, \delta_{np}\delta_{pi}\, Z_j^{(L-2)}$$

$$= -\sum_{k=1}^{K} Y_k \sum_{m=1}^{K} (\delta_{km} - Z_m^{(L)}) \sum_{n=1}^{n^{(L-1)}} w_{mn}^{(L)} Z_n^{(L-1)}(1 - Z_n^{(L-1)})\, \delta_{ni}\, Z_j^{(L-2)}$$

$$= -\sum_{k=1}^{K} Y_k \sum_{m=1}^{K} (\delta_{km} - Z_m^{(L)}) w_{mi}^{(L)} Z_i^{(L-2)}(1 - Z_i^{(L-1)}) Z_j^{(L-2)}$$

$$= -Z_j^{(L-2)} Z_i^{(L-1)}(1 - Z_i^{(L-1)}) \sum_{m=1}^{K} w_{mi}^{(L)} \sum_{k=1}^{K} (\delta_{km} Y_k - Z_m^{(L)} Y_k)$$

$$= Z_j^{(L-2)} Z_i^{(L-1)}(1 - Z_i^{(L-1)}) \sum_{m=1}^{K} w_{mi}^{(L)} (Z_m^{(L)} - Y_m)$$

$$= Z_j^{(L-2)} Z_i^{(L-1)}(1 - Z_i^{(L-1)})(Z^{(L)} - Y)^T \mathbf{w}_{,i}^{(L)}$$

Similarly we have

$$\frac{\partial \mathcal{L}}{\partial b_i^{(L-1)}} = Z_i^{(L-1)}(1 - Z_i^{(L-1)})(Z^{(L)} - Y)^T \mathbf{w}_{,i}^{(L)}.$$

It follows that we can define the following recursion relation for the loss gradient:

$$\nabla_{b^{(L-1)}} \mathcal{L} = Z^{(L-1)} \circ (\mathbf{1} - Z^{(L-1)}) \circ (W^{(L)^T} \nabla_{b^{(L)}} \mathcal{L})$$

$$\begin{aligned} \nabla_{W^{(L-1)}} \mathcal{L} &= \nabla_{b^{(L-1)}} \mathcal{L} \otimes Z^{(L-2)} \\ &= Z^{(L-1)} \circ (\mathbf{1} - Z^{(L-1)}) \circ (W^{(L)^T} \nabla_{W^{(L)}} \mathcal{L}), \end{aligned}$$

where $\circ$ denotes the Hadamard product (element-wise multiplication). This recursion relation can be generalized for all layers and choice of activation functions. To see this, let the back-propagation error $\delta^{(\ell)} := \nabla_{b^{(\ell)}} \mathcal{L}$, and since

$$\begin{aligned} \left[\frac{\partial \sigma^{(\ell)}}{\partial I^{(\ell)}}\right]_{ij} &= \frac{\partial \sigma_i^{(\ell)}}{\partial I_j^{(\ell)}} \\ &= \sigma_i^{(\ell)}(1 - \sigma_i^{(\ell)})\delta_{ij} \end{aligned}$$

or equivalently in matrix–vector form

$$\nabla_{I^{(\ell)}}\sigma^{(\ell)} = \text{diag}(\sigma^{(\ell)} \circ (\mathbf{1} - \sigma^{(\ell)})),$$

we can write, in general, for any choice of activation function for the hidden layer,

$$\delta^{(\ell)} = \nabla_{I^{(\ell)}}\sigma^{(\ell)}(W^{(\ell+1)})^T\delta^{(\ell+1)},$$

and

$$\nabla_{W^{(\ell)}}\mathcal{L} = \delta^{(\ell)} \otimes Z^{(\ell-1)}.$$

Proof of Theorem 4.2

Using the same deep structure shown in Fig. 4.9, Liang and Srikant (2016) find the binary expansion sequence $\{x_0, \ldots, x_n\}$. In this step, they used n binary steps units in total. Then they rewrite $g_{m+1}(\sum_{i=0}^{n} \frac{x_i}{2^n})$,

$$\begin{aligned} g_{m+1}\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right) &= \sum_{j=0}^{n}\left[x_j \cdot \frac{1}{2^j} g_m\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right)\right] \\ &= \sum_{j=0}^{n}\max\left[2(x_j - 1) + \frac{1}{2^j} g_m\left(\sum_{i=0}^{n}\frac{x_i}{2^i}\right), 0\right]. \end{aligned} \tag{4.57}$$

Clearly Eq. 4.57 defines iterations between the outputs of neighboring layers. Defining the output of the multilayer neural network as $\hat{f}(x) = \sum_{i=0}^{p} a_i g_i\left(\sum_{j=0}^{n}\frac{x_j}{2^j}\right)$. For this multilayer network, the approximation error is

$$\begin{aligned} |f(x) - \hat{f}(x)| &= \left|\sum_{i=0}^{p} a_i g_i\left(\sum_{j=0}^{n}\frac{x_j}{2^j}\right) - \sum_{i=0}^{p} a_i x^i\right| \\ &\le \sum_{i=0}^{p}\left[|a_i| \cdot \left|g_i\left(\sum_{j=0}^{n}\frac{x_j}{2^j}\right) - x^i\right|\right] \le \frac{p}{2^{n-1}}. \end{aligned}$$

This indicates, to achieve ε-approximation error, one should choose $n = \lceil \log\frac{p}{\varepsilon} \rceil + 1$. Besides, since $O(n + p)$ layers with $O(n)$ binary step units and $O(pn)$ ReLU units are used in total, this multilayer neural network thus has $O\left(p + \log\frac{p}{\varepsilon}\right)$ layers, $O\left(\log\frac{p}{\varepsilon}\right)$ binary step units, and $O\left(p\log\frac{p}{\varepsilon}\right)$ ReLU units.

Table 4.2 Definitions of the functions $f(x)$ and $g(x)$

$f(x) := \max(x - \frac{1}{4}, 0)$,	$g(x) := \max(x - \frac{1}{2}, 0)$
$\mathrm{cI}_f = \{[0, \frac{1}{4}], (\frac{1}{4}, 1]\}$,	$\mathrm{cI}_g = \{[0, \frac{1}{2}], (\frac{1}{2}, 1]\}$.

Proof of Lemmas from Telgarsky (2016)

Proof (Proof of 4.1) Let cI_f denote the partition of $\mathbb{R}$ corresponding to f, and cI_g denote the partition of $\mathbb{R}$ corresponding to g.

First consider $f + g$, and moreover any intervals $U_f \in \mathrm{cI}_f$ and $U_g \in \mathrm{cI}_g$. Necessarily, $f + g$ has a single slope along $U_f \cap U_g$. Consequently, $f + g$ is $|\mathrm{cI}|$-sawtooth, where cI is the set of all intersections of intervals from cI_f and cI_g, meaning $\mathrm{cI} := \{U_f \cap U_g : U_f \in \mathrm{cI}_f, U_g \in \mathrm{cI}_g\}$. By sorting the left endpoints of elements of cI_f and cI_g, it follows that $|\mathrm{cI}| \leq k + l$ (the other intersections are empty).

For example, consider the example in Fig. 4.11 with partitions given in Table 4.2. The set of all intersections of intervals from cI_f and cI_g contains 3 elements:

$$\mathrm{cI} = \{[0, \frac{1}{4}] \cap [0, \frac{1}{2}], (\frac{1}{4}, 1] \cap [0, \frac{1}{2}], (\frac{1}{4}, 1] \cap (\frac{1}{2}, 1]\} \tag{4.58}$$

Now consider $f \circ g$, and in particular consider the image $f(g(U_g))$ for some interval $U_g \in \mathrm{cI}_g$. g is affine with a single slope along U_g; therefore, f is being considered along a single unbroken interval $g(U_g)$. However, nothing prevents $g(U_g)$ from hitting all the elements of cI_f; since U_g was arbitrary, it holds that $f \circ g$ is $(|\mathrm{cI}_f| \cdot |\mathrm{cI}_g|)$-sawtooth. □

Proof Recall the notation $\tilde{f}(x) := [f(x) \geq 1/2]$, whereby $\mathcal{E}(f) := \frac{1}{n} \sum_i [y_i \neq \tilde{f}(x_i)]$. Since f is piecewise monotonic with a corresponding partition $\mathbb{R}$ having at most t pieces, then f has at most $2t - 1$ crossings of 1/2: at most one within each interval of the partition, and at most 1 at the right endpoint of all but the last interval. Consequently, $\tilde{f}$ is piecewise *constant*, where the corresponding partition of $\mathbb{R}$ is into at most $2t$ intervals. This means n points with alternating labels must land in $2t$ buckets, thus the total number of points landing in buckets with at least three points is at least $n - 4t$. □

Python Notebooks

The notebooks provided in the accompanying source code repository are designed to gain insight in toy classification datasets. They provide examples of deep feedforward classification, back-propagation, and Bayesian network classifiers. Further details of the notebooks are included in the `README.md` file.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). Tensor flow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16 (pp. 265–283).

Adams, R., Wallach, H., & Ghahramani, Z. (2010). Learning the structure of deep sparse graphical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (pp. 1–8).

Andrews, D. (1989). A unified theory of estimation and inference for nonlinear dynamic models a.r. gallant and h. white. *Econometric Theory, 5*(01), 166–171.

Baillie, R. T., & Kapetanios, G. (2007). Testing for neglected nonlinearity in long-memory models. *Journal of Business & Economic Statistics, 25*(4), 447–461.

Barber, D., & Bishop, C. M. (1998). Ensemble learning in Bayesian neural networks. *Neural Networks and Machine Learning, 168*, 215–238.

Bartlett, P., Harvey, N., Liaw, C., & Mehrabian, A. (2017a). Nearly-tight VC-dimension bounds for piecewise linear neural networks. *CoRR, abs/1703.02930*.

Bartlett, P., Harvey, N., Liaw, C., & Mehrabian, A. (2017b). Nearly-tight VC-dimension bounds for piecewise linear neural networks. *CoRR, abs/1703.02930*.

Bengio, Y., Roux, N. L., Vincent, P., Delalleau, O., & Marcotte, P. (2006). Convex neural networks. In Y. Weiss, Schölkopf, B., & Platt, J. C. (Eds.), *Advances in neural information processing systems 18* (pp. 123–130). MIT Press.

Bishop, C. M. (2006). *Pattern recognition and machine learning (information science and statistics)*. Berlin, Heidelberg: Springer-Verlag.

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015a, May). Weight uncertainty in neural networks. *arXiv:1505.05424 [cs, stat]*.

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015b). Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*.

Chataigner, Crepe, & Dixon. (2020). *Deep local volatility*.

Chen, J., Flood, M. D., & Sowers, R. B. (2017). Measuring the unmeasurable: an application of uncertainty quantification to treasury bond portfolios. *Quantitative Finance, 17*(10), 1491–1507.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems* (pp. 1223–1231).

Dixon, M., Klabjan, D., & Bang, J. H. (2016). Classification-based financial markets prediction using deep neural networks. *CoRR, abs/1603.08604*.

Feng, G., He, J., & Polson, N. G. (2018, Apr). Deep learning for predicting asset returns. *arXiv e-prints*, arXiv:1804.09314.

Frey, B. J., & Hinton, G. E. (1999). Variational learning in nonlinear Gaussian belief networks. *Neural Computation, 11*(1), 193–213.

Gal, Y. (2015). A theoretically grounded application of dropout in recurrent neural networks. *arXiv:1512.05287*.

Gal, Y. (2016). *Uncertainty in deep learning*. Ph.D. thesis, University of Cambridge.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *international Conference on Machine Learning* (pp. 1050–1059).

Gallant, A., & White, H. (1988, July). There exists a neural network that does not make avoidable mistakes. In *IEEE 1988 International Conference on Neural Networks* (vol.1 ,pp. 657–664).

Graves, A. (2011). Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems* (pp. 2348–2356).

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference and prediction*. Springer.

Heaton, J. B., Polson, N. G., & Witte, J. H. (2017). Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry, 33*(1), 3–12.

Hernández-Lobato, J. M., & Adams, R. (2015). Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *International Conference on Machine Learning* (pp. 1861–1869).

Hinton, G. E., & Sejnowski, T. J. (1983). Optimal perceptual inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 448–453). IEEE New York.

Hinton, G. E., & Van Camp, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory* (pp. 5–13). ACM.

Hornik, K., Stinchcombe, M., & White, H. (1989, July). Multilayer feedforward networks are universal approximators. *Neural Netw., 2*(5), 359–366.

Horvath, B., Muguruza, A., & Tomas, M. (2019, Jan). *Deep learning volatility*. *arXiv e-prints*, arXiv:1901.09647.

Hutchinson, J. M., Lo, A. W., & Poggio, T. (1994). A nonparametric approach to pricing and hedging derivative securities via learning networks. *The Journal of Finance, 49*(3), 851–889.

Kingma, D. P., & Welling, M. (2013). Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*.

Kuan, C.-M., & White, H. (1994). Artificial neural networks: an econometric perspective. *Econometric Reviews, 13*(1), 1–91.

Lawrence, N. (2005). Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *Journal of Machine Learning Research, 6*(Nov), 1783–1816.

Liang, S., & Srikant, R. (2016). Why deep neural networks? *CoRR abs/1610.04161*.

Lo, A. (1994). Neural networks and other nonparametric techniques in economics and finance. In *AIMR Conference Proceedings*, Number 9.

MacKay, D. J. (1992a). A practical Bayesian framework for backpropagation networks. *Neural Computation, 4*(3), 448–472.

MacKay, D. J. C. (1992b, May). A practical Bayesian framework for backpropagation networks. *Neural Computation, 4*(3), 448–472.

Martin, C. H., & Mahoney, M. W. (2018). Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *CoRR abs/1810.01075*.

Mhaskar, H., Liao, Q., & Poggio, T. A. (2016). Learning real and Boolean functions: When is deep better than shallow. *CoRR abs/1603.00988*.

Mnih, A., & Gregor, K. (2014). Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030*.

Montúfar, G., Pascanu, R., Cho, K., & Bengio, Y. (2014, Feb). On the number of linear regions of deep neural networks. *arXiv e-prints*, arXiv:1402.1869.

Mullainathan, S., & Spiess, J. (2017). Machine learning: An applied econometric approach. *Journal of Economic Perspectives, 31*(2), 87–106.

Neal, R. M. (1990). *Learning stochastic feedforward networks*, Vol. 64. Technical report, Department of Computer Science, University of Toronto.

Neal, R. M. (1992). *Bayesian training of backpropagation networks by the hybrid Monte Carlo method*. Technical report, CRG-TR-92-1, Dept. of Computer Science, University of Toronto.

Neal, R. M. (2012). *Bayesian learning for neural networks*, Vol. 118. Springer Science & Business Media. bibtex: aneal2012bayesian.

Nesterov, Y. (2013). *Introductory lectures on convex optimization: A basic course*, Volume 87. Springer Science & Business Media.

Poggio, T. (2016). Deep learning: mathematics and neuroscience. *A sponsored supplement to science brain-inspired intelligent robotics: The intersection of robotics and neuroscience*, pp. 9–12.

Polson, N., & Rockova, V. (2018, Mar). Posterior concentration for sparse deep learning. *arXiv e-prints*, arXiv:1803.09138.

Polson, N. G., Willard, B. T., & Heidari, M. (2015). A statistical theory of deep learning via proximal splitting. *arXiv:1509.06061*.

Racine, J. (2001). On the nonlinear predictability of stock returns using financial and economic variables. *Journal of Business & Economic Statistics, 19*(3), 380–382.

Rezende, D. J., Mohamed, S., & Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082*.

Ruiz, F. R., Aueb, M. T. R., & Blei, D. (2016). The generalized reparameterization gradient. In *Advances in Neural Information Processing Systems* (pp. 460–468).

Salakhutdinov, R. (2008). *Learning and evaluating Boltzmann machines*. Tech. Rep., Technical Report UTML TR 2008-002, Department of Computer Science, University of Toronto.

Salakhutdinov, R., & Hinton, G. (2009). Deep Boltzmann machines. In *Artificial Intelligence and Statistics* (pp. 448–455).

Saul, L. K., Jaakkola, T., & Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *Journal of Artificial Intelligence Research, 4*, 61–76.

Sirignano, J., Sadhwani, A., & Giesecke, K. (2016, July). Deep learning for mortgage risk. *ArXiv e-prints*.

Smolensky, P. (1986). *Parallel distributed processing: explorations in the microstructure of cognition* (Vol. 1. pp. 194–281). Cambridge, MA, USA: MIT Press.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research, 15*(1), 1929–1958.

Swanson, N. R., & White, H. (1995). A model-selection approach to assessing the information in the term structure using linear models and artificial neural networks. *Journal of Business & Economic Statistics, 13*(3), 265–275.

Telgarsky, M. (2016). Benefits of depth in neural networks. *CoRR abs/1602.04485*.

Tieleman, T. (2008). Training restricted Boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th International Conference on Machine Learning* (pp. 1064–1071). ACM.

Tishby, N., & Zaslavsky, N. (2015). Deep learning and the information bottleneck principle. *CoRR abs/1503.02406*.

Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., & Blei, D. M. (2017, January). Deep probabilistic programming. *arXiv:1701.03757 [cs, stat]*.

Vapnik, V. N. (1998). *Statistical learning theory*. Wiley-Interscience.

Welling, M., Rosen-Zvi, M., & Hinton, G. E. (2005). Exponential family harmoniums with an application to information retrieval. In *Advances in Neural Information Processing Systems* (pp. 1481–1488).

Williams, C. K. (1997). Computing with infinite networks. In *Advances in Neural Information Processing systems* (pp. 295–301).

Chapter 5
Interpretability

This chapter presents a method for interpreting neural networks which imposes minimal restrictions on the neural network design. The chapter demonstrates techniques for interpreting a feedforward network, including how to rank the importance of the features. An example demonstrating how to apply interpretability analysis to deep learning models for factor modeling is also presented.

1 Introduction

Once the neural network has been trained, a number of important issues surface around how to interpret the model parameters. This aspect is a prominent issue for practitioners in deciding whether to use neural networks in favor of other machine learning and statistical methods for estimating factor realizations, sometimes even if the latter's predictive accuracy is inferior.

In this section, we shall introduce a method for interpreting multilayer perceptrons which imposes minimal restrictions on the neural network design.

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Apply techniques for interpreting a feedforward network, including how to rank the importance of the features.
- Learn how to apply interpretability analysis to deep learning models for factor modeling.

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_5

2 Background on Interpretability

There are numerous techniques for interpreting machine learning methods which treat the model as a black-box. A good example are Partial Dependence Plots (PDPs) as described by Greenwell et al. (2018). Other approaches also exist in the literature. Garson (1991) partitions hidden-output connection weights into components associated with each input neuron using absolute values of connection weights. Olden and Jackson (2002) determine the relative importance, $[R]_{ij}$, of the ith output to the jth predictor variable of the model as a function of the weights, according to a simple linear expression.

We seek to understand the limitations on the choice of activation functions and understand the effect of increasing layers and numbers of neurons on probabilistic interpretability. For example, under standard Gaussian i.i.d. data, how robust are the model's estimate of the importance of each input variable with variable number of neurons?

2.1 Sensitivities

We shall therefore turn to a "white-box" technique for determining the importance of the input variables. This approach generalizes Dimopoulos et al. (1995) to a deep neural network with interaction terms. Moreover, the method is directly consistent with how coefficients are interpreted in linear regression—they are *model* sensitivities. Model sensitivities are the change of the fitted model output w.r.t. input.

As a control, we shall use this property to empirically evaluate how reliably neural networks, even deep networks, learn data from a linear model.

Such an approach is appealing to practitioners who are evaluating the comparative performance of linear regression with neural networks and need the assurance that a neural network model is at least able to reproduce and match the coefficients on a linear dataset.

We also offset the common misconception that the activation functions must be deactivated for a neural network model to produce a linear output. Under linear data, any non-linear statistical model should be able to reproduce a statistical linear model under some choice of parameter values. Irrespective of whether data is linear or non-linear in practice - the best control experiment for comparing a neural network estimator with an OLS estimator is to simulate data under a linear regression model. In this scenario, the correct model coefficients are known and the error in the coefficient estimator can be studied.

To evaluate fitted model sensitivities analytically, we require that the function $\hat{Y} = f(X)$ is continuous and differentiable everywhere. Furthermore, for stability of

the interpretation, we shall require that $f(x)$ is Lipschitz continuous.[1] That is, there is a positive real constant K s.t. $\forall x_1, x_2 \in \mathbb{R}^p$, $|F(x_1) - F(x_2)| \leq K|x_1 - x_2|$. Such a constraint is necessary for the first derivative to be bounded and hence amenable to the derivatives, w.r.t. to the inputs, providing interpretability.

Fortunately, provided that the weights and biases are finite, each semi-affine function is Lipschitz continuous everywhere. For example, the function $\tanh(x)$ is continuously differentiable and its derivative $1 - \tanh^2(x)$ is globally bounded. With finite weights, the composition of $tanh(x)$ with an affine function is also Lipschitz. Clearly $\text{ReLU}(x) := \max(\cdot, 0)$ is not continuously differentiable and one cannot use the approach described here. Note that for the following examples, we are indifferent to the choice of homoscedastic or heteroscedastic error, since the model sensitivities are independent of the error.

3 Explanatory Power of Neural Networks

In a linear regression model

$$\hat{Y} = F_{\boldsymbol{\beta}}(X) := \beta_0 + \beta_1 X_1 + \cdots + \beta_K X_K, \tag{5.1}$$

the model sensitivities are

$$\partial_{X_i} \hat{Y} = \beta_i. \tag{5.2}$$

In a feedforward neural network, we can use the chain rule to obtain the model sensitivities

$$\partial_{X_i} \hat{Y} = \partial_{X_i} F_{W,b}(X) = \partial_{X_i} \sigma^{(L)}_{W^{(L)},b^{(L)}} \circ \cdots \circ \sigma^{(1)}_{W^{(1)},b^{(1)}}(X). \tag{5.3}$$

For example, with one hidden layer, $\sigma(x) := \tanh(x)$ and $\sigma^{(1)}_{W^{(1)},b^{(1)}}(X) := \sigma(I^{(1)}) := \sigma(W^{(1)}X + b^{(1)})$:

$$\partial_{X_j} \hat{Y} = \sum_i \mathbf{w}^{(2)}_{,i} (1 - \sigma^2(I^{(1)}_i)) w^{(1)}_{ij} \quad \text{where } \partial_x \sigma(x) = (1 - \sigma^2(x)). \tag{5.4}$$

In matrix form, with general σ, the Jacobian[2] of σ w.r.t X is $J = D(I^{(1)})W^{(1)}$ of σ,

$$\partial_X \hat{Y} = W^{(2)} J(I^{(1)}) = W^{(2)} D(J^{(1)}) W^{(1)}, \tag{5.5}$$

[1] If Lipschitz continuity is not imposed, then a small change in one of the input values could result in an undesirable large variation in the derivative.

[2] When σ is an identity function, the Jacobian $J(I^{(1)}) = W^{(1)}$.

where $D_{ii}(I) = \sigma'(I_i),\ D_{ij} = 0,\ i \neq j$ is a diagonal matrix. Bounds on the sensitivities are given by the product of the weight matrices

$$\min(W^{(2)}W^{(1)}, 0) \leq \partial_X \hat{Y} \leq \max(W^{(2)}W^{(1)}, 0). \tag{5.6}$$

3.1 Multiple Hidden Layers

The model sensitivities can be readily generalized to an L layer deep network by evaluating the Jacobian matrix:

$$\partial_X \hat{Y} = W^{(L)} J(I^{(L-1)}) = W^{(L)} D(I^{(L-1)}) W^{(L-1)} \dots D(I^{(1)}) W^{(1)}. \tag{5.7}$$

3.2 Example: Step Test

To illustrate our interpretability approach, we shall consider a simple example. The model is trained to the following data generation process where the coefficients of the features are stepped and the error, here, is i.i.d. uniform:

$$\hat{Y} = \sum_{i=1}^{10} i X_i,\ X_i \sim \mathcal{U}(0, 1). \tag{5.8}$$

Figure 5.1 shows the ranked importance of the input variables in a neural network with one hidden layer. Our interpretability method is compared with well-known black-box interpretability methods such as Garson's algorithm (Garson 1991) and Olden's algorithm (Olden and Jackson 2002). Our approach is the only technique to interpret the fitted neural network which is consistent with how a linear regression model would interpret the input variables.

4 Interaction Effects

The previous example is too simplistic to illustrate another important property of our interpretability method, namely the ability to capture pairwise interaction terms. The pairwise interaction effects are readily available by evaluating the elements of the Hessian matrix. For example, with one hidden layer, the Hessian takes the form:

$$\partial^2_{X_i X_j} \hat{Y} = W^{(2)} diag(W_i^{(1)}) D'(I^{(1)}) W_j^{(1)}, \tag{5.9}$$

where it is assumed that the activation function is at least twice differentiable everywhere, e.g. $tanh(x)$.

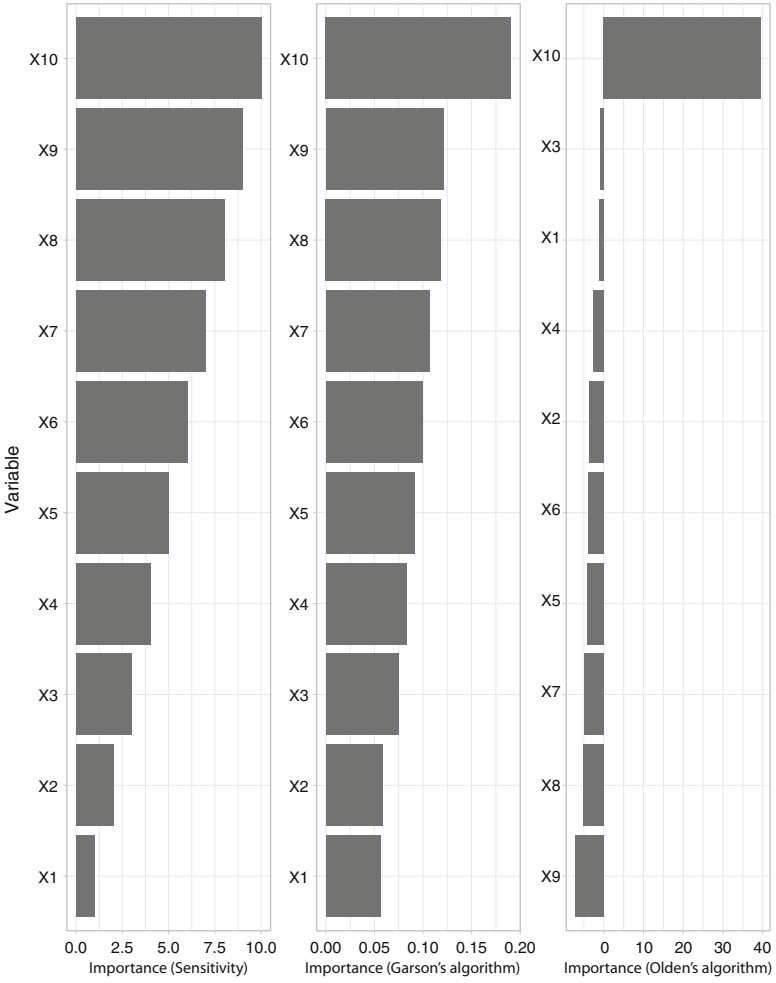


Fig. 5.1 Step test: This figure shows the ranked importance of the input variables in a neural network with one hidden layer. (left) Our sensitivity based approach for input interpretability. (Center) Garson's algorithm and (Right) Olden's algorithm. Our approach is the only technique to interpret the fitted neural network which is consistent with how a linear regression model would interpret the input variables

4.1 Example: Friedman Data

To illustrate our input variable and interaction effect ranking approach, we will use a classical nonlinear benchmark regression problem. The input space consists of ten i.i.d. uniform $\mathcal{U}(0, 1)$ random variables; however, only five out of these ten actually appear in the true model. The response is related to the inputs according to the formula

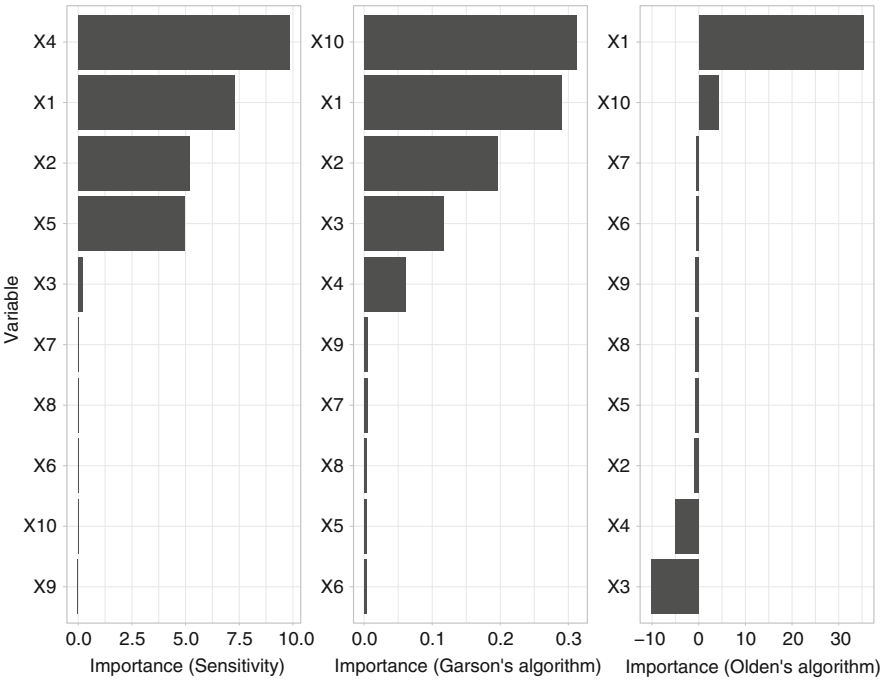


Fig. 5.2 Friedman test: Ranked model sensitivities of the fitted neural network to the input. (left) Our sensitivity based approach for input interpretability. (Center) Garson's algorithm and (Right) Olden's algorithm

$$Y = 10 \sin(\pi X_1 X_2) + 20 (X_3 - 0.5)^2 + 10 X_4 + 5 X_5 + \epsilon,$$

using white noise error, $\epsilon \sim \mathcal{N}(0, \sigma^2)$. We fit a NN with one hidden layer containing eight units and a weight decay of 0.01 (these parameters were chosen using 5-fold cross-validation) to 500 observations simulated from the above model with $\sigma = 1$. The cross-validated R^2 value was 0.94.

Figures 5.2 and 5.3, respectively, compare the ranked model sensitivities and ranked interaction terms of the fitted neural network with Garson's and Olden's algorithm.

5 Bounds on the Variance of the Jacobian

General results on the bound of the variance of the Jacobian for any activation function are difficult to derive. However, we derive the following result for a $ReLU$ activated single-layer feedforward network. In matrix form, with $\sigma(x) = \max(x, 0)$, the Jacobian, J, can be written as a linear combination of Heaviside functions:

$$J := J(X) = \partial_X \hat{Y}(X) = W^{(2)} J(I^{(1)}) = W^{(2)} H(W^{(1)} X + b^{(1)}) W^{(1)}, \qquad (5.10)$$

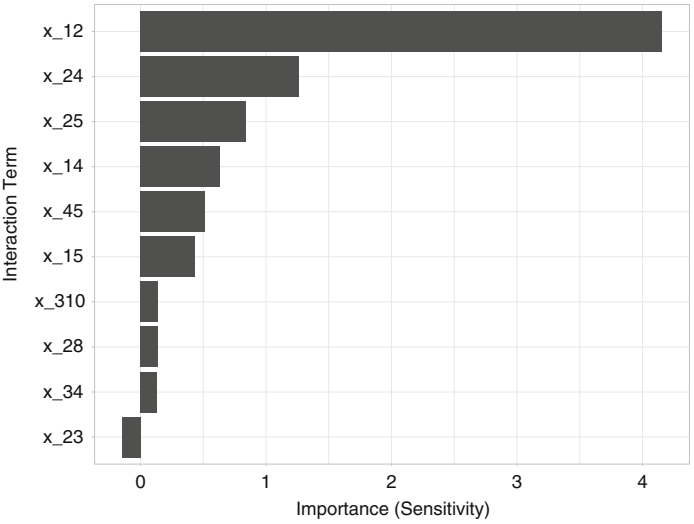


Fig. 5.3 Friedman test: Ranked pairwise interaction terms in the fitted neural network to the input. (Left) Our sensitivity based approach for ranking interaction terms. (Center) Garson's algorithm and (Right) Olden's algorithm

where $H_{ii}(Z) = H(I_i^{(1)}) = \mathbb{1}_{\{I_i^{(1)}>0\}}$, $H_{ij} = 0,\ j \geq i$. We assume that the mean of the Jacobian is independent of the number of hidden units, $\mu_{ij} := \mathbb{E}[J_{ij}]$. Then we can state the following bound on the Jacobian of the network for the special case when the input is one-dimensional.

Theorem (Dixon and Polson 2019) *If $X \in \mathbb{R}^p$ is i.i.d. and there are n hidden units, with ReLU activation i, then the variance of a single-layer feedforward network with K outputs is bounded by μ_{ij}*

$$\mathbb{V}[J_{ij}] = \mu_{ij}\frac{n-1}{n} < \mu_{ij},\ \forall i \in \{1,\ldots,K\}\ and\ \forall j \in \{1,\ldots,p\}. \tag{5.11}$$

See Appendix "Proof of Variance Bound on Jacobian" for the proof.

Remark 5.1 The theorem establishes a negative result for a ReLU activated shallow network—increasing the number of hidden units, increases the bound on the variance of the Jacobian, and hence reduces interpretability of the sensitivities. Note that if we do not assume that the mean of the Jacobian is fixed under varying n, then we have the more general bound:

$$\mathbb{V}[J_{ij}] \leq \mu_{ij}, \tag{5.12}$$

and hence the effect of network architecture on the bound of the variance of the Jacobian is not clear. Note that the theorem holds without (i) distributional assumptions on X other than i.i.d. data and (ii) specifying the number of data points. □

Remark 5.2 This result also suggests that the inputs should be rescaled so that each μ_{ij}, the expected value of the Jacobian, is a small positive value, although it may not be possible to find such a scaling for all (i, j) pairs. □

5.1 Chernoff Bounds

We can derive probabilistic bounds on the Jacobians for any choice of activation function. Let $\delta > 0$ and $a_1, \ldots, a_{n-1}$ be reals in $(0, 1]$. Let $X_1, \ldots, X_{n-1}$ be independent Bernoulli trials with $\mathbb{E}[X_k] = p_k$ so that

$$\mathbb{E}[J] = \sum_{k=1}^{n-1} a_k p_k = \mu. \tag{5.13}$$

The Chernoff-type bound exists on deviations of J above the mean

$$\Pr(J > (1+\delta)\mu) = \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu}. \tag{5.14}$$

A similar bound exists for deviations of J below the mean. For $\gamma \in (0, 1]$:

$$\Pr(J - \mu < -\gamma\mu) < \left[\frac{e^{\gamma}}{(1+\gamma)^{1+\gamma}}\right]^{\mu}. \tag{5.15}$$

These bounds are generally weak and are suited to large deviations, i.e. the tail regions. The bounds are shown in the Fig. 5.4 for different values of μ. Here, μ is increasing towards the upper right-hand corner of the plot.

5.2 Simulated Example

In this section, we demonstrate the estimation properties of neural network sensitivities applied to data simulated from a linear model. We show that the sensitivities in a neural network are consistent with the linear model, even if the neural network model is non-linear. We also show that the confidence intervals, estimated by sampling, converge with increasing hidden units.

We generate 400 simulated training samples from the following linear model with i.i.d. Gaussian error:

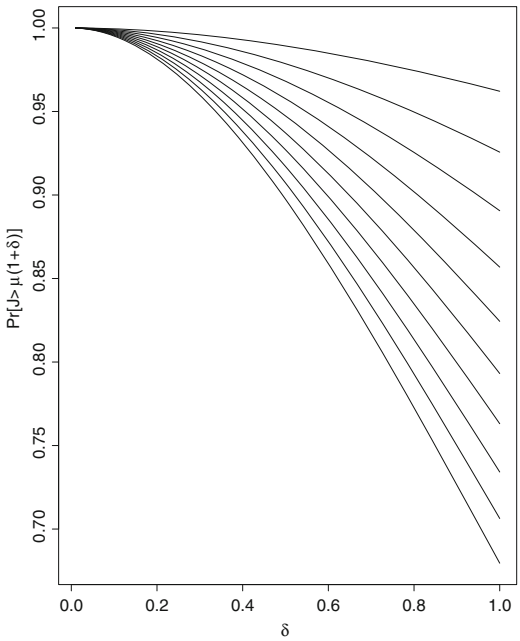


Fig. 5.4 The Chernoff-type bounds for deviations of J above the mean, μ. Various μ are shown in the plot, with μ increasing towards the upper right-hand corner of the plot

Table 5.1 This table compares the functional form of the variable sensitivities and values with an OLS estimator. NN_0 is a zero hidden layer feedforward network and NN_1 is a one hidden layer feedforward network with 10 hidden neurons and tanh activation functions

Model	Intercept		Sensitivity of X_1		Sensitivity of X_2	
OLS	$\hat{\beta}_0$	0.011	$\hat{\beta}_1$	1.015	$\hat{\beta}_2$	1.018
NN_0	$\hat{b}^{(1)}$	0.020	$\hat{W}_1^{(1)}$	1.018	$\hat{W}_2^{(1)}$	1.021
NN_1	$\hat{W}^{(2)}\sigma(\hat{b}^{(1)}) + \hat{b}^{(2)}$	0.021	$\mathbb{E}[\hat{W}^{(2)}D(I^{(1)})\hat{W}_1^{(1)}]$	1.014	$\mathbb{E}[\hat{W}^{(2)}D(I^{(1)})\hat{W}_2^{(1)}]$	1.022

$$Y = \beta_1 X_1 + \beta_2 X_2 + \epsilon, \quad X_1, X_2, \epsilon \sim N(0,1), \quad \beta_1 = 1, \beta_2 = 1. \tag{5.16}$$

Table 5.1 compares an OLS estimator with a zero hidden layer feedforward network (NN_0) and a one hidden layer feedforward network with 10 hidden neurons and tanh activation functions (NN_1). The functional form of the first two regression models is equivalent, although the OLS estimator has been computed using a matrix solver, whereas the zero layer hidden network parameters have been fitted with stochastic gradient descent.

The fitted parameters values will vary slightly with each optimization as the stochastic gradient descent is randomized. However, the sensitivity terms are given in closed form and easily mapped to the linear model. In an industrial setting, such a one-to-one mapping is useful for migrating to a deep factor model where, for model validation purposes, compatibility with linear models should be recovered in a limiting case. Clearly, if the data is not generated from a linear model, then the parameter values would vary across models.

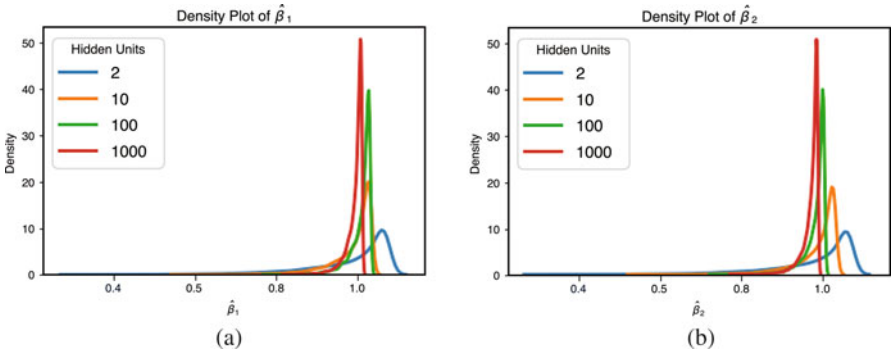


Fig. 5.5 This figure shows the empirical distribution of the sensitivities $\hat{\beta}_1$ and $\hat{\beta}_2$. The sharpness of the distribution is observed to converge with the number of hidden units. (**a**) Density of $\hat{\beta}_1$. (**b**) Density of $\hat{\beta}_2$

Table 5.2 This table shows the moments and 99% confidence interval of the empirical distribution of the sensitivity $\hat{\beta}_1$. The sharpness of the distribution is observed to converge monotonically with the number of hidden units

Hidden Units	Mean	Median	Std.dev	1% C.I.	99% C.I.
2	0.980875	1.0232913	0.10898393	0.58121675	1.0729908
10	0.9866159	1.0083131	0.056483902	0.76814914	1.0322522
50	0.99183553	1.0029879	0.03123002	0.8698967	1.0182846
100	1.0071343	1.0175397	0.028034585	0.89689034	1.0296803
200	1.0152218	1.0249312	0.026156902	0.9119074	1.0363332

Table 5.3 This table shows the moments and the 99% confidence interval of the empirical distribution of the sensitivity $\hat{\beta}_2$. The sharpness of the distribution is observed to converge monotonically with the number of hidden units

Hidden Units	Mean	Median	Std.dev	1% C.I.	99% C.I.
2	0.98129386	1.0233982	0.10931312	0.5787732	1.073728
10	0.9876832	1.0091512	0.057096474	0.76264584	1.0339714
50	0.9903236	1.0020974	0.031827927	0.86471796	1.0152498
100	0.9842479	0.9946766	0.028286876	0.87199813	1.0065105
200	0.9976638	1.0074166	0.026751818	0.8920307	1.0189484

Figure 5.5 and Tables 5.2 and 5.3 show the empirical distribution of the fitted sensitivities using the single hidden layer model with increasing hidden units. The sharpness of the distributions is observed to converge monotonically with the number of hidden units. The confidence intervals are estimated under a non-parametric distribution.

In general, provided the weights and biases of the network are finite, the variances of the sensitivities are bounded for any input and choice of activation function.

We do not recommend using ReLU activation because it does not permit identification of the interaction terms and has provably non-convergent sensitivity variances as a function of the number of hidden units (see Appendix "Proof of Variance Bound on Jacobian").

6 Factor Modeling

Rosenberg and Marathe (1976) introduced a cross-sectional fundamental factor model to capture the effects of macroeconomic events on individual securities. The choice of factors are microeconomic characteristics—essentially common factors, such as industry membership, financial structure, or growth orientation (Nielsen and Bender 2010).

The BARRA fundamental factor model expresses the linear relationship between K fundamental factors and N asset returns:

$$\mathbf{r_t} = B_t \mathbf{f}_t + \boldsymbol{\epsilon}_t, \ t = 1, \ldots, T, \tag{5.17}$$

where $B_t = [\mathbf{1} \mid \boldsymbol{\beta}_1(t) \mid \cdots \mid \boldsymbol{\beta}_K(t)]$ is the $N \times K+1$ matrix of known factor loadings (betas): $\beta_{i,k}(t) := \left(\boldsymbol{\beta}_k\right)_i (t)$ is the exposure of asset i to factor k at time t.

The factors are asset specific attributes such as market capitalization, industry classification, style classification. $\mathbf{f}_t = [\alpha_t, f_{1,t}, \ldots, f_{K,t}]$ is the $K+1$ vector of unobserved factor realizations at time t, including α_t.

$\mathbf{r_t}$ is the N-vector of asset returns at time t. The errors are assumed independent of the factor realizations $\rho(f_{i,t}, \epsilon_{j,t}) = 0, \forall i, j, t$ with Gaussian error, $\mathbb{E}[\epsilon_{j,t}^2] = \sigma^2$.

6.1 Non-linear Factor Models

We can extend the linear model to a non-linear cross-sectional fundamental factor model of the form

$$\mathbf{r}_t = F_t(B_t) + \boldsymbol{\epsilon}_t, \tag{5.18}$$

where $\mathbf{r}_t$ are asset returns, $F_t : \mathbb{R}^K \to \mathbb{R}$ is a differentiable non-linear function that maps the ith row of B to the ith asset return at time t. The map is assumed to incorporate a bias term so that $F_t(\mathbf{0}) = \alpha_t$. In the special case when $F_t(B_t)$ is linear, the map is $F_t(B_t) = B_t \mathbf{f}_t$.

A key feature is that we do not assume that $\boldsymbol{\epsilon}_t$ is described by a parametric distribution, such as a Gaussian distribution. In our example, we shall treat $\boldsymbol{\epsilon}_t$ as i.i.d., however, we can extend the methodology to non-i.i.d. idea as in Dixon and Polson (2019). In our setup, the model shall just be used to predict the next period returns only and stationarity of the factor realizations is not required.

We approximate a non-linear map, $F_t(B_t)$, with a feedforward neural network cross-sectional factor model:

$$\mathbf{r}_t = F_{W_t,b_t}(B_t) + \boldsymbol{\epsilon}_t, \tag{5.19}$$

where F_{W_t,b_t} is a deep neural network with L layers.

6.2 Fundamental Factor Modeling

This section presents an application of deep learning to a toy fundamental factor model. Factor models in practice include significantly more fundamental factors that used here. But the purpose, here, is to illustrate the application of interpretable deep learning to financial data.

We define the universe as the top 250 stocks from the S&P 500 index, ranked by market cap. Factors are given by Bloomberg and reported monthly over a hundred month period beginning in February 2008. We remove stocks with too many missing factor values, leaving 218 stocks.

The historical factors are inputs to the model and are standardized to enable model interpretability. These factors are (i) current enterprise value; (ii) Price-to-Book ratio; (iii) current enterprise value to trailing 12 month EBITDA; (iv) Price-to-Sales ratio; (v) Price-to-Earnings ratio; and (vi) log market cap. The responses are the monthly asset returns for each stock in our universe based on the daily adjusted closing prices of the stocks.

We use `Tensorflow` (Abadi et al. 2016) to implement a two hidden layer feedforward network and develop a custom implementation for the least squares error and variable sensitivities and is available in the deep factor models notebook. The OLS regression is implemented by the Python `StatsModels` module.

All deep learning results are shown using L_1 regularization and $tanh$ activation functions. The number of hidden units and regularization parameters are found by three-fold cross-validation and reported alongside the results.

Figure 5.6 compares the performance of an OLS estimator with the feedforward neural network with 10 hidden units in the first hidden layer and 10 hidden units in the second layer and $\lambda_1 = 0.001$.

Figure 5.7 shows the in-sample MSE as a function of the number of hidden units in the hidden layer. The neural networks are trained here without L_1 regularization to demonstrate the effect of solely increasing the number of hidden units in the first layer. Increasing the number of hidden units reduces the bias in the model.

Figure 5.8 shows the effect of L_1 regularization on the MSE errors for a network with 10 units in each of the two hidden layers. Increasing the level of L_1 regularization increases the in-sample bias but reduces the out-of-sample bias, and hence the variance of the estimation error.

Figure 5.9 compares the distribution of sensitivities to each factor over the entire 100 month period using the neural network (top) and OLS regression (bottom). The

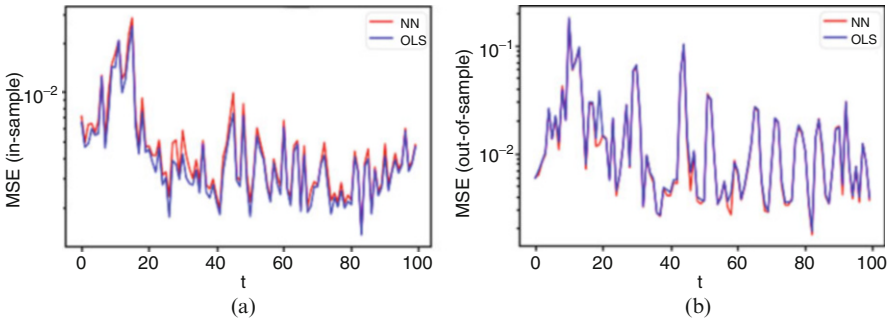
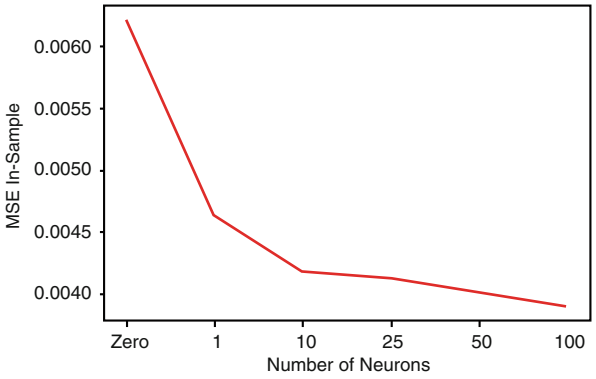


Fig. 5.6 This figures compares the in-sample and out-of-sample performances of an OLS estimator (OLS) with a feedforward neural network (NN), as measured by the mean squared error (MSE). The neural network is observed to always exhibit slightly lower out-of-sample MSE, although the effect of deep networks on this problem is marginal because the dataset is too simplistic. (**a**) In-sample error. (**b**) Out-of-sample error

Fig. 5.7 This figure shows the in-sample MSE as a function of the number of hidden units in the hidden layer. Increasing the number of hidden units reduces the bias in the model



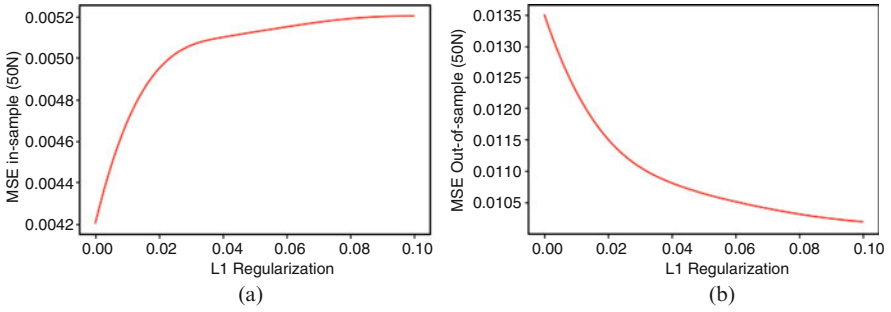


Fig. 5.8 These figures show the effect of L_1 regularization on the MSE errors for a network with 10 neurons in each of the two hidden layers. (**a**) In-sample. (**b**) Out-of-sample

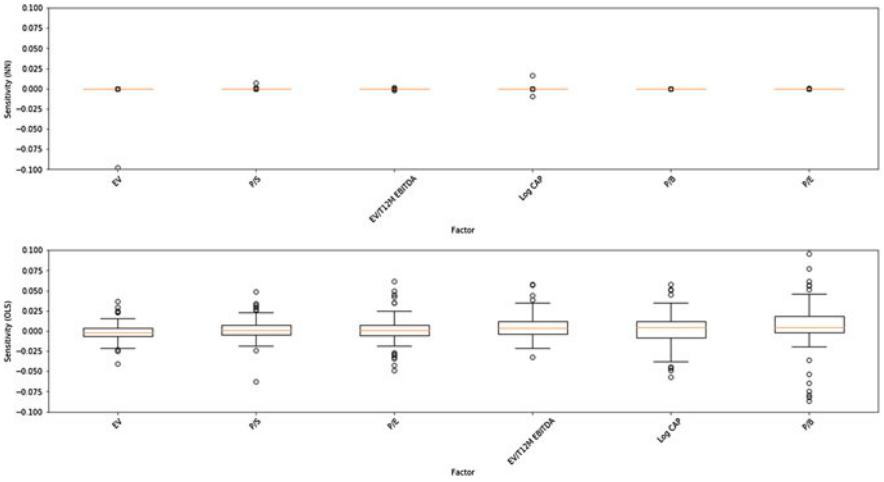


Fig. 5.9 The distribution of sensitivities to each factor over the entire 100 month period using the neural network (top). The sensitivities are sorted in ascending order from left to right by their median values. The same sensitivities using OLS linear regression (bottom)

sensitivities are sorted in ascending order from left to right by their median values. We observe that the OLS regression is much more sensitive to the factors than the NN. We further note that the NN ranks the top sensitivities differently to OLS.

Clearly, the above results are purely illustrative of the interpretability methodology and not intended to be representative of a real-world factor model. Such a choice of factors is observed to provide little benefit on the information ratios of a simple stock selection strategy.

Larger Dataset

For completeness, we provide evidence that our neural network factor model generates positive and higher information ratios than OLS when used to sort portfolios from a larger universe, using up to 50 factors (see Table 5.4 for a description of the factors). The dataset is not provided due to data licensing restrictions.

We define the universe as 3290 stocks from the Russell 3000 index. Factors are given by Bloomberg and reported monthly over the period from November 2008 to November 2018. We train a two-hidden layer deep network with 50 hidden units using ReLU activation.

Figure 5.10 compares the out-of-sample performance of neural networks and OLS regression by the MSE (left) and the information ratios of a portfolio selection strategy which selects the n stocks with the highest predicted monthly returns (right). The information ratios are evaluated for various size portfolios, using the Russell 3000 index as the benchmark. Also shown, for control, are randomly selected portfolios.

Table 5.4 A short description of the factors used in the Russell 3000 deep learning factor model demonstrated at the end of this chapter

	Value factors
B/P	Book to price
CF/P	Cash flow to price
E/P	Earning to price
S/EV	Sales to enterprise value (EV). EV is given by EV=Market Cap + LT Debt + max(ST Debt-Cash,0), where LT (ST) stands for long (short) term
EB/EV	EBIDTA to EV
FE/P	Forecasted E/P. Forecast earnings are calculated from Bloomberg earnings consensus estimates data
	For coverage reasons, Bloomberg uses the 1-year and 2-year forward earnings
DIV	Dividend yield. The exposure to this factor is just the most recently announced annual net dividends divided by the market price
	Stocks with high dividend yields have high exposures to this factor
	Size factors
MC	Log (Market capitalization)
S	Log (sales)
TA	Log (total assets)
	Trading activity factors
TrA	Trading activity is a turnover based measure
	Bloomberg focuses on turnover which is trading volume normalized by shares outstanding
	This indirectly controls for the Size effect
	The exponential weighted average (EWMA) of the ratio of shares traded to shares outstanding
	In addition, to mitigate the impacts of those sharp short-lived spikes in trading volume,
	Bloomberg winsorizes the data first daily trading volume data is compared to the long-term EWMA volume(180 day half-life), then the data is capped at 3 standard deviations away from the EWMA average
	Earnings variability factors
EaV/TA	Earnings volatility to total assets
	Earnings volatility is measured over the last 5 years/median total assets over the last 5 years
CFV/TA	Cash flow volatility to total assets
	Cash flow volatility is measured over the last 5 years/median total assets over the last 5 years
SV/TA	Sales volatility to total assets
	Sales volatility over the last 5 years/median total assets over the last 5 year

(continued)

Table 5.4 (continued)

	Volatility factors
RV	Rolling volatility which is the return volatility over the latest 252 trading days
CB	Rolling CAPM beta which is the regression coefficient from the rolling window regression of stock returns on local index returns
	Growth factors
TAG	Total asset growth is the 5-year average growth in total assets divided by the average total assets over the last 5 years
EG	Earnings growth is the 5-year average growth in earnings divided by the average total assets over the last 5 years
	GSIC sectorial codes
(I)ndustry	{10, 20, 30, 40, 50, 60, 70}
(S)ub-(I)ndustry	{10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, 80}
(S)ector	{10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60}
(I)ndustry (G)roup	{10, 20, 30, 40, 50}

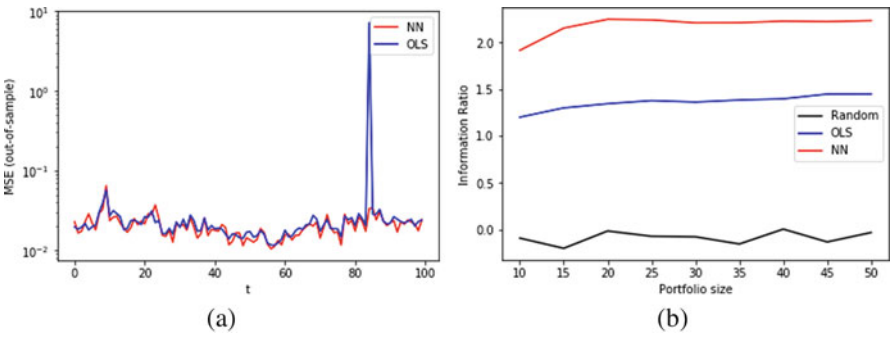


Fig. 5.10 (**a**) The out-of-sample MSE is compared between OLS and a two-hidden layer deep network applied to a universe of 3290 stocks from the Russell 3000 index over the period from November 2008 to November 2018. (**b**) The information ratios of a portfolio selection strategy which selects the n stocks from the universe with the highest predicted monthly returns. The information ratios are evaluated for various size portfolios. The information ratios are based on out-of-sample predicted asset returns using OLS regression, neural networks, and randomized selection with no predictive model

Finally, Fig. 5.11 compares the distribution of sensitivities to each factor over the entire 100 month period using the neural network (top) and OLS regression (bottom). The sensitivities are sorted in ascending order from left to right by their median values. We observe that the NN ranking of the factors differs substantially from the OLS regression.

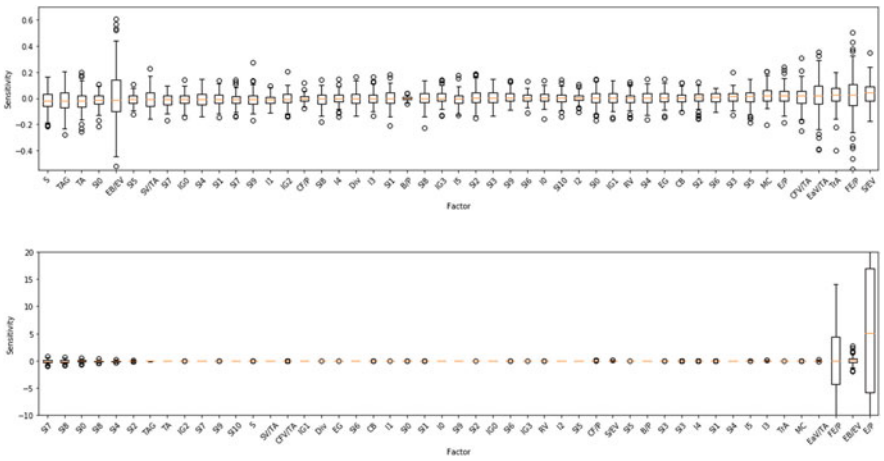


Fig. 5.11 The distribution of factor model sensitivities to each factor over the entire ten-year period using the neural network applied to the Russell 3000 asset factor loadings (top). The sensitivities are sorted in ascending order from left to right by their median values. The same sensitivities using OLS linear regression (bottom). See Table 5.4 for a short description of the fundamental factors

7 Summary

An important aspect in adoption of neural networks in factor modeling is the existence of a statistical framework which provides the transparency and statistical interpretability of linear least squares estimation. Moreover, one should expect to use such a framework applied to linear data and obtain similar results to linear regression, thus isolating the effects of non-linearity versus the effect of using different optimization algorithms and model implementation environments.

In this chapter, we introduce a deep learning framework for interpretable cross-sectional modeling and demonstrate its application to a simple fundamental factor model. Deep learning generalizes the linear fundamental factor models by capturing non-linearity, interaction effects, and non-parametric shocks in financial econometrics. This framework provides interpretability, with confidence intervals, and ranking of the factor importance and interaction effects. In the case when the network contains no hidden layers, our approach recovers a linear fundamental factor model. The framework allows the impact of non-linearity and non-parametric treatment of the error on the factors over time and forms the basis for generalized interpretability of fundamental factors.

8 Exercises

Exercise 5.1*
Consider the following data generation process

$$Y = X_1 + X_2 + \epsilon,\ X_1, X_2, \epsilon \sim N(0, 1),$$

i.e. $\beta_0 = 0$ and $\beta_1 = \beta_2 = 1$.

a. For this data, write down the mathematical expression for the sensitivities of the fitted neural network when the network has

   – zero hidden layers;
   – one hidden layer, with n unactivated hidden units;
   – one hidden layer, with n tanh activated hidden units;
   – one hidden layer, with n ReLU activated hidden units; and
   – two hidden layers, each with n tanh activated hidden units.

Exercise 5.2**
Consider the following data generation process

$$Y = X_1 + X_2 + X_1 X_2 + \epsilon,\ X_1, X_2 \sim N(0, 1),\ \epsilon \sim N(0, \sigma_n^2),$$

i.e. $\beta_0 = 0$ and $\beta_1 = \beta_2 = \beta_{12} = 1$, where β_{12} is the interaction term. σ_n^2 is the variance of the noise and $\sigma_n = 0.01$.

a. For this data, write down the mathematical expression for the interaction term (i.e., the off-diagonal components of the Hessian matrix) of the fitted neural network when the network has

   – zero hidden layers;
   – one hidden layer, with n unactivated hidden units;
   – one hidden layer, with n tanh activated hidden units;
   – one hidden layer, with n ReLU activated hidden units; and
   – two hidden layers, each with n tanh activated hidden units.

Why is the ReLU activated network problematic for estimating interaction terms?

*8.1 Programming Related Questions**

Exercise 5.3*
For the same problem in the previous exercise, use 5000 simulations to generate a regression training set dataset for the neural network with one hidden layer. Produce a table showing how the mean and standard deviation of the sensitivities β_i behave as the number of hidden units is increased. Compare your result with $tanh$ and $ReLU$ activation. What do you conclude about which

activation function to use for interpretability? Note that you should use the notebook `Deep-Learning-Interpretability.ipynb` as the starting point for experimental analysis.

Exercise 5.4*
Generalize the `sensitivities` function in Exercise 5.3 to L layers for either tanh or ReLU activated hidden layers. Test your function on the data generation process given in Exercise 5.1.

Exercise 5.5**
Fixing the total number of hidden units, how do the mean and standard deviation of the sensitivities β_i behave as the number of layers is increased? Your answer should compare using either tanh or ReLU activation functions. Note, do not mix the type of activation functions across layers. What you conclude about the effect of the number of layers, keeping the total number of units fixed, on the interpretability of the sensitivities?

Exercise 5.6**
For the same data generation process as the previous exercise, use 5000 simulations to generate a regression training set for the neural network with one hidden layer. Produce a table showing how the mean and standard deviation of the interaction term behave as the number of hidden units is increased, fixing all other parameters. What do you conclude about the effect of the number of hidden units on the interpretability of the interaction term? Note that you should use the notebook `Deep-Learning-Interaction.ipynb` as the starting point for experimental analysis.

Appendix

Other Interpretability Methods

Partial Dependence Plots (PDPs) evaluate the expected output w.r.t. the marginal density function of each input variable, and allow the importance of the predictors to be ranked. More precisely, partitioning the data X into an *interest* set, $\boldsymbol{X}_s$, and its complement, $\boldsymbol{X}_c = \mathcal{X} \setminus \boldsymbol{X}_s$, then the "partial dependence" of the response on $\boldsymbol{X}_s$ is defined as

$$f_s\left(\boldsymbol{X}_s\right) = E_{\boldsymbol{X}_c}\left[\widehat{f}\left(\boldsymbol{X}_s, \boldsymbol{X}_c\right)\right] = \int \widehat{f}\left(\boldsymbol{X}_s, \boldsymbol{X}_c\right) p_c\left(\boldsymbol{X}_c\right) d\boldsymbol{X}_c, \tag{5.20}$$

where $p_c\left(\boldsymbol{X}_c\right)$ is the marginal probability density of $\boldsymbol{X}_c$: $p_c\left(\boldsymbol{X}_c\right) = \int p\left(\boldsymbol{x}\right) d\boldsymbol{x}_s$. Equation (5.20) can be estimated from a set of training data by

$$\bar{f}_s\left(\boldsymbol{X}_s\right) = \frac{1}{n}\sum_{i=1}^{n} \widehat{f}\left(\boldsymbol{X}_s, \boldsymbol{X}_{i,c}\right), \tag{5.21}$$

where $\boldsymbol{X}_{i,c}$ $(i = 1, 2, \ldots, n)$ are the observations of $\boldsymbol{X}_c$ in the training set; that is, the effects of all the other predictors in the model are averaged out. There are a number of challenges with using PDPs for model interpretability. First, the interaction effects are ignored by the simplest version of this approach. While Greenwell et al. (2018) propose a methodology extension to potentially address the modeling of interactive effects, PDPs do not provide a 1-to-1 correspondence with the coefficients in a linear regression. Instead, we would like to know, under strict control conditions, how the fitted weights and biases of the MLP correspond to the fitted coefficients of linear regression. Moreover in the context of neural networks, by treating the model as a black-box, it is difficult to gain theoretical insight in to how the choice of the network architecture affects its interpretability from a probabilistic perspective.

Garson (1991) partitions hidden-output connection weights into components associated with each input neuron using absolute values of connection weights. Garson's algorithm uses the absolute values of the connection weights when calculating variable contributions, and therefore does not provide the direction of the relationship between the input and output variables.

Olden and Jackson (2002) determines the relative importance, $r_{ij} = [R]_{ij}$, of the ith output to the jth predictor variable of the model as a function of the weights, according to the expression

$$r_{ij} = W^{(2)}_{jk} W^{(1)}_{ki}. \tag{5.22}$$

The approach does not account for non-linearity introduced into the activation, which is the most critical aspects of the model. Furthermore, the approach presented was limited to a single hidden layer.

Proof of Variance Bound on Jacobian

Proof The Jacobian can be written in matrix element form as

$$J_{ij} = [\partial_X \hat{Y}]_{ij} = \sum_{k=1}^{n} w^{(2)}_{ik} w^{(1)}_{kj} H(I^{(1)}_k) = \sum_{k=1}^{n} c_k H_k(I) \tag{5.23}$$

where $c_k := c_{ijk} := w^{(2)}_{ik} w^{(1)}_{kj}$ and $H_k(I) := H(I^{(1)}_k)$ is the Heaviside function. As a linear combination of indicator functions, we have

$$J_{ij} = \sum_{k=1}^{n-1} a_k \mathbb{1}_{\{I^{(1)}_k > 0, I^{(1)}_{k+1} \leq 0\}} + a_n \mathbb{1}_{\{I^{(1)}_n > 0\}},\ a_k := \sum_{i=1}^{k} c_i. \tag{5.24}$$

Alternatively, the Jacobian can be expressed in terms of a weighted sum of independent Bernoulli trials involving X:

$$J_{ij} = \sum_{k=1}^{n-1} a_k \mathbb{1}_{\{\mathbf{w}_{k,}^{(1)} X > -b_k^{(1)}, \mathbf{w}_{k+1,}^{(1)} X \leq -b_{k+1}^{(1)}\}} + a_n \mathbb{1}_{\{\mathbf{w}_{n,}^{(1)} X > -b_n^{(1)}\}}. \tag{5.25}$$

Without loss of generality, consider the case when $p = 1$, the dimension of the input space is one. Then Eq. 5.25 simplifies to:

$$J_{ij} = \sum_{k=1}^{n-1} a_k \mathbb{1}_{x_k < X \leq x_{k+1}} + a_n \mathbb{1}_{x_n < X}, \;\; j = 1, \tag{5.26}$$

where $x_k := -\frac{b_k^{(1)}}{W_k^{(1)}}$. The expectation of the Jacobian is given by

$$\mu_{ij} := \mathbb{E}[J_{ij}] = \sum_{k=1}^{n} a_k p_k, \tag{5.27}$$

where $p_k := \Pr(x_k < X \leq x_{k+1}) \; \forall k = 1, \ldots, n-1, \;\; p_n := \Pr(x_n < X)$. For finite weights, the expectation is bounded above by $\sum_{k=1}^{n} a_k$. We can write the variance of the Jacobian as:

$$\mathbb{V}[J_{ij}] = \sum_{k=1}^{n-1} a_k \mathbb{V}[\mathbb{1}_{\{Z_k^{(1)} > 0, Z_{k+1}^{(1)} \leq 0\}}] + a_n \mathbb{V}[\mathbb{1}_{\{Z_n^{(1)} > 0\}}] = \sum_{k=1}^{n} a_k p_k (1 - p_k). \tag{5.28}$$

Under the assumption that the mean of the Jacobian is invariant to the number of hidden units, or if the weights are constrained so that the mean is constant, then the weights are $a_k = \frac{\mu_{ij}}{n p_k}$. Then the variance is bounded by the mean:

$$\mathbb{V}[J_{ij}] = \mu_{ij} \frac{n-1}{n} < \mu_{ij}. \tag{5.29}$$

If we relax the assumption that μ_{ij} is independent of n then, under the original weights $a_k := \sum_{i=1}^{k} c_i$:

$$\begin{aligned} \mathbb{V}[J_{ij}] &= \sum_{k=1}^{n} a_k p_k (1 - p_k) \\ &\leq \sum_{k=1}^{n} a_k p_k \\ &= \mu_{ij} \\ &\leq \sum_{k=1}^{n} a_k. \end{aligned}$$

□

Russell 3000 Factor Model Description

Python Notebooks

The notebooks provided in the accompanying source code repository are designed to gain familiarity with how to implement interpretable deep networks. The examples include toy simulated data and a simple factor model. Further details of the notebooks are included in the `README.md` file.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). Tensor flow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16 (pp. 265–283).

Dimopoulos, Y., Bourret, P., & Lek, S. (1995, Dec). Use of some sensitivity criteria for choosing networks with good generalization ability. *Neural Processing Letters, 2*(6), 1–4.

Dixon, M. F., & Polson, N. G. (2019). Deep fundamental factor models.

Garson, G. D. (1991, April). Interpreting neural-network connection weights. *AI Expert, 6*(4), 46–51.

Greenwell, B. M., Boehmke, B. C., & McCarthy, A. J. (2018, May). A simple and effective model-based variable importance measure. *arXiv e-prints*, arXiv:1805.04755.

Nielsen, F., & Bender, J. (2010). The fundamentals of fundamental factor models. Technical Report 24, MSCI Barra Research Paper.

Olden, J. D., & Jackson, D. A. (2002). Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks. *Ecological Modelling, 154*(1), 135–150.

Rosenberg, B., & Marathe, V. (1976). Common factors in security returns: Microeconomic determinants and macroeconomic correlates. Research Program in Finance Working Papers 44, University of California at Berkeley.

Part II
Sequential Learning

Chapter 6
Sequence Modeling

This chapter provides an overview of the most important modeling concepts in financial econometrics. Such methods form the conceptual basis and performance baseline for more advanced neural network architectures presented in the next chapter. In fact, each type of architecture is a generalization of many of the models presented here. This chapter is especially useful for students from an engineering or science background, with little exposure to econometrics and time series analysis.

1 Introduction

More often in finance, the data consists of observations of a variable over time, e.g. stock prices, bond yields, etc. In such a case, the observations are not independent over time, rather observations are often strongly related to their recent histories. For this reason, the ordering of the data matters (unlike cross-sectional data). This is in contrast to most methods of machine learning which assume that the data is i.i.d. Moreover algorithms and techniques for fitting machine learning models, such as back-propagation for neural networks and cross-validation for hyperparameter tuning, must be modified for use on time series data.

"Stationarity" of the data is a further important delineation necessary to successfully apply models to time series data. If the estimated moments of the data change depending on the window of observation, then the modeling problem is much more difficult. Neural network approaches to addressing these challenges are presented in the next chapter.

An additional consideration is the *data frequency*—the frequency at which the data is observed assuming that the timestamps are uniform. In general, the frequency of the data governs the frequency of the time series model. For example, support that we seek to predict the week ahead stock price from daily historical adjusted close prices on business days. In such a case, we would build a model from daily prices

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_6

and then predict 5 daily steps ahead, rather than building a model using only weekly intervals of data.

In this chapter we shall primarily consider applications of parametric, linear, and frequentist models to uniform time series data. Such methods form the conceptual basis and performance baseline for more advanced neural network architectures presented in the next chapter. In fact, each type of architecture is a generalization of many of the models presented here. Please note that the material presented in this chapter is not intended as a substitute for a more comprehensive and rigorous treatment of econometrics, but rather to provide enough background for Chap. 8.

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Explain and analyze linear autoregressive models;
- Understand the classical approaches to identifying, fitting, and diagnosing autoregressive models;
- Apply simple heteroscedastic regression techniques to time series data;
- Understand how exponential smoothing can be used to predict and filter time series; and
- Project multivariate time series data onto lower dimensional spaces with principal component analysis.

Note that this chapter can be skipped if the reader is already familiar with econometrics. This chapter is especially useful for students from an engineering or physical sciences background, with little exposure to econometrics and time series analysis.

2 Autoregressive Modeling

We begin by considering a single variable Y_t indexed by t to indicate the variable changes over time. This variable may depend on other variables X_t; however, we shall simply consider the case when the dependence of Y_t is on past observations of itself—this is known as *univariate time series analysis*.

2.1 Preliminaries

Before we can build a model to predict Y_t, we recall some basic definitions and terminology, starting with a continuous time setting and then continuing thereafter solely in a discrete-time setting.

> Stochastic Process

A stochastic process is a sequence of random variables, indexed by continuous time: $\{Y_t\}_{t=-\infty}^{\infty}$.

> Time Series

A time series is a sequence of observations of a stochastic process at discrete times over a specific interval: $\{y_t\}_{t=1}^{n}$.

> Autocovariance

The jth autocovariance of a time series is $\gamma_{jt} := \mathbb{E}[(y_t - \mu_t)(y_{t-j} - \mu_{t-j})]$, where $\mu_t := \mathbb{E}[y_t]$.

> Covariance (Weak) Stationarity

A time series is weak (or wide-sense) covariance stationary if it has time constant mean and autocovariances of all orders:

$$\mu_t = \mu, \qquad \forall t$$
$$\gamma_{jt} = \gamma_j, \qquad \forall t.$$

As we have seen, this implies that $\gamma_j = \gamma_{-j}$: the autocovariances depend only on the interval between observations, but not the time of the observations.

> Autocorrelation

The jth autocorrelation, τ_j is just the jth autocovariance divided by the variance:

$$\tau_j = \frac{\gamma_j}{\gamma_0}. \tag{6.1}$$

> White Noise

White noise, ϵ_t, is i.i.d. error which satisfies all three conditions:

a. $\mathbb{E}[\epsilon_t] = 0, \forall t$;
b. $\mathbb{V}[\epsilon_t] = \sigma^2, \forall t$; and
c. ϵ_t and ϵ_s are independent, $t \neq s, \forall t, s$.

Gaussian white noise just adds a normality assumption to the error. White noise error is often referred to as a "disturbance," "shock," or "innovation" in the financial econometrics literature.

With these definitions in place, we are now ready to define autoregressive processes. Tacit in our usage of these models is that the time series exhibits autocorrelation.[1] If this is not the case, then we would choose to use cross-sectional models seen in Part I of this book.

2.2 *Autoregressive Processes*

Autoregressive models are parametric time series models describing y_t as a linear combination of p past observations and white noise. They are referred to as "processes" as they are representative of random processes which are dependent on one or more past values.

> AR(p) Process

The pth order autoregressive process of a variable Y_t depends only on the previous values of the variable plus a white noise disturbance term

$$y_t = \mu + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t, \tag{6.2}$$

where ϵ_t is independent of $\{y_{t-1}\}_{i=1}^{p}$. We refer to μ as the *drift term*. p is referred to as the *order* of the model.

Defining the polynomial function $\phi(L) := (1 - \phi_1 L - \phi_2 L^2 - \cdots - \phi_p L^p)$, where y_{t-j} is a jth lagged observation of y_t given by the *Lag operator* or *Backshift* operator, $y_{t-j} = L^j[y_j]$.

The AR(p) process can be expressed in the more compact form

$$\phi(L)[y_t] = \mu + \epsilon_t. \tag{6.3}$$

[1] We shall identify statistical tests for establishing autocorrelation later in this chapter.

This compact form shall be conducive to analysis describing the properties of the $AR(p)$ process. We mention in passing that the identification of the parameter p from data, i.e. the number of lags in the model rests on the data being weakly covariance stationary.[2]

2.3 Stability

An important property of AR(p) processes is whether past disturbances exhibit an inclining or declining impact on the current value of y as the lag increases. For example, think of the impact of a news event about a public company on the stock price movement over the next minute versus if the same news event had occurred, say, six months in the past. One should expect that the latter is much less significant than the former.

To see this, consider the AR(1) process and write y_t in terms of the inverse of $\Phi(L)$

$$y_t = \Phi^{-1}(L)[\mu + \epsilon_t], \tag{6.4}$$

so that for an AR(1) process

$$y_t = \frac{1}{1 - \phi L}[\mu + \epsilon_t] = \sum_{j=0}^{\infty} \phi^j L^j [\mu + \epsilon_t], \tag{6.5}$$

and the infinite sum will be stable, i.e. the ϕ^j terms do not grow with j, provided that $|\phi| < 1$. Conversely, unstable AR(p) processes exhibit the counter-intuitive behavior that the error disturbance terms become increasingly influential as the lag increases. We can calculate the *Impulse Response Function* (IRF), $\frac{\partial y_t}{\partial \epsilon_{t-j}} \ \forall j$, to characterize the influence of past disturbances. For the AR(p) model, the IRF is given by ϕ^j and hence is geometrically decaying when the model is stable.

2.4 Stationarity

Another desirable property of AR(p) models is that their autocorrelation function convergences to zero as the lag increases. A sufficient condition for convergence is *stationary*. From the characteristic equation

[2] Statistical tests for identifying the order of the model will be discussed later in the chapter.

$$\Phi(z) = (1 - \frac{z}{\lambda_1}) \cdot (1 - \frac{z}{\lambda_2}) \cdot \ldots \cdot (1 - \frac{z}{\lambda_p}) = 0, \tag{6.6}$$

it follows that a $AR(p)$ model is strictly stationary and ergodic if all the roots lie outside the unit sphere in the complex plane $\mathbb{C}$. That is $|\lambda_i| > 1,\ i \in \{1, \ldots, p\}$ and $|\cdot|$ is the modulus of a complex number. Note that if the characteristic equation has at least one unit root, with all other roots lying outside the unit sphere, then this is a special case of non-stationarity but not strict stationarity.

Stationarity of Random Walk

We can show that the following random walk (zero mean AR(1) process) is not strictly stationary:

$$y_t = y_{t-1} + \epsilon_t \tag{6.7}$$

Written in compact form gives

$$\Phi(L)[y_t] = \epsilon_t,\ \Phi(L) = 1 - L, \tag{6.8}$$

and the characteristic polynomial, $\Phi(z) = 1 - z = 0$, implies that the real root $z = 1$. Hence the root is on the unit circle and the model is a special case of non-stationarity.

Finding roots of polynomials is equivalent to finding eigenvalues. The Cayley–Hamilton theorem states that the roots of any polynomial can be found by turning it into a matrix and finding the eigenvalues.

Given the p degree polynomial[3]:

$$q(z) = c_0 + c_1 z + \ldots + c_{p-1} z^{p-1} + z^p, \tag{6.9}$$

we define the $p \times p$ companion matrix

$$C := \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ -c_0 & -c_1 & \ldots & -c_{p-2} & -c_{p-1} \end{pmatrix}, \tag{6.10}$$

[3]Notice that the z^p coefficient is 1.

then the characteristic polynomial $det(C - \lambda I) = q(\lambda)$, and so the eigenvalues of C are the roots of q. Note that if the polynomial does not have a unit leading coefficient, then one can just divide the polynomial by that coefficient to arrive at the form of Eq. 6.9, without changing its roots. Hence the roots of any polynomial can be found by computing the eigenvalues of a companion matrix.

The AR(p) has a characteristic polynomial of the form

$$\Phi(z) = 1 - \phi_1 z - \cdots - \phi_p z^p \tag{6.11}$$

and dividing by $-\phi_p$ gives

$$q(z) = -\frac{\Phi(z)}{\phi_p} = -\frac{1}{\phi_p} + \frac{\phi_1}{\phi_p} z + \cdots + z^p \tag{6.12}$$

and hence the companion matrix is of the form

$$C := \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ \frac{1}{\phi_p} & -\frac{\phi_1}{\phi_p} & \cdots & -\frac{\phi_{p-1}}{\phi_p} & -\frac{\phi_{p-1}}{\phi_p} \end{pmatrix}. \tag{6.13}$$

2.5 Partial Autocorrelations

Autoregressive models carry a signature which allows its order, p, to be determined from time series data provided the data is stationary. This signature encodes the memory in the model and is given by "partial autocorrelations." Informally each partial autocorrelation measures the correlation of a random variable, y_t, with its lag, y_{t-h}, while controlling for intermediate lags. The formal definition of the partial autocorrelation is now given.

Partial Autocorrelation

A partial autocorrelation at lag $h \geq 2$ is a conditional autocorrelation between a variable, y_t, and its hth lag, y_{t-h} under the assumption that the values of the intermediate lags, $y_{t-1}, \ldots, y_{t-h+1}$ are controlled:

$$\tilde{\tau}_h := \tilde{\tau}_{t,t-h} := \frac{\tilde{\gamma}_h}{\sqrt{\tilde{\gamma}_{t,h}}\sqrt{\tilde{\gamma}_{t-h,h}}},$$

where $\tilde{\gamma}_h := \tilde{\gamma}_{t,t-h} := \mathbb{E}[y_t - P(y_t \mid y_{t-1}, \ldots, y_{t-h+1}), y_{t-h} - P(y_{t-h} \mid y_{t-1}, \ldots, y_{t-h+1})]$ is the lag-h partial autocovariance, $P(W \mid Z)$ is an orthogonal projection of W onto the set Z and

$$\tilde{\gamma}_{t,h} := \mathbb{E}[(y_t - P(y_t \mid y_{t-1}, \ldots, y_{t-h+1}))^2]. \tag{6.14}$$

The partial autocorrelation function $\tilde{\tau}_h : \mathbb{N} \to [-1, 1]$ is a map $h :\mapsto \tilde{\tau}_h$. The plot of $\tilde{\tau}_h$ against h is referred to as the partial *correlogram*.

AR(p) Processes

Using the property that a linear orthogonal projection $\hat{y}_t = P(y_t \mid y_{t-1}, \ldots, y_{t-h+1})$ is given by the OLS estimator as $\hat{y}_t = \phi_1 y_{t-1} + \cdots + \phi_{h-1} y_{t-h+1}$, gives the *Yule-Walker equations* for an AR(p) process, relating the partial autocorrelations $\tilde{\mathcal{T}}_p := [\tilde{\tau}_1, \ldots, \tilde{\tau}_p]$ to the autocorrelations $\mathcal{T}_p := [\tau_1, \ldots, \tau_p]$:

$$R_p \tilde{\mathcal{T}}_p = \mathcal{T}_p, \;\; R_p = \begin{bmatrix} 1 & \tau_1 & \ldots & \tau_{p-1} \\ \tau_1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \tau_{p-1} & \tau_{p-2} & \ldots & 1 \end{bmatrix}. \tag{6.15}$$

For $h \leq p$, we can solve for the hth lag partial autocorrelation by writing

$$\tilde{\tau}_h = \frac{|R_h^*|}{|R_h|}, \tag{6.16}$$

where $|\cdot|$ is the matrix determinant and the jth column of $[R_h^*]_{,j} = [R_h]_{,j}, \;\; j \neq h$ and the hth column is $[R_h^*]_{,h} = \mathcal{T}_h$.

For example, the lag-1 partial autocorrelation is $\tilde{\tau}_1 = \tau_1$ and the lag-2 partial autocorrelation is

$$\tilde{\tau}_2 = \frac{\begin{vmatrix} 1 & \tau_1 \\ \tau_1 & \tau_2 \end{vmatrix}}{\begin{vmatrix} 1 & \tau_1 \\ \tau_1 & 1 \end{vmatrix}} = \frac{\tau_2 - \tau_1^2}{1 - \tau_1^2}. \tag{6.17}$$

We note, in particular, that the lag-2 partial autocorrelation of an AR(1) process, with autocorrelation $\mathcal{T}_2 = [\tau_1, \tau_1^2]$ is

$$\tilde{\tau}_2 = \frac{\tau_1^2 - \tau_1^2}{1 - \tau_1^2} = 0, \tag{6.18}$$

and this is true for all lag orders greater than the order of the AR process. We can reason about this property from another perspective—through the partial autocovariances. The lag-2 partial autocovariance of an AR(1) process is

$$\tilde{\gamma}_2 := \tilde{\gamma}_{t,t-2} := \mathbb{E}[y_t - \hat{y}_t, y_{t-2} - \hat{y}_{t-2}], \tag{6.19}$$

where $\hat{y} = P(y_t \mid y_{t-1})$ and $\hat{y}_{t-2} = P(y_{t-2} \mid y_{t-1})$. When P is a linear orthogonal projection, we have from the property of an orthogonal projection

$$P(W \mid Z) = \mu_W + \frac{Cov(W, Z)}{\mathbb{V}[Z]}(Z - \mu_Z) \tag{6.20}$$

and $P(y_t \mid y_{t-1}) = \phi \frac{\mathbb{V}(y_{t-1})}{\mathbb{V}(y_{t-1})} = \phi$ so that

$\hat{y}_t = \phi y_{t-1}$, $\hat{y}_{t-2} = \phi y_{t-1}$ and hence $\epsilon_t = y_t - \hat{y}_t$ so the lag-2 partial autocovariance is

$$\tilde{\gamma}_2 = \mathbb{E}[\epsilon_t, y_{t-2} - \phi y_{t-1}] = 0. \tag{6.21}$$

Clearly the lag-1 partial autocovariance of an AR(1) process is

$$\tilde{\gamma}_1 = \mathbb{E}[y_t - \mu, y_{t-1} - \mu] = \gamma_1 = \phi\gamma_0. \tag{6.22}$$

2.6 Maximum Likelihood Estimation

The exact likelihood when the density of the data is independent of (ϕ, σ_n^2) is

$$\mathcal{L}(y, x; \phi, \sigma_n^2) = \prod_{t=1}^{T} f_{Y_t|X_t}(y_t|x_t; \phi, \sigma_n^2) f_{X_t}(x_t). \tag{6.23}$$

Under this assumption, the exact likelihood is proportional to the conditional likelihood function:

$$\begin{aligned}
\mathcal{L}(y, x; \phi, \sigma_n^2) &\propto L(y|x; \phi, \sigma_n^2) \\
&= \prod_{t=1}^{T} f_{Y_t|X_t}(y_t|x_t; \phi, \sigma_n^2) \\
&= (\sigma_n^2 2\pi)^{-T/2} \exp\{-\frac{1}{2\sigma_n^2} \sum_{t=1}^{T} (y_t - \phi^T \mathbf{x}_t)^2\}.
\end{aligned}$$

In many cases such an assumption about the independence of the density of the data and the parameters is not warranted. For example, consider the zero mean AR(1) with unknown noise variance:

$$y_t = \phi y_{t-1} + \epsilon_t,\ \epsilon_t \sim \mathcal{N}(0, \sigma_n^2) \tag{6.24}$$

$$Y_t | Y_{t-1} \sim \mathcal{N}(\phi y_{t-1}, \sigma_n^2)$$

$$Y_1 \sim \mathcal{N}(0, \frac{\sigma_n^2}{1-\phi^2}).$$

The exact likelihood is

$$\begin{aligned}\mathcal{L}(x; \phi, \sigma_n^2) &= f_{Y_t|Y_{t-1}}(y_t|y_{t-1}; \phi, \sigma_n^2) f_{Y_1}(y_1; \phi, \sigma_n^2) \\ &= \left(\frac{\sigma_n^2}{1-\phi^2} 2\pi\right)^{-1/2} \exp\{-\frac{1-\phi^2}{2\sigma_n^2} y_1^2\} (\sigma_n^2 2\pi)^{-\frac{T-1}{2}} \\ &\qquad \exp\{-\frac{1}{2\sigma_n^2} \sum_{t=2}^{T} (y_t - \phi y_{t-1})^2\},\end{aligned}$$

where we made use of the moments of Y_t—a result which is derived in Sect. 2.8.

Despite the dependence of the density of the data on the parameters, there may be practically little advantage of using exact maximum likelihood against the conditional likelihood method (i.e., dropping the $f_{Y_1}(y_1; \phi, \sigma_n^2)$ term). This turns out to be the case for linear models. Maximizing the conditional likelihood is equivalent to ordinary least squares estimation.

2.7 Heteroscedasticity

The AR model assumes that the noise is i.i.d. This may be an overly optimistic assumption which can be relaxed by assuming that the noise is time dependent. Treating the noise as time dependent is exemplified by a heteroscedastic AR(p) model

$$y_t = \mu + \sum_{i=1}^{p} \phi_i y_{t-i} + \epsilon_t,\ \epsilon_t \sim \mathcal{N}(0, \sigma_{n,t}^2). \tag{6.25}$$

There are many tests for heteroscedasticity in time series models and one of them, the ARCH test, is summarized in Table 6.3. The estimation procedure for heteroscedastic models is more complex and involves two steps: (i) estimation of the errors from the maximum likelihood function which treats the errors as independent and (ii) estimation of model parameters under a more general maximum likelihood

estimation which treats the errors as time-dependent. Note that such a procedure could be generalized further to account for correlation in the errors but requires the inversion of the covariance matrix, which is computationally intractable with large time series.

The conditional likelihood is

$$\begin{aligned}\mathcal{L}(\mathbf{y}|X;\phi,\sigma_n^2) &= \prod_{t=1}^{T} f_{Y_t|X_t}(y_t|x_t;\phi,\sigma_n^2) \\ &= (2\pi)^{-T/2} det(D)^{-1/2} \exp\{-\frac{1}{2}(\mathbf{y}-\phi^T X)^T D^{-1}(\mathbf{y}-\phi^T X)\},\end{aligned}$$

where $D_{tt} = \sigma_{n,t}^2$ is the diagonal covariance matrix and $X \in \mathbb{R}^{T\times p}$ is the data matrix defined as $[X]_t = \mathbf{x}_t$.

The advantage of this approach is its relative simplicity. The treatment of noise variance as time dependent in finance has long been addressed by more sophisticated econometrics models and the approach presented here brings AR models into line with the specifications of more realistic models.

On the other hand, the use of the sample variance of the residuals is only appropriate when the sample size is sufficient. In practice, this translates into the requirement for a sufficiently large historical period before a prediction can be made. Another disadvantage is that the approach does not explicitly define the relationship between the variances. We shall briefly revisit heteroscedastic models and explore a model for regressing the conditional variance on previous conditional variances in Sect. 2.9.

2.8 Moving Average Processes

The Wold representation theorem (a.k.a. Wold decomposition) states that every covariance stationary time series can be written as the sum of two time series, one deterministic and one stochastic. In effect, we have already considered the deterministic component when choosing an AR process.[4] The stochastic component can be represented as a "moving average process" or MA(q) process which expresses y_t as a linear combination of current and q past disturbances. Its definition is as follows:

MA(q) Process

The qth order moving average process is the linear combination of the white noise process $\{\epsilon_{t-i}\}_{t=0}^{q}$, $\forall t$

[4]This is an overly simplistic statement because the AR(1) process can be expressed as a MA(∞) process and vice versa.

$$y_t = \mu + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t. \tag{6.26}$$

It turns out that y_{t-1} depends on $\{\epsilon_{t-1}, \epsilon_{t-2}, \dots\}$, but not ϵ_t and hence $\gamma^2_{t,t-2} = 0$. It should be apparent that this property holds even when P is a non-linear projection provided that the errors are independent (but not necessarily identical).

Another brief point of discussion is that an AR(1) process can be rewritten as a MA(∞) process. Suppose that the AR(1) process has a mean μ and the variance of the noise is σ_n^2, then by a binomial expansion of the operator $(1 - \phi L)^{-1}$ we have

$$y_t = \frac{\mu}{1-\phi} + \sum_{j=0}^{\infty} \phi^j \epsilon_{t-j}, \tag{6.27}$$

where the moments can be easily found and are

$$\begin{aligned}
\mathbb{E}[y_t] &= \frac{\mu}{1-\phi} \\
\mathbb{V}[y_t] &= \sum_{j=0}^{\infty} \phi^{2j} \mathbb{E}[\epsilon^2_{t-j}] \\
&= \sigma_n^2 \sum_{j=0}^{\infty} \phi^{2j} = \frac{\sigma_n^2}{1-\phi^2}.
\end{aligned}$$

AR and MA models are important components of more complex models which are known as ARMA or, more generally, ARIMA models. The expression of a pattern as a linear combination of past observations and past innovations turns out to be more flexible in time series modeling than any single component. These are by no means the only useful techniques and we briefly turn to another technique which smooths out shorter-term fluctuations and consequently boosts the signal to noise ratio in longer term predictions.

2.9 GARCH

Recall from Sect. 2.7 that heteroscedastic time series models treat the error as time dependent. A popular parametric, linear, and heteroscedastic method used in financial econometrics is the Generalized Autoregressive Conditional Heteroscedastic (GARCH) model (Bollerslev and Taylor) . A GARCH(p,q) model specifies that the conditional variance (i.e., volatility) is given by an ARMA(p,q) model—there are p lagged conditional variances and q lagged squared noise terms:

$$\sigma_t^2 := \mathbb{E}[\epsilon_t^2|\Omega_{t-1}] = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2.$$

This model gives an explicit relationship between the current volatility and previous volatilities. Such a relationship is useful for predicting volatility in the model, with obvious benefits for volatility modeling in trading and risk management. This simple relationship enables us to characterize the behavior of the model, as we shall see shortly.

A necessary condition for model stationarity is the following constraint:

$$(\sum_{i=1}^{q} \alpha_i + \sum_{i=1}^{p} \beta_i) < 1.$$

When the model is stationary, the long-run volatility converges to the unconditional variance of ϵ_t:

$$\sigma^2 := var(\epsilon_t) = \frac{\alpha_0}{1 - (\sum_{i=1}^{q} \alpha_i + \sum_{i=1}^{p} \beta_i)}.$$

To see this, let us consider the l-step ahead forecast using a GARCH(1,1) model:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2 \tag{6.28}$$

$$\hat{\sigma}_{t+1}^2 = \alpha_0 + \alpha_1 \mathbb{E}[\epsilon_t^2|\Omega_{t-1}] + \beta_1 \sigma_t^2 \tag{6.29}$$

$$= \sigma^2 + (\alpha_1 + \beta_1)(\sigma_t^2 - \sigma^2) \tag{6.30}$$

$$\hat{\sigma}_{t+2}^2 = \alpha_0 + \alpha_1 \mathbb{E}[\epsilon_{t+1}^2|\Omega_{t-1}] + \beta_1 \mathbb{E}[\sigma_{t+1}^2|\Omega_{t-1}] \tag{6.31}$$

$$= \sigma^2 + (\alpha_1 + \beta_1)^2(\sigma_t^2 - \sigma^2) \tag{6.32}$$

$$\hat{\sigma}_{t+l}^2 = \alpha_0 + \alpha_1 \mathbb{E}[\epsilon_{t+l-1}^2|\Omega_{t-1}] + \beta_1 \mathbb{E}[\sigma_{t+l-1}^2|\Omega_{t-1}] \tag{6.33}$$

$$= \sigma^2 + (\alpha_1 + \beta_1)^l(\sigma_t^2 - \sigma^2), \tag{6.34}$$

where we have substituted for the unconditional variance, $\sigma^2 = \alpha_0/(1 - \alpha_1 - \beta_1)$. From the above equation we can see that $\hat{\sigma}_{t+1}^2 \to \sigma^2$ as $l \to \infty$ so as the forecast horizon goes to infinity, the variance forecast approaches the unconditional variance of ϵ_t. From the l-step ahead variance forecast, we can see that $(\alpha_1 + \beta_1)$ determines how quickly the variance forecast converges to the unconditional variance. If the variance sharply rises during a crisis, the number of periods, K, until it is halfway between the first forecast and the unconditional variance is $(\alpha_1 + \beta_1)^K = 0.5$, so the half-life[5] is given by $K = ln(0.5)/ln(\alpha_1 + \beta_1)$.

[5]The half-life is the lag k at which its coefficient is equal to a half.

For example, if

$$(\alpha_1 + \beta_1) = 0.97$$

and steps are measured in days, the half-life is approximately 23 days.

2.10 Exponential Smoothing

Exponential smoothing is a type of forecasting or filtering method that exponentially decreases the weight of past and current observations to give smoothed predictions $\tilde{y}_{t+1}$. It requires a single parameter, α, also called the smoothing factor or smoothing coefficient. This parameter controls the rate at which the influence of the observations at prior time steps decay exponentially. α is often set to a value between 0 and 1. Large values mean that the model pays attention mainly to the most recent past observations, whereas smaller values mean more of the history is taken into account when making a prediction. Exponential smoothing takes the forecast for the previous period $\tilde{y}_t$ and adjusts with the forecast error, $y_t - \tilde{y}_t$. The forecast for the next period becomes

$$\tilde{y}_{t+1} = \tilde{y}_t + \alpha(y_t - \tilde{y}_t), \tag{6.35}$$

or equivalently

$$\tilde{y}_{t+1} = \alpha y_t + (1 - \alpha)\tilde{y}_t. \tag{6.36}$$

Writing this as a geometric decaying autoregressive series back to the first observation:

$$\begin{aligned}\tilde{y}_{t+1} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \alpha(1-\alpha)^3 y_{t-3} \\ + \cdots + \alpha(1-\alpha)^{t-1} y_1 + \alpha(1-\alpha)^t \tilde{y}_1,\end{aligned}$$

hence we observe that smoothing introduces a long-term model of the entire observed data, not just a sub-sequence used for prediction in a AR model, for example. For geometrically decaying models, it is useful to characterize it by the half-life—the lag k at which its coefficient is equal to a half:

$$\alpha(1-\alpha)^k = \frac{1}{2}, \tag{6.37}$$

or

$$k = -\frac{ln(2\alpha)}{ln(1-\alpha)}. \tag{6.38}$$

The optimal amount of smoothing, $\hat{\alpha}$, is found by maximizing a likelihood function.

3 Fitting Time Series Models: The Box–Jenkins Approach

While maximum likelihood estimation is the approach of choice for fitting the ARMA models described in this chapter, there are many considerations beyond fitting the model parameters. In particular, we know from earlier chapters that the bias–variance tradeoff is a central consideration which is not addressed in maximum likelihood estimation without adding a penalty term.

Machine learning achieves generalized performance through optimizing the bias–variance tradeoff, with many of the parameters being optimized through cross-validation. This is both a blessing and a curse. On the one hand, the heavy reliance on numerical optimization provides substantial flexibility but at the expense of computational cost and, often-times, under-exploitation of structure in the time series. There are also potential instabilities whereby small changes in hyperparameters lead to substantial differences in model performance.

If one were able to restrict the class of functions represented by the model, using knowledge of the relationship and dependencies between variables, then one could in principle reduce the complexity and improve the stability of the fitting procedure.

For some 75 years, econometricians and statisticians have approached the problem of time series modeling with ARIMA in a simple and intuitive way. They follow a three-step process to fit and assess $AR(p)$. This process is referred to as a Box–Jenkins approach or framework. The three basic steps of the Box–Jenkins modeling approach are:

a. (I)dentification—determining the order of the model (a.k.a. model selection);
b. (E)stimation—estimation of model parameters; and
c. (D)iagnostic checking—evaluating the fit of the model.

This modeling approach is iterative and *parsimonious*—it favors models with fewer parameters.

3.1 Stationarity

Before the order of the model can be determined, the time series must be tested for stationarity. A standard statistical test for covariance stationarity is the Augmented Dickey-Fuller (ADF) test which often accounts for the (c)onstant drift and (t)ime trend. The ADF test is a unit root test—the Null hypothesis is that the characteristic polynomial exhibits at least a unit root and hence the data is non-stationary. If the Null can be rejected at a confidence level, α, then the data is stationary. Attempting to fit a time series model to non-stationary data will result in dubious interpretations of the estimated partial autocorrelation function and poor predictions and should therefore be avoided.

3.2 Transformation to Ensure Stationarity

Any trending time series process is non-stationary. Before we can fit an AR(p) model, it is first necessary to transform the original time series into a stationary form. In some instances, it may be possible to simply detrend the time series (a transformation which works in a limited number of cases). However this is rarely full proof. To the potential detriment of the predictive accuracy of the model, we can however systematically difference the original time series one or more times until we arrive at a stationary time series.

To gain insight, let us consider a simple example. Suppose we are given the following linear model with a time trend of the form:

$$y_t = \alpha + \beta t + \epsilon_t,\ \epsilon_t \sim \mathcal{N}(0, 1). \tag{6.39}$$

We first observe that the mean of y_t is time dependent:

$$\mathbb{E}[y_t] = \alpha + \beta t, \tag{6.40}$$

and thus this is model is non-stationary. Instead we can difference the process to give

$$y_t - y_{t-1} = (\alpha + \beta t + \epsilon_t) - (\alpha + \beta(t-1) + \epsilon_{t-1}) = \beta + \epsilon_t - \epsilon_{t-1}, \tag{6.41}$$

and hence the mean and the variance of this difference process are constant and the difference process is stationary:

$$\mathbb{E}[y_t - y_{t-1}] = \beta \tag{6.42}$$

$$\mathbb{E}[(y_t - y_{t-1} - \beta)^2] = 2\sigma^2. \tag{6.43}$$

Any difference process can be written as an ARIMA(p,d,q) process, where here $d = 1$ is the order of differencing to achieve stationarity. There is, in general, no guarantee that first order differencing yields a stationary difference process. One can apply higher order differencing, $d > 1$, to the detriment of recovering the original signal, but one must often resort to non-stationary time series methods such as Kalman filters, Markov-switching models, and advanced neural networks for sequential data covered later in this part of the book.

3.3 Identification

A common approach for determining the order of a $AR(p)$ from a stationary time series is to estimate the partial autocorrelations and determine the largest lag which is significant. Figure 6.1 shows the *partial correlogram*, the plot of the estimated

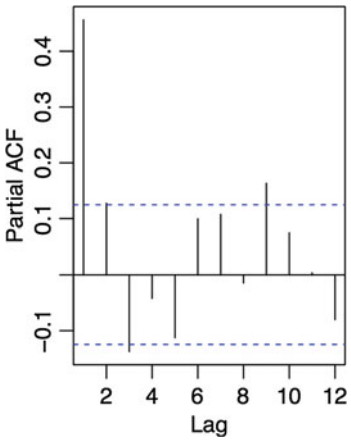


Fig. 6.1 This plot shows the partial correlogram, the plot of the estimated partial autocorrelations against the lag. The solid horizontal lines define the 95% confidence interval. We observe that all but the first lag are approximately within the envelope and hence we may determine the order of the AR(p) model as $p = 1$

partial autocorrelations against the lag. The solid horizontal lines define the 95% confidence interval which can be constructed for each coefficient using

$$\pm 1.96 \times \frac{1}{\sqrt{T}}, \tag{6.44}$$

where T is the number of observations. Note that we have assumed that T is sufficiently large that the autocorrelation coefficients are assumed to be normally distributed with zero mean and standard error of $\frac{1}{\sqrt{T}}$.[6]

We observe that all but the first lag are approximately within the envelope and hence we may determine the order of the AR(p) model as $p = 1$.

The properties of the partial autocorrelation and autocorrelation plots reveal the orders of the AR and MA models. In Fig. 6.1, there is an immediate cut-off in the partial autocorrelation (acf) plot after 1 lag indicating an AR(1) process. Conversely, the location of a sharp cut-off in the estimated autocorrelation function determines the order, q, of a MA process. It is often assumed that the data generation process is a combination of an AR and MA model—referred to as an ARMA(p,q) model.

Information Criterion

While the partial autocorrelation function is useful for determining the AR(p) model order, in many cases there is an undesirable element of subjectively in the choice.

It is often preferable to use the Akaike Information Criteria (AIC) to measure the quality of fit. The AIC is given by

$$AIC = ln(\hat{\sigma}^2) + \frac{2k}{T}, \tag{6.45}$$

[6]This assumption is admitted by the Central Limit Theorem.

where $\hat{\sigma}^2$ is the residual variance (the residual sums of squares divided by the number of observations T) and $k = p + q + 1$ is the total number of parameters estimated. This criterion expresses a bias–variance tradeoff between the first term, the quality of fit, and the second term, a penalty function proportional to the number of parameters. The goal is to select the model which minimizes the AIC by first using maximum likelihood estimation and then adding the penalty term. Adding more parameters to the model reduces the residuals but increases the right-hand term, thus the AIC favors the best fit with the fewest number of parameters.

On the surface, the overall approach has many similarities with regularization in machine learning where the loss function is penalized by a LASSO penalty (L_1 norm of the parameters) or a ridge penalty (L_2 norm of the parameters). However, we emphasize that AIC is *estimated post-hoc*, once the maximum likelihood function is evaluated, whereas in machine learning models, the penalized loss function is directly minimized.

3.4 Model Diagnostics

Once the model is fitted we must assess whether the residual exhibits autocorrelation, suggesting the model is underfitting. The residual of fitted time series model should be white noise. To test for autocorrelation in the residual, Box and Pierce propose the Portmanteau statistic

$$Q^*(m) = T \sum_{l=1}^{m} \hat{\rho}_l^2,$$

as a test statistic for the Null hypothesis

$$H_0 : \hat{\rho}_1 = \cdots = \hat{\rho}_m = 0$$

against the alternative hypothesis

$$H_a : \hat{\rho}_i \neq 0$$

for some $i \in \{1, \ldots, m\}$. $\hat{\rho}_i$ are the sample autocorrelations of the residual.

The Box-Pierce statistic follows an asymptotically chi-squared distribution with m degrees of freedom. The closely related Ljung–Box test statistic increases the power of the test in finite samples:

$$Q(m) = T(T+2) \sum_{l=1}^{m} \frac{\hat{\rho}_l^2}{T-l}. \tag{6.46}$$

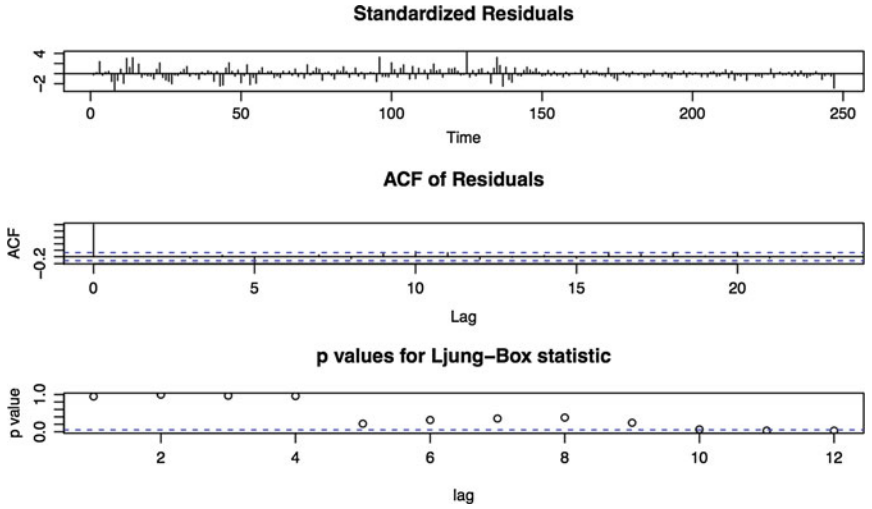


Fig. 6.2 This plot shows the results of applying a Ljung–Box test to the residuals of an AR(p) model. (Top) The standardized residuals are shown against time. (Center) The estimated ACF of the residuals is shown against the lag index. (Bottom) The p-values of the Ljung–Box test statistic are shown against the lag index

This statistic also follows asymptotically a chi-squared distribution with m degrees of freedom. The decision rule is to reject H_0 if $Q(m) > \chi^2_\alpha$ where χ^2_α denotes the $100(1-\alpha)$th percentile of a chi-squared distribution with m degrees of freedom and is the significance level for rejecting H_0.

For a $AR(p)$ model, the Ljung–Box statistic follows asymptotically a chi-squared distribution with $m - p$ degrees of freedom. Figure 6.2 shows the results of applying a Ljung–Box test to the residuals of an AR(p) model. (Top) The standardized residuals are shown against time. (Center) The estimated ACF of the residuals is shown against the lag index. (Bottom) The p-values of the Ljung–Box test statistic are shown against the lag index. The figure shows that if the maximum lag in the model is sufficiently large, then the p-value is small and the Null is rejected in favor of the alternative hypothesis.

Failing the test requires repeating the Box–Jenkins approach until the model no longer under-fits. The only mild safe-guard against over-fitting is the use of AIC for model selection, but in general there is no strong guarantee of avoiding over-fitting as the performance of the model is not assessed out-of-sample in this framework. Assessing the bias variance tradeoff by cross-validation has arisen as the better approach for generalizing model performance. Therefore any model that has been fitted under a Box–Jenkins approach needs to be assessed out-of-sample by time series cross-validation—the topic of the next section.

There are many diagnostic tests which have been developed for time series modeling which we have not discussed here. A small subset has been listed in Table 6.3. The readers should refer to a standard financial econometrics textbook

such as Tsay (2010) for further details of these tests and elaboration on their application to linear models.

4 Prediction

While the Box–Jenkins approach is useful in identifying, fitting, and critiquing models, there is no guarantee that such a model shall exhibit strong predictive properties of course. We seek to predict the value of y_{t+h} given information Ω_t up to and including time t. Producing a forecast is simply a matter of taking the conditional expectation of the data under the model. The h-step ahead forecast from a $AR(p)$ model is given by

$$\hat{y}_{t+h} = \mathbb{E}[y_{t+h} \mid \Omega_t] = \sum_{i=1}^{p} \phi_i \hat{y}_{t+h-i}, \tag{6.47}$$

where $\hat{y}_{t+h} = y_{t+h}, h \leq 0$ and $\mathbb{E}[\epsilon_{t+h} \mid \Omega_t] = 0, h > 0$. Note that conditional expectations of observed variables are not equal to the unconditional expectations. In particular $\mathbb{E}[\epsilon_{t+h} \mid \Omega_t] = \epsilon_t + h,\ h \leq 0$, whereas $\mathbb{E}[\epsilon_{t+h}] = 0$. The quality of the forecast is measured over the forecasting horizon from either the MSE or the MAE.

4.1 Predicting Events

If the output is categorical, rather than continuous, then the ARMA model is used to predict the log-odds ratio of the binary event rather than the conditional expectation of the response. This is analogous to using a logit function as a link in logistic regression.

Other general metrics are also used to assess model accuracy such as a confusion matrix, the F1-score and Receiver Operating Characteristic (ROC) curves. These metrics are not specific to time series data and could be applied to cross-section models to. The following example will illustrate a binary event prediction problem using time series data.

Example 6.1 Predicting Binary Events

Suppose we have conditionally i.i.d. Bernoulli r.v.s X_t with $p_t := \mathbb{P}(X_t = 1 \mid \Omega_t)$ representing a binary event and conditional moments given by

(continued)

Example 6.1 (continued)

- $\mathbb{E}[X_t \mid \Omega] = 0 \cdot (1 - p_t) + 1 \cdot p_t = p_t$
- $\mathbb{V}[X_t \mid \Omega] = p_t(1 - p_t)$

The log-odds ratio shall be assumed to follow an ARMA model,

$$\ln\left(\frac{p_t}{1-p_t}\right) = \phi^{-1}(L)(\mu + \theta(L)\epsilon_t). \tag{6.48}$$

and the category of the model output is determined by a threshold, e.g. $p_t >= 0.5$ corresponds to a positive event. If the number of out-of-sample observations is 24, we can compare the prediction with the observed event and construct a truth table (a.k.a. confusion matrix) as illustrated in Table 6.1.

In this example, the accuracy is $(12 + 2)/24$—the ratio of the sum of the diagonal terms to the set size. The type I (false positive) and type II (false negative) errors, shown by the off-diagonal elements as 8 and 2, respectively.

Table 6.1 The confusion matrix for the above example

	Predicted		
Actual	1	0	Sum
1	12	2	14
0	8	2	10
Sum	20	4	24

In this example, the accuracy is $(12+2)/24$—the ratio of the sum of the diagonal terms to the set size. Of special interest are the type I (false positive) and type II (false negative) errors, shown by the off-diagonal elements as 8 and 2, respectively. In practice, careful consideration must be given as to whether there is equal tolerance for type 1 and type 2 errors.

The significance of the classifier can be estimated from a chi-squared statistic with one degree of freedom under a Null hypothesis that the classifier is a white noise. In general, Chi-squared testing is used to determine whether two variables are independent of one another. In this case, if the Chi-squared statistic is above a given critical threshold value, associated with a significance level, then we can say that the classifier is not white noise.

Let us label the elements of the confusion matrix as in Table 6.2 below. The column and row sums of the confusion matrices and the total number of test samples, m, are also shown.

The chi-squared statistic with one degree of freedom is given by the squared difference of the expected result (i.e., a white noise model where the prediction is independent of the observations) and the model prediction, $\hat{Y}$, relative to the expected result. When normalized by the number of observations, each element of the confusion matrix is the joint probability $[\mathbb{P}(Y, \hat{Y})]_{ij}$. Under a white noise model,

Table 6.2 The confusion matrix of a binary classification is shown together with the column and row sums and the total number of test samples, m

	Predicted		
Actual	1	0	Sum
1	m_{11}	m_{12}	$m_{1,}$
0	m_{21}	m_{22}	$m_{1,}$
Sum	$m_{,1}$	$m_{,2}$	m

the observed outcome, Y, and the predicted outcome, $\hat{Y}$, are independent and so $[\mathbb{P}(Y, \hat{Y})]_{ij} = [\mathbb{P}(Y)]_i[\mathbb{P}(\hat{Y})]_j$ which is the ith row sum, $m_{i,}$, multiplied by the jth column sum, $m_{,j}$, divided by m. Since $m_{i,j}$ is based on the model prediction, the chi-squared statistic is thus

$$\chi^2 = \sum_{i=1}^{2}\sum_{j=1}^{2} \frac{(m_{ij} - m_{i,}m_{,j}/m)^2}{m_{i,}m_{,j}/m}. \tag{6.49}$$

Returning to the example above, the chi-squared statistic is

$$\begin{aligned}\chi^2 &= (12 - (14 \times 20)/24)^2/(14 \times 20)/24 \\ &+ (2 - (14 \times 4)/24)^2/(14 \times 4)/24 \\ &+ (8 - (10 \times 20)/24)^2/(10 \times 20)/24 \\ &+ (2 - (10 \times 4)/24)^2/(10 \times 4)/24 \\ &= 0.231.\end{aligned}$$

This value is far below the threshold value of 6.635 for a chi-squared statistic with one degree of freedom to be significant. Thus we cannot reject the Null hypothesis. The predicted model is not sufficiently indistinguishable from white noise.

The example classification model shown above used a threshold of $p_t >= 0.5$ to classify an event as positive. This choice of threshold is intuitive but arbitrary. How can we measure the performance of a classifier for a range of thresholds?

A ROC-Curve contains information about all possible thresholds. The ROC-Curve plots *true positive rates* against *false positive rates*, where these terms are defined as follows:

- *True Positive Rate (TPR)* is $TP/(TP + FN)$: fraction of positive samples which the classifier correctly identified. This is also known as *Recall* or *Sensitivity*. Using the confusion matrix in Table 6.1, the TPR=$12/(12 + 2) = 6/7$.
- *False Positive Rate (FPR)* is $FP/(FP+TN)$: fraction of positive samples which the classifier misidentified. In the example confusion matrix, the FPR=$8/(8 + 2) = 4/5$.
- *Precision* is $TP/(TP + FP)$: fraction of samples that were positive from the group that the classifier predicted to be positive. From the example confusion matrix, the precision is $12/(12 + 8) = 3/5$.

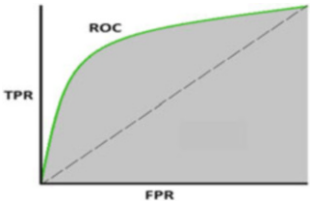


Fig. 6.3 The ROC curve for an example model shown by the green line

Each point in a ROC curve is a (TPR, FPR) pair for a particular choice of the threshold in the classifier. The straight dashed black line in Fig. 6.3 represents a random model. The green line shows the ROC curve of the model—importantly it is should always be above the line. The perfect model would exhibit a TPR of unity for all FPRs, so that there is no area above the curve.

The advantage of this performance measure is that it is robust to class imbalance, e.g. rare positive events. This is not true of classification accuracy which leads to misinterpretation of the quality of the fit when the data is imbalanced. For example, a constant model $\hat{Y} = f(X) = 1$ would be x% accurate if the data consists of x% positive events. Additional related metrics can be derived. Common ones include Area Under the Curve (AUC), which is the area under the green line in Fig. 6.3.

The *F1-score* is the harmonic mean of the precision and recall and is also frequently used. The F1-score reaches its best value at unity and worst at zero and is given by $F1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. From the example above $F_1 = \frac{2 \times 3/5 \times 6/7}{3/5 + 6/7} = 0.706$.

4.2 Time Series Cross-Validation

Cross-validation—the method of hyperparameter tuning by rotating through K folds (or subsets) of training-test data—differs for time series data. In prediction models over time series data, no future observations can be used in the training set. Instead, a sliding window must be used to train and predict out-of-sample over multiple repetitions to allow for parameter tuning as illustrated in Fig. 6.4. One frequent challenge is whether to fix the length of the window or allow it to "telescope" by including the ever extending history of observations as the window is "walked forward." In general, the latter has the advantage of including more observations in the training set but can lead to difficulty in interpreting the confidence of the parameters, due to the loss of control of the sample size.

5 Principal Component Analysis

The final section in this chapter approaches data modeling from quite a different perspective, with the goal being to reduce the dimension of multivariate time series

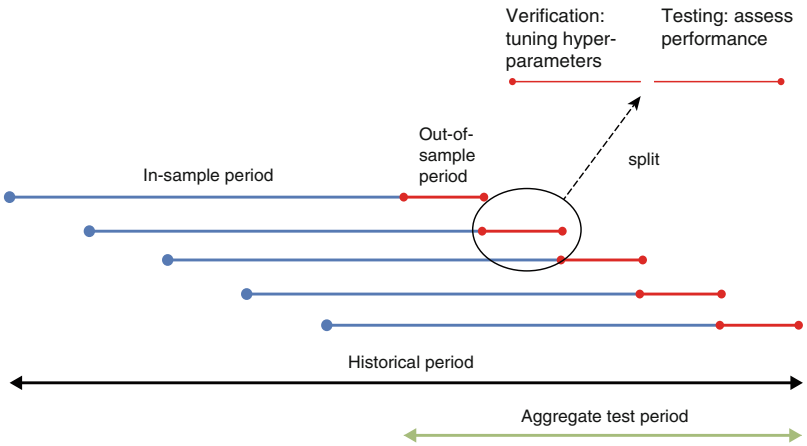


Fig. 6.4 Times series cross-validation, also referred to as "walk forward optimization," is used instead of standard cross-validation for cross-sectional data to preserve the ordering of observations in time series data. This experimental design avoids look-ahead bias in the fitted model which occurs when one or more observations in the training set are from the future

data. The approach is widely used in finance, especially when the dimensionality of the data presents barriers to computational tractability or practical risk management and trading challenges such as hedging exposure to market risk factors. For example, it may be advantageous to monitor a few risk factors in a large portfolio rather than each instrument. Moreover such factors should provide economic insight into the behavior of the financial markets and be actionable from an investment management perspective.

Formally, let $\{\mathbf{y}_i\}_{i=1}^N$ be a set of N observation vectors, each of dimension n. We assume that $n \leq N$. Let $\mathrm{Y} \in \mathbb{R}^{n \times N}$ be a matrix whose columns are $\{\mathbf{y}_i\}_{i=1}^N$,

$$\mathrm{Y} = \begin{bmatrix} | & & | \\ \mathbf{y}_1 & \cdots & \mathbf{y}_N \\ | & & | \end{bmatrix}.$$

The element-wise average of the N observations is an n dimensional signal which may be written as:

$$\bar{\mathbf{y}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{y}_i = \frac{1}{N}\mathrm{Y}\mathbb{1}_N,$$

where $\mathbb{1}_N \in \mathbb{R}^{N \times 1}$ is a column vector of all-ones. Denote Y_0 as a matrix whose columns are the demeaned observations (we center each observation $\mathbf{y}_i$ by subtracting $\bar{\mathbf{y}}$ from it):

$$\mathrm{Y}_0 = \mathrm{Y} - \bar{\mathbf{y}}\mathbb{1}_N^T.$$

Projection

A linear projection from $\mathbb{R}^m$ to $\mathbb{R}^n$ is a linear transformation of a finite dimensional vector given by a matrix multiplication:

$$\mathbf{x}_i = \mathrm{W}^T\mathbf{y}_i,$$

where $\mathbf{y}_i \in \mathbb{R}^n$, $\mathbf{x}_i \in \mathbb{R}^m$, and $\mathrm{W} \in \mathbb{R}^{n\times m}$. Each element j in the vector $\mathbf{x}_i$ is an inner product between $\mathbf{y}_i$ and the j-th column of W, which we denote by $\mathbf{w}_j$.

Let $\mathrm{X} \in \mathbb{R}^{m\times N}$ be a matrix whose columns are the set of N vectors of transformed observations, let $\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i = \frac{1}{N}\mathrm{X}\mathbb{1}_N$ be the element-wise average, and $\mathrm{X}_0 = \mathrm{X} - \bar{\mathbf{x}}\mathbb{1}_N^T$ the demeaned matrix. Clearly, $\mathrm{X} = \mathrm{W}^T\mathrm{Y}$ and $\mathrm{X}_0 = \mathrm{W}^T\mathrm{Y}_0$.

5.1 Principal Component Projection

When the matrix W^T represents the transformation that applies principal component analysis, we denote $\mathrm{W} = \mathrm{P}$, and the columns of the orthonormal matrix,[7] P, denoted $\{\mathbf{p}_j\}_{j=1}^n$, are referred to as *loading vectors*. The transformed vectors $\{\mathbf{x}_i\}_{i=1}^N$ are referred to as *principal components* or *scores*.

The first loading vector is defined as the unit vector with which the inner products of the observations have the greatest variance:

$$\mathbf{p}_1 = \max_{\mathbf{w}_1} \mathbf{w}_1^T\mathrm{Y}_0\mathrm{Y}_0^T\mathbf{w}_1 \; s.t. \; \mathbf{w}_1^T\mathbf{w}_1 = 1. \tag{6.50}$$

The solution to Eq. 6.50 is known to be the eigenvector of the sample covariance matrix $\mathrm{Y}_0\mathrm{Y}_0^T$ corresponding to its largest eigenvalue.[8]

Next, $\mathbf{p}_2$ is the unit vector which has the largest variance of inner products between it and the observations after removing the orthogonal projections of the observations onto $\mathbf{p}_1$. It may be found by solving:

$$\mathbf{p}_2 = \max_{\mathbf{w}_2} \mathbf{w}_2^T\left(\mathrm{Y}_0 - \mathbf{p}_1\mathbf{p}_1^T\mathrm{Y}_0\right)\left(\mathrm{Y}_0 - \mathbf{p}_1\mathbf{p}_1^T\mathrm{Y}_0\right)^T\mathbf{w}_2 \; s.t. \; \mathbf{w}_2^T\mathbf{w}_2 = 1. \tag{6.51}$$

[7] That is, $\mathrm{P}^{-1} = \mathrm{P}^T$.

[8] We normalize the eigenvector and disregard its sign.

The solution to Eq. 6.51 is known to be the eigenvector corresponding to the largest eigenvalue under the constraint that it is not collinear with $\mathbf{p}_1$. Similarly, the remaining loading vectors are equal to the remaining eigenvectors of $\mathrm{Y}_0\mathrm{Y}_0^T$ corresponding to descending eigenvalues.

The eigenvalues of $\mathrm{Y}_0\mathrm{Y}_0^T$, which is a positive semi-definite matrix, are non-negative. They are not necessarily distinct, but since it is a symmetric matrix it has n eigenvectors that are all orthogonal, and it is always diagonalizable. Thus, the matrix P may be computed by diagonalizing the covariance matrix:

$$\mathrm{Y}_0\mathrm{Y}_0^T = \mathrm{P}\Lambda\mathrm{P}^{-1} = \mathrm{P}\Lambda\mathrm{P}^T,$$

where $\Lambda = \mathrm{X}_0\mathrm{X}_0^T$ is a diagonal matrix whose diagonal elements $\{\lambda_i\}_{i=1}^n$ are sorted in descending order.

The transformation back to the observations is $\mathrm{Y} = \mathrm{PX}$. The fact that the covariance matrix of X is diagonal means that PCA is a decorrelation transformation and is often used to denoise data.

5.2 Dimensionality Reduction

PCA is often used as a method for dimensionality reduction, the process of reducing the number of variables in a model in order to avoid the curse of dimensionality. PCA gives the first m principal components ($m < n$) by applying the truncated transformation

$$\mathrm{X}_m = \mathrm{P}_m^T\mathrm{Y},$$

where each column of $\mathrm{X}_m \in \mathbb{R}^{m\times N}$ is a vector whose elements are the first m principal components, and P_m is a matrix whose columns are the first m loading vectors,

$$\mathrm{P}_m = \begin{bmatrix} | & & | \\ \mathbf{p}_1 & \cdots & \mathbf{p}_m \\ | & & | \end{bmatrix} \in \mathbb{R}^{n\times m}.$$

Intuitively, by keeping only m principal components, we are losing information, and we minimize this loss of information by maximizing their variances.

An important concept in measuring the amount of information lost is the total reconstruction error $\|Y - \hat{Y}\|_F$, where F denotes the Frobenius matrix norm. P_m is also a solution to the minimum total squared reconstruction

$$\min_{\mathrm{W}\in\mathbb{R}^{n\times m}} \left\|\mathrm{Y}_0 - \mathrm{W}\mathrm{W}^T\mathrm{Y}_0\right\|_F^2 \; s.t.\; \mathrm{W}^T\mathrm{W} = \mathrm{I}_{m\times m}. \tag{6.52}$$

The m leading loading vectors form an orthonormal basis which spans the m dimensional subspace onto which the projections of the demeaned observations have the minimum squared difference from the original demeaned observations.

In other words, P_m compresses each demeaned vector of length n into a vector of length m (where $m \leq n$) in such a way that minimizes the sum of total squared reconstruction errors.

The minimizer of Eq. 6.52 is not unique: $\mathrm{W} = \mathrm{P}_m\mathrm{Q}$ is also a solution, where $\mathrm{Q} \in \mathbb{R}^{m\times m}$ is any orthogonal matrix, $\mathrm{Q}^T = \mathrm{Q}^{-1}$. Multiplying P_m from the right by Q transforms the first m loading vectors into a different orthonormal basis for the same subspace.

6 Summary

This chapter has reviewed foundational material in time series analysis and econometrics. Such material is not intended to substitute more comprehensive and formal treatment of methodology, but rather provide enough background for Chap. 8 where we shall develop neural networks analogues. We have covered the following objectives:

- Explain and analyze linear autoregressive models;
- Understand the classical approaches to identifying, fitting, and diagnosing autoregressive models;
- Apply simple heteroscedastic regression techniques to time series data;
- Understand how exponential smoothing can be used to predict and filter time series; and
- Project multivariate time series data onto lower dimensional spaces with principal component analysis.

It is worth noting that in industrial applications the need to forecast more than a few steps ahead often arises. For example, in algorithmic trading and electronic market making, one needs to forecast far enough into the future, so as to make the forecast economically realizable either through passive trading (skewing of the price) or through aggressive placement of trading orders. This economic realization of the trading signals takes time, whose actual duration is dependent on the frequency of trading.

We should also note that in practice linear regressions predicting the difference between a future and current price, taking as inputs various moving averages, are often used in preference to parametric models, such as GARCH. These linear regressions are often cumbersome, taking as inputs hundreds or thousands of variables.

7 Exercises

Exercise 6.1
Calculate the mean, variance, and autocorrelation function (acf) of the following zero-mean AR(1) process:

$$y_t = \phi_1 y_{t-1} + \epsilon_t,$$

where $\phi_1 = 0.5$. Determine whether the process is stationary by computing the root of the characteristic equation $\Phi(z) = 0$.

Exercise 6.2
You have estimated the following ARMA(1,1) model for some time series data

$$y_t = 0.036 + 0.69 y_{t-1} + 0.42 u_{t-1} + u_t,$$

where you are given the data at time $t-1$, $y_{t-1} = 3.4$ and $\hat{u}_{t-1} = -1.3$. Obtain the forecasts for the series y for times $t, t+1, t+2$ using the estimated ARMA model.

If the actual values for the series are $-0.032, 0.961, 0.203$ for $t, t+1, t+2$, calculate the out-of-sample Mean Squared Error (MSE) and Mean Absolute Error (MAE).

Exercise 6.3
Derive the mean, variance, and autocorrelation function (ACF) of a zero mean MA(1) process.

Exercise 6.4
Consider the following log-GARCH(1,1) model with a constant for the mean equation

$$y_t = \mu + u_t,\ u_t \sim N(0, \sigma_t^2)$$
$$ln(\sigma_t^2) = \alpha_0 + \alpha_1 u_{t-1}^2 + \beta_1 ln\sigma_{t-1}^2$$

– What are the advantages of a log-GARCH model over a standard GARCH model?

– Estimate the unconditional variance of y_t for the values $\alpha_0 = 0.01, \alpha_1 = 0.1, \beta_1 = 0.3$.
– Derive an algebraic expression relating the conditional variance with the unconditional variance.
– Calculate the half-life of the model and sketch the forecasted volatility.

Exercise 6.5
Consider the simple moving average (SMA)

$$S_t = \frac{X_t + X_{t-1} + X_{t+2} + \ldots + X_{t-N+1}}{N},$$

and the exponential moving average (EMA), given by $E_1 = X_1$ and, for $t \geq 2$,

$$E_t = \alpha X_t + (1 - \alpha) E_{t-1},$$

where N is the time horizon of the SMA and the coefficient α represents the degree of weighting decrease of the EMA, a constant smoothing factor between 0 and 1. A higher α discounts older observations faster.

a. Suppose that, when computing the EMA, we stop after k terms, instead of going after the initial value. What fraction of the total weight is obtained?
b. Suppose that we require 99.9% of the weight. What k do we require?
c. Show that, by picking $\alpha = 2/(N + 1)$, one achieves the same center of mass in the EMA as in the SMA with the time horizon N.
d. Suppose that we have set $\alpha = 2/(N+1)$. Show that the first N points in an EMA represent about 87.48% of the total weight.

Exercise 6.6
Suppose that, for the sequence of random variables $\{y_t\}_{t=0}^{\infty}$ the following model holds:

$$y_t = \mu + \phi y_{t-1} + \epsilon_t, \quad |\phi| \leq 1, \quad \epsilon_t \sim \text{i.i.d.}(0, \sigma^2).$$

Derive the conditional expectation $\mathbb{E}[y_t \mid y_0]$ and the conditional variance $\text{Var}[y_t \mid y_0]$.

Appendix

Hypothesis Tests

Table 6.3 A short summary of some of the most useful diagnostic tests for time series modeling in finance

Name	Description
Chi-squared test	Used to determine whether the confusion matrix of a classifier is statistically significant, or merely white noise
t-test	Used to determine whether the output of two separate regression models are statistically different on i.i.d. data
Mariano-Diebold test	Used to determine whether the output of two separate time series models are statistically different
ARCH test	The ARCH Engle's test is constructed based on the property that if the residuals are heteroscedastic, the squared residuals are autocorrelated. The Ljung–Box test is then applied to the squared residuals
Portmanteau test	A general test for whether the error in a time series model is auto-correlated
	Example tests include the Box-Ljung and the Box-Pierce test

Python Notebooks

Please see the code folder of Chap. 6 for e.g., implementations of ARIMA models applied to time series prediction. An example, applying PCA to decompose stock prices is also provided in this folder. Further details of these notebooks are included in the `README.md` file for Chap. 6.

Reference

Tsay, R. S. (2010). *Analysis of financial time series* (3rd ed.). Wiley.

Chapter 7
Probabilistic Sequence Modeling

This chapter presents a powerful class of probabilistic models for financial data. Many of these models overcome some of the severe stationarity limitations of the frequentist models in the previous chapters. The fitting procedure demonstrated is also different—the use of Kalman filtering algorithms for state-space models rather than maximum likelihood estimation or Bayesian inference. Simple examples of hidden Markov models and particle filters in finance and various algorithms are presented.

1 Introduction

So far we have seen how sequences can be modeled using autoregressive processes, moving averages, GARCH, and similar methods. There exists another school of thought, which gave rise to hidden Markov models, Baum–Welch and Viterbi algorithms, Kalman and particle filters.

In this school of thought, one assumes the existence of a certain latent process (say X), which evolves over time (so we may write X_t). This unobservable, latent process drives another, observable process (say Y_t), which we may observe either at all times or at some subset of times.

The evolution of the latent process X_t, as well as the dependence of the observable process Y_t on X_t, may be driven by random factors. We therefore talk about a *stochastic* or *probabilistic* model. We also refer to such a model as a state-space model. The state-space model consists in a description of the evolution of the latent state over time and the dependence of the observables on the latent state.

We have already seen probabilistic methods presented in Chaps. 2 and 3. These methods primarily assume that the data is i.i.d. On the other hand, the time series methods presented in the previous chapter are designed for time series data but are not probabilistic. This chapter shall build on these earlier chapters by considering a

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_7

powerful class of models for financial data. Many of these models overcome some of the severe stationarity limitations of the frequentist models in the previous chapters. The fitting procedure is also different—we will see the use of Kalman filtering algorithms for state-space models rather than maximum likelihood estimation or Bayesian inference.

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Formulate hidden Markov models (HMMs) for probabilistic modeling over hidden states;
- Gain familiarity with the Baum–Welch algorithmic for fitting HMMs to time series data;
- Use the Viterbi algorithm to find the most likely path;
- Gain familiarity with state-space models and the application of Kalman filters to fit them; and
- Apply particle filters to financial time series.

2 Hidden Markov Modeling

A hidden Markov model (HMM) is a probabilistic model representing probability distributions over *sequences of observations*. HMMs are the simplest "dynamic" Bayesian network[1] and have proven a powerful model in many applied fields including finance. So far in this book, we have largely considered only i.i.d. observations.[2] Of course, financial modeling is often seated in a Markovian setting where observations depend on, and only on, the previous observation.

We shall briefly review HMMs in passing as they encapsulate important ideas in probabilistic modeling. In particular, they provide intuition for understanding hidden variables and switching. In the next chapter we shall see the examples of switching in dynamic recurrent neural networks, such as GRUs and LSTMs, which use gating. However, this gating is an implicit modeling step and cannot be controlled explicitly as may be needed for regime switching in finance.

Let us assume at time t that the discrete state, s_t, is hidden from the observer. Furthermore, we shall assume that the hidden state is a Markov process. Note this setup differs from mixture models which treat the hidden variable as i.i.d. The time t observation, y_t, is assumed to be independent of the state at all other times. By the Markovian property, the joint distribution of the sequence of states, $\mathbf{s} := \{s_t\}_{i=1}^{T}$,

[1] Dynamic Bayesian networks models are a graphical model used to model dynamic processes through hidden state evolution.

[2] With the exception of heteroscedastic modeling in Chap. 6.

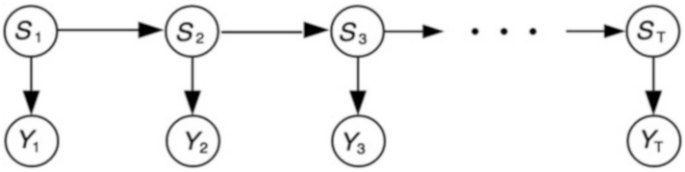


Fig. 7.1 This figure shows the probabilistic graph representing the conditional dependence relations between the observed and the hidden variables in the HMM

and sequence of observations, $\mathbf{y} = \{y_t\}_{t=1}^{T}$ is given by the product of transition probability densities $p(s_t \mid s_{t-1})$ and emission probability densities, $p(y_t \mid s_t)$:

$$p(\mathbf{s}, \mathbf{y}) = p(s_1)p(y_1 \mid s_1) \prod_{t=2}^{T} p(s_t \mid s_{t-1})p(y_t \mid s_t). \tag{7.1}$$

Figure 7.1 shows the Bayesian network representing the conditional dependence relations between the observed and the hidden variables in the HMM. The conditional dependence relationships define the edges of a graph between parent nodes, Y_t, and child nodes S_t.

Example 7.1 Bull or Bear Market?

Suppose that the market is either in a Bear or Bull market regime represented by $s = 0$ or $s = 1$, respectively. Such states or regimes are assumed to be hidden. Over each period, the market is observed to go up or down and represented by $y = -1$ or $y = 1$. Assume that the emission probability matrix—the conditional dependency matrix between observed and hidden variables—is independent of time and given by

$$\mathbb{P}(y_t = y \mid s_t = s) = \begin{bmatrix} y/\,s & 0 & 1 \\ -1 & 0.8 & 0.2 \\ 1 & 0.2 & 0.8 \end{bmatrix}, \tag{7.2}$$

and the transition probability density matrix for the Markov process $\{S_t\}$ is given by

$$A = \begin{bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{bmatrix}, \quad [A]_{ij} := \mathbb{P}(S_t = s_i \mid S_{t-1} = s_j). \tag{7.3}$$

Given the observed sequence $\{-1, 1, 1\}$ (i.e., $T = 3$), we can compute the probability of a realization of the hidden state sequence $\{1, 0, 0\}$ using Eq. 7.1. Assuming that $\mathbb{P}(s_1 = 0) = \mathbb{P}(s_1 = 1) = \frac{1}{2}$, the computation is

(continued)

Example 7.1 (continued)

$$\begin{aligned}\mathbb{P}(\mathbf{s},\mathbf{y}) &= \mathbb{P}(s_1=1)\mathbb{P}(y_1=-1\mid s_1=1)\mathbb{P}(s_2=0\mid s_1=1)\mathbb{P}(y_2=1\mid s_2=0)\\ &\quad \mathbb{P}(s_3=0\mid s_2=0)\mathbb{P}(y_3=1\mid s_3=0),\\ &= 0.5\cdot 0.2\cdot 0.1\cdot 0.2\cdot 0.9\cdot 0.2 = 0.00036.\end{aligned}$$

We first introduce the so-called *forward* and *backward* quantities, respectively, defined for all states $s_t \in \{1,\ldots,K\}$ and over all times

$$F_t(s) := \mathbb{P}(s_t = s, \mathbf{y}_{1:t}) \quad \text{and} \quad B_t(s) := p(\mathbf{y}_{t+1:T} \mid s_t = s) \tag{7.4}$$

with the convention that $B_T(s) = 1$. For all $t \in \{1,\ldots,T\}$ and for all $r, s \in \{1,\ldots,K\}$ we have

$$\mathbb{P}(s_t = s, \mathbf{y}) = F_t(s)B_t(s), \tag{7.5}$$

and combining the forward and backward quantities gives

$$\mathbb{P}(s_{t-1} = r, s_t = s, \mathbf{y}) = F_{t-1}(r)\mathbb{P}(s_t = s \mid s_{t-1} = r)p(y_t \mid s_t = s)B_t(s). \tag{7.6}$$

The forward–backward algorithm, also known as the *Baum–Welch* algorithm, is an *unsupervised* learning algorithm for fitting HMMs which belongs to the class of EM algorithms.

2.1 The Viterbi Algorithm

In addition to finding the probability of the realization of a particularly hidden state sequence, we may also seek the most likely sequence realization. This sequence can be estimated using the Viterbi algorithm.

Suppose once again that we observe a sequence of T *observations*,

$$\mathbf{y} = \{y_1,\ldots,y_T\}.$$

However, for each $1 \le t \le T$, $y_t \in O$, where $O = \{o_1, o_2, \ldots, o_N\}$, $N \in \mathbb{N}$, is now in some *observation space*.

We suppose that, for each $1 \le t \le T$, the observation y_t is driven by a *(hidden) state* $s_t \in \mathcal{S}$, where $\mathcal{S} := \{\int_1,\ldots,\int_K\}$, $K \in \mathbb{N}$, is some *state space*. For example, y_t might be the credit rating of a corporate bond and s_t might indicate some latent variable such as the overall health of the relevant industry sector.

Given $\mathbf{y}$, what is the most likely sequence of hidden states,

$$\mathbf{x} = \{x_1, x_2, \ldots, x_T\}?$$

To answer this question, we need to introduce a few more constructs. First, the set of *initial probabilities* must be given:

$$\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_K\},$$

so that π_i is the probability that $s_1 = \int_i$, $1 \leq i \leq K$.

We also need to specify the *transition matrix* $A \in \mathbb{R}^{K \times K}$, such that the element A_{ij}, $1 \leq i, j \leq K$ is the transition probability of transitioning from state $\int_i$ to state $\int_j$.

Finally, we need the *emission matrix* $B \in \mathbb{R}^{K \times N}$, such that the element B_{ij}, $1 \leq i \leq K$, $1 \leq j \leq N$ is the probability of observing o_j from state $\int_i$.

Let us now consider a simple example to fix ideas.

Example 7.2 The Crooked Dealer

A dealer has two coins, a fair coin, with $\mathbb{P}(\text{Heads}) = \frac{1}{2}$, and a loaded coin, with $\mathbb{P}(\text{Heads}) = \frac{4}{5}$. The dealer starts with the fair coin with probability $\frac{3}{5}$. The dealer then tosses the coin several times. After each toss, there is a $\frac{2}{5}$ probability of a switch to the other coin. The observed sequence is Heads, Tails, Heads, Tails, Heads, Heads, Heads, Tails, Heads.

In this case, the state space and observation space are, respectively,

$$\mathcal{S} = \{\int_1 = \text{Fair}, \int_2 = \text{Loaded}\}, \quad O = \{o_1 = \text{Heads}, o_2 = \text{Tails}\},$$

with initial probabilities

$$\boldsymbol{\pi} = \{\pi_1 = 0.6, \pi_2 = 0.4\},$$

transition probabilities

$$A = \begin{pmatrix} 0.6 & 0.4 \\ 0.4 & 0.6 \end{pmatrix},$$

and the emission matrix is

$$B = \begin{pmatrix} 0.5 & 0.5 \\ 0.8 & 0.2 \end{pmatrix}.$$

Given the sequence of observations

$$\mathbf{y} = (\text{Heads}, \text{Tails}, \text{Heads}, \text{Tails}, \text{Heads}, \text{Heads}, \text{Heads}, \text{Tails}, \text{Heads}),$$

we would like to find the most likely sequence of hidden states, $\mathbf{s} = \{s_1, \ldots, s_T\}$, i.e., determine which of the two coins generated which of the coin tosses.

One way to answer this question is by applying the *Viterbi algorithm* as detailed in the notebook `Viterbi.ipynb`. We note that the most likely state sequence $\mathbf{s}$, which produces the observation sequence $\mathbf{y} = \{y_1, \ldots, y_T\}$, satisfies the recurrence relations

$$V_{1,k} = \mathbb{P}(y_1 \mid s_1 = \int_k) \cdot \pi_k,$$
$$V_{t,k} = \max_{1 \le i \le K} \left(\mathbb{P}(y_t \mid s_t = \int_k) \cdot A_{ik} \cdot V_{t-1,i} \right),$$

where $V_{t,k}$ is the probability of the most probable state sequence $\{s_1, \ldots, s_t\}$ such that $s_t = \int_k$,

$$V_{t,k} = \mathbb{P}(s_1, \ldots, s_t, y_1, \ldots, y_t \mid s_t = \int_k).$$

The actual *Viterbi path* can be obtained by, at each step, keeping track of which state index i was used in the second equation. Let $\xi(k, t)$ be the function that returns the value of i that was used to compute $V_{t,k}$ if $t > 1$, or k if $t = 1$. Then

$$s_T = \int_{\arg \max_{1 \le i \le K} (V_{T,k})},$$
$$s_{t-1} = \int_{\xi(s_t, t)}.$$

We leave the application of the Viterbi algorithm to our example as an exercise for the reader.

Note that the Viterbi algorithm determines the most likely complete sequence of hidden states given the sequence of observations and the model specification, including the *known* transition and emission matrices. If these matrices are known, there is no reason to use the Baum–Welch algorithm. If they are unknown, then the Baum–Welch algorithm must be used.

2.1.1 Filtering and Smoothing with HMMs

Financial data is typically noisy and we need techniques which can extract the signal from the noise. There are many techniques for reducing the noise. Filtering is a general term for extracting information from a noisy signal. Smoothing is a particular kind of filtering in which low-frequency components are passed and high-frequency components are attenuated. Filtering and smoothing produce distributions of states at each time step. Whereas maximum likelihood estimation chooses the state with the highest probability at the "best" estimate at each time step, this may not lead to the best path in HMMs. We have seen that the Baum–Welch algorithm can be deployed to find the optimal state trajectory, not just the optimal sequence of "best" states.

2.2 State-Space Models

HMMs belong in the same class as linear Gaussian state-space models. These are known as "Kalman filters" which are *continuous* latent state analogues of HMMs. Note that we have already seen examples of continuous state-space models, which are not necessarily Gaussian, in our exposition on RNNs in Chap. 8.

The state transition probability $p(s_t \mid s_{t-1})$ can be decomposed into deterministic and noise:

$$s_t = F_t(s_{t-1}) + \epsilon_t, \tag{7.7}$$

for some deterministic function and ϵ_t is zero-mean i.i.d. noise. Similarly, the emission probability $p(y_t \mid s_t)$ can be decomposed as:

$$y_t = G_t(s_t) + \xi_t, \tag{7.8}$$

with zero-mean i.i.d. observation noise. If the functions F_t and G_t are linear and time independent, then we have

$$s_t = As_{t-1} + \epsilon_t, \tag{7.9}$$

$$y_t = Cs_t + \xi_t, \tag{7.10}$$

where A is the state transition matrix and C is the observation matrix. For completeness, we contrast the Kalman filter with a univariate RNN, as described in Chap. 8. When the observations are predictors, x_t, and the hidden variables are s_t we have

$$s_t = F(s_{t-1}, y_t) := \sigma(As_{t-1} + By_t), \tag{7.11}$$

$$y_t = Cs_t + \xi_t, \tag{7.12}$$

where we have ignored the bias terms for simplicity. Hence, the RNN state equation differs from the Kalman filter in that (i) it is a non-linear function of both the previous state and the observation; and (ii) it is noise-free.

3 Particle Filtering

A Kalman filter maintains its state as moments of the multivariate Gaussian distribution, $\mathcal{N}(\mathbf{m}, \mathbf{P})$. This approach is appropriate when the state is Gaussian, or when the true distribution can be closely approximated by the Gaussian. What if the distribution is, for example, bimodal?

Arguably the simplest way to approximate more or less any distribution, including a bimodal distribution, is by data points sampled from that distribution. We refer to those data points as "particles."

The more particles we have, the more closely we can approximate the target distribution. The approximate, empirical distribution is then given by the histogram. Note that the particles need not be univariate, as in our example. They may be multivariate if we are approximating a multivariate distribution. Also, in our example the particles all have the same weight. More generally, we may consider weighted particles, whose weights are unequal.

This setup gives rise to the family of algorithms known as *particle filtering* algorithms (Gordon et al. 1993; Kitagawa 1993). One of the most common of them is the *Sequential Importance Resampling (SIR)* algorithm:

3.1 Sequential Importance Resampling (SIR)

a. *Initialization step*: At time $t = 0$, draw M i.i.d. samples from the initial distribution τ_0. Also, initialize M normalized (to 1) weights to an identical value $\frac{1}{M}$. For $i = 1, \ldots, M$, the samples will be denoted $\hat{\mathbf{x}}_{0\,|\,0}^{(i)}$ and the normalized weights $\lambda_0^{(i)}$.

b. *Recursive step*: At time $t = 1, \ldots, T$, let $(\hat{\mathbf{x}}_{t-1\,|\,t-1}^{(i)})_{i=1,\ldots,M}$ be the particles generated at time $t-1$.

   – *Importance sampling*: For $i = 1, \ldots M$, sample $\hat{\mathbf{x}}_{t\,|\,t-1}^{(i)}$ from the Markov transition kernel $\tau_t(\cdot \,|\, \hat{\mathbf{x}}_{t-1\,|\,t-1}^{(i)})$. For $i = 1, \ldots, M$, use the observation density to compute the non-normalized weights

$$\omega_t^{(i)} := \lambda_{t-1}^{(i)} \cdot p(\mathbf{y}_t \,|\, \hat{\mathbf{x}}_{t\,|\,t-1}^{(i)})$$

   and the values of the normalized weights before resampling ("br")

$$^{\mathrm{br}}\lambda_t^{(i)} := \frac{\omega_t^{(i)}}{\sum_{k=1}^{M} \omega_t^{(k)}}.$$

   – *Resampling* (or *selection*): For $i = 1, \ldots, M$, use an appropriate resampling algorithm (such as *multinomial resampling*—see below) to sample $\mathbf{x}_{t\,|\,t}^{(i)}$ from the mixture

$$\sum_{k=1}^{M} {}^{\mathrm{br}}\lambda_t^{(k)} \delta(\mathbf{x}_t - \mathbf{x}_{t\,|\,t-1}^{(k)}),$$

where $\delta(\cdot)$ denotes the Dirac delta generalized function, and set the normalized weights after resampling, $\lambda_t^{(i)}$, appropriately (for most common resampling algorithms this means $\lambda_t^{(i)} := \frac{1}{M}$).

Informally, SIR shares some of the characteristics of genetic algorithms; based on the likelihoods $p(\mathbf{y}_t \mid \hat{\mathbf{x}}_{t\,|\,t-1}^{(i)})$, we increase the weights of the more "successful" particles, allowing them to "thrive" at the resampling step.

The resampling step was introduced to avoid the degeneration of the particles, with all the weight concentrating on a single point. The most common resampling scheme is the so-called *multinomial resampling* which we now review.

3.2 Multinomial Resampling

Notice, from above, that we are using with the *normalized* weights computed before resampling, ${}^{\mathrm{br}}\lambda_t^{(1)}, \ldots, {}^{\mathrm{br}}\lambda_t^{(M)}$:

a. For $i = 1, \ldots, M$, compute the cumulative sums

$$ {}^{\mathrm{br}}\Lambda_t^{(i)} = \sum_{k=1}^{i} {}^{\mathrm{br}}\lambda_t^{(k)}, $$

so that, by construction, ${}^{\mathrm{br}}\Lambda_t^{(M)} = 1$.

b. Generate M random samples from $\mathcal{U}(0, 1)$, $u_1, u_2, \ldots, u_M$.

c. For each $i = 1, \ldots, M$, choose the particle $\hat{\mathbf{x}}_{t\,|\,t}^{(i)} = \hat{\mathbf{x}}_{t\,|\,t-1}^{(j)}$ with $j \in \{1, 2, \ldots, M-1\}$ such that $u_i \in \left[{}^{\mathrm{br}}\Lambda_t^{(j)}, {}^{\mathrm{br}}\Lambda_t^{(j+1)} \right]$.

Thus we end up with M new particles (*children*), $\mathbf{x}_{t\,|\,t}^{(1)}, \ldots, \mathbf{x}_{t\,|\,t}^{(M)}$ sampled from the existing set $\mathbf{x}_{t\,|\,t-1}^{(1)}, \ldots, \mathbf{x}_{t\,|\,t-1}^{(M)}$, so that some of the existing particles may disappear, while others may appear multiple times. For each $i = 1, \ldots, M$ the number of times $\mathbf{x}_{t\,|\,t-1}^{(i)}$ appears in the resampled set of particles is known as the particle's *replication factor*, $N_t^{(i)}$.

We set the normalized weights after resampling: $\lambda_t^{(i)} := \frac{1}{M}$. We could view this algorithm as the sampling of the replication factors $N_t^{(1)}, \ldots N_t^{(M)}$ from the multinomial distribution with probabilities ${}^{\mathrm{br}}\lambda_t^{(1)}, \ldots, {}^{\mathrm{br}}\lambda_t^{(M)}$, respectively. Hence the name of the method. ■

3.3 Application: Stochastic Volatility Models

Stochastic Volatility (SV) models have been studied extensively in the literature, often as applications of *particle filtering* and *Markov chain Monte Carlo (MCMC)*. Their broad appeal in finance is their ability to capture the "leverage effect"—the observed tendency of an asset's volatility to be negatively correlated with the asset's returns (Black 1976).

In particular, Pitt, Malik, and Doucet apply the particle filter to the *stochastic volatility with leverage and jumps (SVLJ)* (Malik and Pitt 2009, 2011a,b; Pitt et al. 2014). The model has the general form of Taylor (1982) with two modifications. For $t \in \mathbb{N} \cup \{0\}$, let y_t denote the log-return on an asset and x_t denote the log-variance of that return. Then

$$y_t = \epsilon_t e^{x_t/2} + J_t \varpi_t, \tag{7.13}$$

$$x_{t+1} = \mu(1-\phi) + \phi x_t + \sigma_v \eta_t, \tag{7.14}$$

where μ is the mean log-variance, ϕ is the persistence parameter, σ_v is the volatility of log-variance.

The first modification to Taylor's model is the introduction of correlation between ϵ_t and η_t:

$$\begin{pmatrix} \epsilon_t \\ \eta_t \end{pmatrix} \sim \mathcal{N}(0, \Sigma), \quad \Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}.$$

The correlation ρ is the *leverage* parameter. In general, $\rho < 0$, due to the leverage effect.

The second change is the introduction of jumps. $J_t \in \{0, 1\}$ is a Bernoulli counter with intensity p (thus p is the jump intensity parameter), $\varpi_t \sim \mathcal{N}(0, \sigma_J^2)$ determines the jump size (thus σ_J is the jump volatility parameter).

We obtain a *stochastic volatility with leverage (SVL)*, but no jumps, if we delete the $J_t \varpi_t$ term or, equivalently, set p to zero. Taylor's original model is a special case of SVLJ with $p = 0$, $\rho = 0$.

This, then, leads to the following adaptation of SIR, developed by Doucet, Malik, and Pitt, for this special case with nonadditive, correlated noises. The initial distribution of x_0 is taken to be $\mathcal{N}\left(0, \sigma_v^2/(1-\phi^2)\right)$.

a. *Initialization step*: At time $t = 0$, draw M i.i.d. particles from the initial distribution $\mathcal{N}(0, \sigma_v^2/(1-\phi^2))$. Also, initialize M normalized (to 1) weights to an identical value of $\frac{1}{M}$. For $i = 1, 2, \ldots, M$, the samples will be denoted $\hat{x}_{0\,|\,0}^{(i)}$ and the normalized weights $\lambda_0^{(i)}$.

b. *Recursive step*: At time $t \in \mathbb{N}$, let $(\hat{x}_{t-1\,|\,t-1}^{(i)})_{i=1,\ldots,M}$ be the particles generated at time $t-1$.

i *Importance sampling*:

– First,

– For $i = 1, \ldots, M$, sample $\hat{\epsilon}_{t-1}^{(i)}$ from $p(\epsilon_{t-1} \mid x_{t-1} = \hat{x}_{t-1 \mid t-1}^{(i)}, y_{t-1})$. (If no y_{t-1} is available, as at $t = 1$, sample from $p(\epsilon_{t-1} \mid x_{t-1} = \hat{x}_{t-1 \mid t-1}^{(i)})$).

– For $i = 1, \ldots, M$, sample $\hat{x}_{t \mid t-1}^{(i)}$ from $p(x_t \mid x_{t-1} = \hat{x}_{t-1 \mid t-1}^{(i)}, y_{t-1}, \hat{\epsilon}_{t-1}^{(i)})$.

– For $i = 1, \ldots, M$, compute the non-normalized weights:

$$\omega_t^{(i)} := \lambda_{t-1}^{(i)} \cdot p_{\gamma_t}(y_t \mid \hat{x}_{t \mid t-1}^{(i)}), \tag{7.15}$$

using the observation density

$$p(y_t \mid \hat{x}_{t \mid t-1}^{(i)}, p, \sigma_J^2) = (1-p)\left[\left(2\pi e^{\hat{x}_{t \mid t-1}^{(i)}}\right)^{-1/2} \exp\left(-y_t^2/(2e^{\hat{x}_{t \mid t-1}^{(i)}})\right)\right] + p\left[\left(2\pi(e^{\hat{x}_{t \mid t-1}^{(i)}} + \sigma_J^2)\right)^{-1/2} \exp\left(-y_t^2/(2e^{\hat{x}_{t \mid t-1}^{(i)}} + 2\sigma_J^2)\right)\right],$$

and the values of the normalized weights before resampling ('br'):

$${}^{\text{br}}\lambda_t^{(i)} := \frac{\omega_t^{(i)}}{\sum_{k=1}^M \omega_t^{(k)}}.$$

ii *Resampling* (or *selection*): For $i = 1, \ldots, M$, use an appropriate resampling algorithm (such as multinomial resampling) sample $\hat{x}_{t \mid t}^{(i)}$ from the mixture

$$\sum_{k=1}^{M} {}^{\text{br}}\lambda_t^{(k)} \delta(x_t - \hat{x}_{t \mid t-1}^{(k)}),$$

where $\delta(\cdot)$ denotes the Dirac delta generalized function, and set the normalized weights after resampling, $\lambda_t^{(i)}$, according to the resampling algorithm.

4 Point Calibration of Stochastic Filters

We have seen in the example of the stochastic volatility model with leverage and jumps (SVLJ) that the state-space model may be parameterized by a parameter vector, $\boldsymbol{\theta} \in \mathbb{R}^{d_\theta}$, $d_\theta \in \mathbb{N}$. In that particular case,

$$\boldsymbol{\theta} = \begin{pmatrix} \mu \\ \phi \\ \sigma_\eta^2 \\ \rho \\ \sigma_J^2 \\ p \end{pmatrix}.$$

We may not know the true value of this parameter. How do we estimate it? In other words, how do we *calibrate* the model, given a time series of either historical or generated observations, $\mathbf{y}_1, \ldots, \mathbf{y}_T$, $T \in \mathbb{N}$.

The frequentist approach relies on the (joint) probability density function of the observations, which depends on the parameters, $p(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T \mid \boldsymbol{\theta})$. We can regard this as a function of $\boldsymbol{\theta}$ with $\mathbf{y}_1, \ldots, \mathbf{y}_T$ fixed, $p(\mathbf{y}_1, \ldots, \mathbf{y}_T \mid \boldsymbol{\theta}) =: \mathcal{L}(\boldsymbol{\theta})$—the *likelihood function*.

This function is sometimes referred to as *marginal* likelihood, since the hidden states, $\mathbf{x}_1, \ldots, \mathbf{x}_T$, are marginalized out. We seek a *maximum likelihood estimator (MLE)*, $\hat{\boldsymbol{\theta}}_{ML}$, the value of $\boldsymbol{\theta}$ that maximizes the likelihood function.

Each evaluation of the objective function, $\mathcal{L}(\boldsymbol{\theta})$, constitutes a run of the stochastic filter over the observations $\mathbf{y}_1, \ldots, \mathbf{y}_T$. By the chain rule (i), and since we use a Markov chain (ii),

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_T) \overset{\text{(i)}}{=} \prod_{t=1}^{T} p(\mathbf{y}_t \mid \mathbf{y}_0, \ldots, \mathbf{y}_{t-1}) \overset{\text{(ii)}}{=} \prod_{t=1}^{T} \int p(\mathbf{y}_t \mid \mathbf{x}_t) p(\mathbf{x}_t \mid \mathbf{y}_0, \ldots, \mathbf{y}_{t-1}) \, d\mathbf{x}_t.$$

Note that, for ease of notation, we have omitted the dependence of all the probability densities on $\boldsymbol{\theta}$, e.g., instead of writing $p(\mathbf{y}_1, \ldots, \mathbf{y}_T; \boldsymbol{\theta})$.

For the particle filter, we can estimate the log-likelihood function from the non-normalized weights:

$$p(\mathbf{y}_1, \ldots, \mathbf{y}_T) = \prod_{t=1}^{T} \int p(\mathbf{y}_t \mid \mathbf{x}_t) p(\mathbf{x}_t \mid \mathbf{y}_0, \ldots, \mathbf{y}_{t-1}) \, d\mathbf{x}_t \approx \prod_{t=1}^{T} \left(\frac{1}{M} \sum_{k=1}^{M} \omega_t^{(k)} \right),$$

whence

$$\ln(\mathcal{L}(\boldsymbol{\theta})) = \ln \left\{ \prod_{t=1}^{T} \left(\frac{1}{M} \sum_{k=1}^{M} \omega_t^{(k)} \right) \right\} = \sum_{t=1}^{T} \ln \left(\frac{1}{M} \sum_{k=1}^{M} \omega_t^{(k)} \right). \tag{7.16}$$

This was first proposed by Kitagawa (1993, 1996) for the purposes of approximating $\hat{\boldsymbol{\theta}}_{ML}$.

In most practical applications one needs to resort to numerical methods, perhaps quasi-Newton methods, such as *Broyden–Fletcher–Goldfarb–Shanno (BFGS)* (Gill et al. 1982), to find $\hat{\boldsymbol{\theta}}_{ML}$.

Pitt et al. (2014) point out the practical difficulties which result when using the above as an objective function in an optimizer. In the resampling (or selection) step of the particle filter, we are sampling from a discontinuous empirical distribution function. Therefore, $\ln(\mathcal{L}(\boldsymbol{\theta}))$ will not be continuous as a function of $\boldsymbol{\theta}$. To remedy this, they rely on an alternative, continuous, resampling procedure. A quasi-Newton method is then used to find $\hat{\boldsymbol{\theta}}_{ML}$ for the parameters $\boldsymbol{\theta} = (\mu, \phi, \sigma_v^2, \rho, p, \sigma_J^2)^\intercal$ of the SVLJ model.

We note in passing that Kalman filters can also be calibrated using a similar maximum likelihood approach.

5 Bayesian Calibration of Stochastic Filters

Let us briefly discuss how filtering methods relate to *Markov chain Monte Carlo methods (MCMC)*—a vast subject in its own right; therefore, our discussion will be cursory at best. The technique takes its origin from Metropolis et al. (1953).

Following Kim et al. (1998) and Meyer and Yu (2000); Yu (2005), we demonstrate how MCMC techniques can be used to estimate the parameters of the SVL model. They calibrate the parameters to the time series of observations of daily mean-adjusted log-returns, $y_1, \ldots, y_T$ to obtain the joint prior density

$$p(\boldsymbol{\theta}, x_0, \ldots, x_T) = p(\boldsymbol{\theta}) p(x_0 \mid \boldsymbol{\theta}) \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \boldsymbol{\theta})$$

by successive conditioning. Here $\boldsymbol{\theta} := (\mu, \phi, \sigma_v^2, \rho)^\intercal$ is, as before, the vector of the model parameters. We assume prior independence of the parameters and choose the same priors (as in Kim et al. (1998)) for μ, ϕ, and σ_v^2, and a uniform prior for ρ. The observation model and the conditional independence assumption give the likelihood

$$p(y_1, \ldots, y_T \mid \boldsymbol{\theta}, x_0, \ldots, x_T) = \prod_{t=1}^{T} p(y_t \mid x_t),$$

and the joint posterior distribution of the *unobservables* (the parameters $\boldsymbol{\theta}$ and the hidden states $x_0, \ldots, x_T$; in the Bayesian perspective these are treated identically and estimated in a similar manner) follows from Bayes' theorem; for the SVL model, this posterior satisfies

$$\begin{aligned} p(\boldsymbol{\theta}, x_0, \ldots, x_T \mid y_1, \ldots, y_T) &\propto p(\mu) p(\phi) p(\sigma_v^2) p(\rho) \\ &\prod_{t=1}^{T} p(x_{t+1} \mid x_t, \mu, \phi, \sigma_v^2) \prod_{t=1}^{T} p(y_t \mid x_{t+1}, x_t, \mu, \phi, \sigma_v^2, \rho), \end{aligned}$$

where $p(\mu)$, $p(\phi)$, $p(\sigma_v^2)$, $p(\rho)$ are the appropriately chosen priors,

$$x_{t+1} \mid x_t, \mu, \phi, \sigma_v^2 \sim \mathcal{N}\left(\mu(1-\phi) + \phi x_t, \sigma_v^2\right),$$

$$y_t \mid x_{t+1}, x_t, \mu, \phi, \sigma_v^2, \rho \sim \mathcal{N}\left(\frac{\rho}{\sigma_v} e^{x_t/2}\left(x_{t+1} - \mu(1-\phi) - \phi x_t\right), e^{x_t}(1-\rho^2)\right).$$

Meyer and Yu use the software package BUGS[3] (Spiegelhalter et al. 1996; Lunn et al. 2000) represent the resulting Bayesian model as a *directed acyclic graph (DAG)*, where the nodes are either constants (denoted by rectangles), stochastic nodes (variables that are given a distribution, denoted by ellipses), or deterministic nodes (logical functions of other nodes); the arrows either indicate stochastic dependence (solid arrows) or logical functions (hollow arrows). This graph helps visualize the conditional (in)dependence assumptions and is used by BUGS to construct full univariate conditional posterior distributions for all unobservables. It then uses Markov chain Monte Carlo algorithms to sample from these distributions.

The algorithm based on the original work (Metropolis et al. 1953) is now known as the *Metropolis algorithm*. It has been generalized by Hastings (1930–2016) to obtain the *Metropolis–Hastings algorithm* (Hastings 1970) and further by Green to obtain what is known as the *Metropolis–Hastings–Green algorithm* (Green 1995). A popular algorithm based on a special case of the Metropolis–Hastings algorithm, known as the *Gibbs sampler*, was developed by Geman and Geman (1984) and, independently, Tanner and Wong (1987).[4] It was further popularized by Gelfand and Smith (1990). Gibbs sampling and related algorithms (Gilks and Wild 1992; Ritter and Tanner 1992) are used by BUGS to sample from the univariate conditional posterior distributions for all unobservables. As a result we perform Bayesian estimation—obtain estimates of the *distributions* of the parameters μ, ϕ, σ_v^2, ρ—rather than frequentist estimation, where a single value of the parameters vector, which maximizes the likelihood, $\hat{\boldsymbol{\theta}}_{ML}$, is produced. Stochastic filtering, sometimes in combination with MCMC, can be used for both frequentist and Bayesian parameter estimation (Chen 2003). Filtering methods that update estimates of the parameters online, while processing observations in real-time, are referred to as *adaptive filtering* (see Sayed (2008); Vega and Rey (2013); Crisan and Míguez (2013); Naesseth et al. (2015) and references therein).

We note that a Gibbs sampler (or variants thereof) is a highly nontrivial piece of software. In addition to the now classical BUGS/WinBUGS there exist powerful Gibbs samplers accessible via modern libraries, such as Stan, Edward, and PyMC3.

[3] An acronym for Bayesian inference Using Gibbs Sampling.

[4] Sometimes the Gibbs sampler is referred to as *data augmentation* following this paper.

6 Summary

This chapter extends Chap. 2 by presenting probabilistic methods for time series data. The key modeling assumption is the existence of a certain latent process X_t, which evolves over time. This unobservable, latent process drives another, observable process. Such an approach overcomes limitations of stationarity imposed on the methods in the previous chapter. The reader should verify that they have achieved the primary learning objectives of this chapter:

- Formulate hidden Markov models (HMMs) for probabilistic modeling over hidden states;
- Gain familiarity with the Baum–Welch algorithm for fitting HMMs to time series data;
- Use the Viterbi algorithm to find the most likely path;
- Gain familiarity with state-space models and the application of Kalman filters to fit them; and
- Apply particle filters to financial time series.

7 Exercises

Exercise 7.1: Kalman Filtering of Autoregressive Moving Average $ARMA(p, q)$ Model
The *autoregressive moving average* ARMA(p, q) model can be written as

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \eta_t + \theta_1 \eta_{t-1} + \ldots + \theta_q \eta_{t-q},$$

where $\eta_t \sim \mathcal{N}(0, \sigma^2)$ and includes as special cases all AR(p) and MA(q) models. Such models are often fitted to financial time series. Suppose that we would like to filter this time series using a Kalman filter. Write down a suitable process and the observation models.

Exercise 7.2: The Ornstein–Uhlenbeck Process
Consider the one-dimensional *Ornstein–Uhlenbeck (OU)* process, the stationary Gauss–Markov process given by the SDE

$$dX_t = \theta(\mu - X_t)\, dt + \sigma\, dW_t,$$

where $X_t \in \mathbb{R}$, $X_0 = x_0$, and $\theta > 0$, μ, and $\sigma > 0$ are constants. Formulate the Kalman process model for this process.

Exercise 7.3: Deriving the Particle Filter for Stochastic Volatility with Leverage and Jumps

We shall regard the log-variance x_t as the hidden states and the log-returns y_t as observations. How can we use the particle filter to estimate x_t on the basis of the observations y_t?

a. Show that, in the absence of jumps,

$$x_t = \mu(1-\phi) + \phi x_{t-1} + \sigma_v \rho y_{t-1} e^{-x_{t-1}/2} + \sigma_v \sqrt{1-\rho^2}\xi_{t-1}$$

for some $\xi_t \overset{i.i.d.}{\sim} \mathcal{N}(0,1)$.

b. Show that

$$\begin{aligned} p(\epsilon_t \mid x_t, y_t) =& \delta(\epsilon_t - y_t e^{-x_t/2})\mathbb{P}[J_t = 0 \mid x_t, y_t] \\ &+ \phi(\epsilon_t; \mu_{\epsilon_t \mid J_t=1}, \sigma^2_{\epsilon_t \mid J_t=1})\mathbb{P}[J_t = 1 \mid x_t, y_t], \end{aligned}$$

where

$$\mu_{\epsilon_t \mid J_t=1} = \frac{y_t \exp(x_t/2)}{\exp(x_t) + \sigma_J^2}$$

and

$$\sigma^2_{\epsilon_t \mid J_t=1} = \frac{\sigma_J^2}{\exp(x_t) + \sigma_J^2}.$$

c. Explain how you could implement random sampling from the probability distribution given by the density $p(\epsilon_t \mid x_t, y_t)$.

d. Write down the probability density $p(x_t \mid x_{t-1}, y_{t-1}, \epsilon_{t-1})$.

e. Explain how you could sample from this distribution.

f. Show that the observation density is given by

$$\begin{aligned} p(y_t \mid \hat{x}^{(i)}_{t\mid t-1}, p, \sigma_J^2) = (1-p)\left[\left(2\pi e^{\hat{x}^{(i)}_{t\mid t-1}}\right)^{-1/2} \exp\left(-y_t^2/(2e^{\hat{x}^{(i)}_{t\mid t-1}})\right)\right] + \\ p\left[\left(2\pi(e^{\hat{x}^{(i)}_{t\mid t-1}} + \sigma_J^2)\right)^{-1/2} \exp\left(-y_t^2/(2e^{\hat{x}^{(i)}_{t\mid t-1}} + 2\sigma_J^2)\right)\right]. \end{aligned}$$

Exercise 7.4: The Viterbi Algorithm and an Occasionally Dishonest Casino
The dealer has two coins, a fair coin, with $\mathbb{P}(\text{Heads}) = \frac{1}{2}$, and a loaded coin, with $\mathbb{P}(\text{Heads}) = \frac{4}{5}$. The dealer starts with the fair coin with probability $\frac{3}{5}$. The dealer then tosses the coin several times. After each toss, there is a $\frac{2}{5}$ probability of a switch to the other coin. The observed sequence is Heads, Tails, Tails, Heads, Tails, Heads, Heads, Heads, Tails, Heads. Run the Viterbi algorithm to determine which coin the dealer was most likely using for each coin toss.

Appendix

Python Notebooks

The notebooks provided in the accompanying source code repository are designed to gain familiarity with how to implement the Viterbi algorithm and particle filtering for stochastic volatility model calibration. Further details of the notebooks are included in the `README.md` file.

References

Black, F. (1976). Studies of stock price volatility changes. In *Proceedings of the Business and Economic Statistics Section*.

Chen, Z. (2003). Bayesian filtering: From Kalman filters to particle filters, and beyond. *Statistics, 182*(1), 1–69.

Crisan, D., & Míguez, J. (2013). Nested particle filters for online parameter estimation in discrete-time state-space Markov models. *ArXiv:1308.1883*.

Gelfand, A. E., & Smith, A. F. M. (1990, June). Sampling-based approaches to calculating marginal densities. *Journal of the American Statistical Association, 85*(410), 398–409.

Geman, S. J., & Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 6*, 721–741.

Gilks, W. R., & Wild, P. P. (1992). Adaptive rejection sampling for Gibbs sampling, Vol. 41, pp. 337–348.

Gill, P. E., Murray, W., & Wright, M. H. (1982). *Practical optimization*. Emerald Group Publishing Limited.

Gordon, N. J., Salmond, D. J., & Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE Proceedings F (Radar and Signal Processing)*.

Green, P. J. (1995). Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika, 82*(4), 711–32.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika, 57*(1), 97–109.

Kim, S., Shephard, N., & Chib, S. (1998, July). Stochastic volatility: Likelihood inference and comparison with ARCH models. *The Review of Economic Studies, 65*(3), 361–393.

Kitagawa, G. (1993). A Monte Carlo filtering and smoothing method for non-Gaussian nonlinear state space models. In *Proceedings of the 2nd U.S.-Japan Joint Seminar on Statistical Time Series Analysis* (pp. 110–131).

Kitagawa, G. (1996). Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics, 5*(1), 1–25.

Lunn, D. J., Thomas, A., Best, N. G., & Spiegelhalter, D. (2000). WinBUGS – a Bayesian modelling framework: Concepts, structure and extensibility. *Statistics and Computing, 10*, 325–337.

Malik, S., & Pitt, M. K. (2009, April). *Modelling stochastic volatility with leverage and jumps: A simulated maximum likelihood approach via particle filtering*. Warwick Economic Research Papers 897, The University of Warwick, Department of Economics, Coventry CV4 7AL.

Malik, S., & Pitt, M. K. (2011a, February). *Modelling stochastic volatility with leverage and jumps: A simulated maximum likelihood approach via particle filtering*. document de travail 318, Banque de France Eurosystème.

Malik, S., & Pitt, M. K. (2011b). Particle filters for continuous likelihood evaluation and maximisation. *Journal of Econometrics, 165*, 190–209.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics, 21*.

Meyer, R., & Yu, J. (2000). BUGS for a Bayesian analysis of stochastic volatility models. *Econometrics Journal, 3*, 198–215.

Naesseth, C. A., Lindsten, F., & Schön, T. B. (2015). Nested sequential Monte Carlo methods. In *Proceedings of the 32nd International Conference on Machine Learning*.

Pitt, M. K., Malik, S., & Doucet, A. (2014). Simulated likelihood inference for stochastic volatility models using continuous particle filtering. *Annals of the Institute of Statistical Mathematics, 66*, 527–552.

Ritter, C., & Tanner, M. A. (1992). Facilitating the Gibbs sampler: The Gibbs stopper and the Griddy-Gibbs sampler. *Journal of the American Statistical Association, 87*(419), 861–868.

Sayed, A. H. (2008). *Adaptive filters*. Wiley-Interscience.

Spiegelhalter, D., Thomas, A., Best, N. G., & Gilks, W. R. (1996, August). *BUGS 0.5: Bayesian inference using Gibbs sampling manual (version ii)*. Robinson Way, Cambridge CB2 2SR: MRC Biostatistics Unit, Institute of Public Health.

Tanner, M. A., & Wong, W. H. (1987, June). The calculation of posterior distributions by data augmentation. *Journal of the American Statistical Association, 82*(398), 528–540.

Taylor, S. J. (1982). *Time series analysis: theory and practice*. Chapter Financial returns modelled by the product of two stochastic processes, a study of daily sugar prices, pp. 203–226. North-Holland.

Vega, L. R., & H. Rey (2013). *A rapid introduction to adaptive filtering*. Springer Briefs in Electrical and Computer Engineering. Springer.

Yu, J. (2005). On leverage in a stochastic volatility model. *Journal of Econometrics, 127*, 165–178.

Chapter 8
Advanced Neural Networks

This chapter presents various neural network models for financial time series analysis, providing examples of how they relate to well-known techniques in financial econometrics. Recurrent neural networks (RNNs) are presented as non-linear time series models and generalize classical linear time series models such as $AR(p)$. They provide a powerful approach for prediction in financial time series and generalize to non-stationary data. This chapter also presents convolution neural networks for filtering time series data and exploiting different scales in the data. Finally, this chapter demonstrates how autoencoders are used to compress information and generalize principal component analysis.

1 Introduction

The universal approximation theorem states that a feedforward network is capable of approximating any function. So why do other types of neural networks exist? One answer to this is efficiency. In this chapter, different architectures shall be explored for their ability to exploit the structure in the data, resulting in fewer weights. Hence the main motivation for different architectures is often parsimony of parameters and therefore less propensity to overfit and reduced training time. We shall see that other architectures can be used, in particular ones that change their behavior over time, without the need to retrain the networks. And we will see how neural networks can be used to compress data, analogously to principal component analysis.

There are other neural network architectures which are used in financial applications but are too esoteric to list here. However, we shall focus on three other classes of neural networks which have proven to be useful in the finance industry. The first two are supervised learning techniques and the latter is an unsupervised learning technique.

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_8

Recurrent neural networks (RNNs) are non-linear time series models and generalize classical linear time series models such as $AR(p)$. They provide a powerful approach for prediction in financial time series and share parameters across time. Convolution neural networks are useful as spectral transformations of spatial and temporal data and generalize techniques such as wavelets, which use fixed basis functions. They share parameters across space. Finally, autoencoders are used to compress information and generalize principal component analysis.

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Characterize RNNs as non-linear autoregressive models and analyze their stability;
- Understand how gated recurrent units and long short-term memory architectures give a dynamic autoregressive model with variable memory;
- Characterize CNNs as regression, classification, and time series regression of filtered data;
- Understand principal component analysis for dimension reduction;
- Formulate a linear autoencoder and extract the principal components; and
- Understand how to build more complex networks by aggregating these different concepts.

The notebooks provided in the accompanying source code repository demonstrate many of the methods in this chapter. See Appendix "Python Notebooks" for further details.

2 Recurrent Neural Networks

Recall that if the data $\mathcal{D} := \{x_t, y_t\}_{t=1}^N$ is auto-correlated observations of X and Y at times $t = 1, \ldots, N$, then the prediction problem can be expressed as a sequence prediction problem: construct a non-linear times series predictor, $\hat{y}_{t+h}$, of a response, y_{t+h}, using a high-dimensional input matrix of T length sub-sequences $\mathcal{X}_t$:

$$\hat{y}_{t+h} = f(\mathcal{X}_t) \text{ where } \mathcal{X}_t := seq_{T,t}(X) = (x_{t-T+1}, \ldots, x_t),$$

where x_{t-j} is a jth lagged observation of x_t, $x_{t-j} = L^j[x_t]$, for $j = 0, \ldots, T-1$. Sequence learning, then, is just a composition of a non-linear map and a vectorization of the lagged input variables. If the data is i.i.d., then no sequence is needed (i.e., $T = 1$), and we recover a feedforward neural network.

Recurrent neural networks (RNNs) are times series methods or sequence learners which have achieved much success in applications such as natural language understanding, language generation, video processing, and many other tasks (Graves 2012). There are many types of RNNs—we will just concentrate on simple RNN models for brevity of notation. Like multivariate structural autoregressive models, RNNs apply an autoregressive function $f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_t)$ to each input sequence $\mathcal{X}_t$, where T denotes the look back period at each time step—the maximum number of lags. However, rather than directly imposing an autocovariance structure, a RNN provides a flexible functional form to directly model the predictor, $\hat{Y}$.

As illustrated in Fig. 8.1, this simple RNN is an unfolding of a single hidden layer neural network (a.k.a. Elman network (Elman 1991)) over all time steps in the sequence, $j = 0, \ldots, T-1$. For each time step, j, this function $f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_{t,j})$ generates a hidden state z_{t-j} from the current input x_t and the previous hidden state z_{t-1} and $\mathcal{X}_{t,j} = seq_{T,t-j}(X) \subset \mathcal{X}_t$:

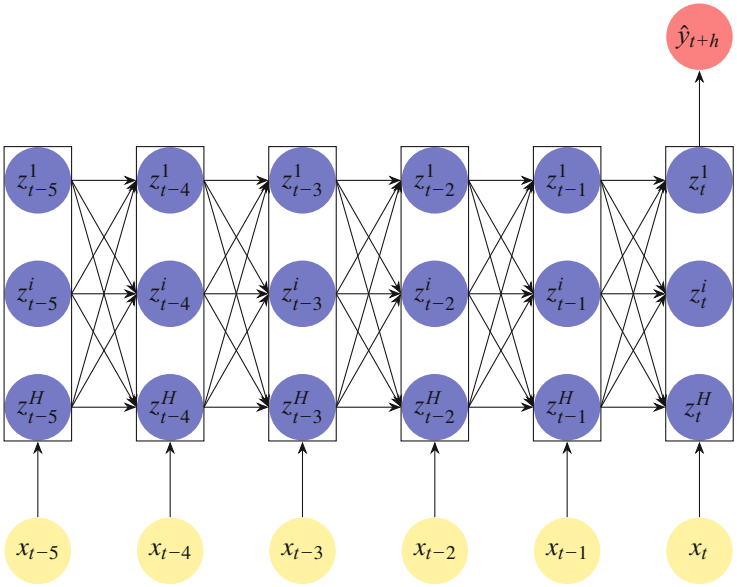


Fig. 8.1 An illustrative example of a recurrent neural network with one hidden layer, "unfolded" over a sequence of six time steps. Each input x_t is in the sequence $\mathcal{X}_t$. The hidden layer contains H units and the ith output at time step t is denoted by z^i_t. The connections between the hidden units are recurrent and are weighted by the matrix $W^{(1)}_z$. At the last time step t, the hidden units connect to a single unit output layer with continuous $\hat{y}_{t+h}$

$$\text{response:} \quad \hat{y}_{t+h} = f^{(2)}_{W^{(2)},b^{(2)}}(z_t) := \sigma^{(2)}(W^{(2)} z_t + b^{(2)}),$$

$$\text{hidden states: } z_{t-j} = f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_{t,j})$$
$$:= \sigma^{(1)}(W_z^{(1)} z_{t-j-1} + W_x^{(1)} x_{t-j} + b^{(1)}),\ j \in \{T-1, \ldots, 0\},$$

where $\sigma^{(1)}$ is an activation function such as $\tanh(x)$, and $\sigma^{(2)}$ is either a softmax function or identity map depending on whether the response is categorical or continuous, respectively. The connections between the extremal inputs x_t and the H hidden units are weighted by the time invariant matrix $W_x^{(1)} \in \mathbb{R}^{H \times P}$. The recurrent connections between the H hidden units are weighted by the time invariant matrix $W_z^{(1)} \in \mathbb{R}^{H \times H}$. Without such a matrix, the architecture is simply a single-layered feedforward network without memory—each independent observation x_t is mapped to an output $\hat{y}_t$ using the same hidden layer.

The sets $W^{(1)} = (W_x^{(1)}, W_z^{(1)})$ refer to the input and recurrence weights. $W^{(2)}$ denotes the weights tied to the output of the H hidden units at the last time step, z_t, and the output layer. If the response is a continuous vector, $Y \in \mathbb{R}^M$, then $W^{(2)} \in \mathbb{R}^{M \times H}$. If the response is categorical, with K states, then $W^{(2)} \in \mathbb{R}^{K \times H}$. The number of hidden units determines the degree of non-linearity in the model and must be at least the dimensionality of the input p. In our experiments the hidden layer is generally under a hundred units, but can increase to thousands in higher dimensional datasets.

There are a number of issues in the RNN design. How many times should the network being unfolded? How many hidden neurons H in the hidden layer? How to perform "variable selection"? The answer to the first question lies in tests for autocorrelation of the data—the sequence length needed in a RNN can be determined by the largest significant lag in an estimated "partial autocorrelation" function. The answer to the second is no different to the problem of how to choose the number of hidden neurons in a MLP—the bias–variance tradeoff being the most important consideration. And indeed the third question is also closely related to the problem of choosing features in a MLP. One can take a principled approach to feature selection, first identifying a subset of features using a Granger-causality test, or the "laissez-machine" approach of including all potentially relevant features and allowing auto-shrinkage to determine the most important weights are more aligned with contemporary experimental design in machine learning. One important caveat on feature selection for RNNs is that each feature must be a time series and therefore exhibit autocorrelation.

We shall begin with a simple, univariate, example to illustrate how a RNN, without activation, is a $AR(p)$ time series model.

Example 8.1 RNNs as Non-linear AR(p) Models

Consider the simplest case of a RNN with one hidden unit, $H = 1$, no activation function, and the dimensionality of the input vector is $P = 1$. Suppose further that $W_z^{(1)} = \phi_z$, $|\phi_z| < 1$, $W_x^{(1)} = \phi_x$, $W_y = 1$, $b_h = 0$ and $b_y = \mu$. Then we can show that $f^{(1)}_{W^{(1)},b^{(1)}}(\mathcal{X}_t)$ is of an autoregressive, $AR(p)$, model of order p with geometrically decaying autoregressive coefficients $\phi_i = \phi_x \phi_z^{i-1}$:

$$\begin{aligned} z_{t-p} &= \phi_x x_{t-p} \\ z_{t-T+2} &= \phi_z z_{t-T+1} + \phi_x x_{t-T+2} \\ \dots &= \dots \\ z_{t-1} &= \phi_z z_{t-2} + \phi_x x_{t-1} \\ \hat{x}_t &= z_{t-1} + \mu \end{aligned}$$

then

$$\begin{aligned} \hat{x}_t &= \mu + \phi_x (L + \phi_z L^2 + \dots + \phi_z^{p-1} L^p)[x_t] \\ &= \mu + \sum_{i=1} \phi_i x_{t-i} \end{aligned}$$

This special type of autoregressive model $\hat{x}_t$ is "stable" and the order can be identified through autocorrelation tests on X such as the Durbin–Watson, Ljung–Box, or Box–Pierce tests. Note that if we modify the architecture so that the recurrence weights $W_{z,i}^{(1)} = \phi_{z,i}$ are lag dependent then the unactivated hidden layer is

$$z_{t-i} = \phi_{z,i} z_{t-i-1} + \phi_x x_{t-i} \tag{8.1}$$

which gives

$$\hat{x}_t = \mu + \phi_x (L + \phi_{z,1} L^2 + \dots + \prod_{i=1}^{p-1} \phi_{z,i} L^p)[x_t], \tag{8.2}$$

and thus the weights in this $AR(p)$ model are $\phi_j = \phi_x \prod_{i=1}^{j-1} \phi_{z,i}$ which allows a more flexible presentation of the autocorrelation structure than the plain RNN—which is limited to geometrically decaying weights. Note that a linear RNN with infinite number of lags and no bias corresponds to an exponential smoother, $z_t = \alpha x_t + (1 - \alpha) z_{t-1}$ when $W_z = 1 - \alpha$, $W_x = \alpha$, and $W_y = 1$.

The generalization of a linear RNN from $AR(p)$ to $VAR(p)$ is trivial and can be written as

$$\hat{x}_t = \boldsymbol{\mu} + \sum_{j=1}^{p} \phi_j x_{t-j}, \phi_j := W^{(2)}(W_z^{(1)})^{j-1} W_x^{(1)}, \boldsymbol{\mu} := W^{(2)} \sum_{j=1}^{p} (W_z^{(1)})^{j-1} b^{(1)} + b^{(2)}, \tag{8.3}$$

where the square matrix $\phi_j \in \mathbb{R}^{P \times P}$ and bias vector $\boldsymbol{\mu} \in \mathbb{R}^P$.

2.1 RNN Memory: Partial Autocovariance

Generally, with non-linear activation, it is more difficult to describe the RNN as a classical model. However, the partial autocovariance function provides some additional insight here. Let us first consider a RNN(1) process. The lag-1 partial autocovariance is

$$\tilde{\gamma}_1 = \mathbb{E}[y_t - \mu, y_{t-1} - \mu] = \mathbb{E}[\hat{y}_t + \epsilon_t - \mu, y_{t-1} - \mu], \tag{8.4}$$

and using the RNN(1) model with, for simplicity, a single recurrence weight, ϕ:

$$\hat{y}_t = \sigma(\phi y_{t-1}) \tag{8.5}$$

gives

$$\tilde{\gamma}_1 = \mathbb{E}[\sigma(\phi y_{t-1}) + \epsilon_t - \mu, y_{t-1} - \mu] = \mathbb{E}[y_{t-1}\sigma(\phi y_{t-1})], \tag{8.6}$$

where we have assumed $\mu = 0$ in the second part of the expression. Checking that we recover the AR(1) covariance, set $\sigma := Id$ so that

$$\tilde{\gamma}_1 = \phi \mathbb{E}[y_{t-1}^2] = \phi \mathbb{V}[y_{t-1}]. \tag{8.7}$$

Continuing with the lag-2 autocovariance gives:

$$\tilde{\gamma}_2 = \mathbb{E}[y_t - P(y_t \mid y_{t-1}), y_{t-2} - P(y_{t-2} \mid y_{t-1})], \tag{8.8}$$

and $P(y_t \mid y_{t-1})$ is approximated by the RNN(1):

$$\hat{y}_t = \sigma(\phi y_{t-1}). \tag{8.9}$$

Substituting $y_t = \hat{y}_t + \epsilon_t$ into the above gives

$$\tilde{\gamma}_2 = \mathbb{E}[\epsilon_t, y_{t-2} - P(y_{t-2} \mid y_{t-1})]. \tag{8.10}$$

Approximating $P(y_{t-2} \mid y_{t-1})$ with the backward RNN(1)

$$\hat{y}_{t-2} = \sigma(\phi(\hat{y}_{t-1} + \epsilon_{t-1})), \tag{8.11}$$

we see, crucially, that $\hat{y}_{t-2}$ depends on ϵ_{t-1} but not on ϵ_t. $y_{t-2} - P(y_{t-2} \mid y_{t-1})$, hence depends on $\{\epsilon_{t-1}, \epsilon_{t-2}, \dots\}$. Thus we have that $\tilde{\gamma}_2 = 0$.

As a counterexample, consider the lag-2 partial autocovariance of the RNN(2) process

$$\hat{y}_{t-2} = \sigma\left(\phi\sigma(\phi(\hat{y}_t + \epsilon_t) + \epsilon_{t-1})\right), \tag{8.12}$$

which depends on ϵ_t and hence the lag-2 partial autocovariance is not zero.

It is easy to show that the partial autocorrelation $\tilde{\tau}_s = 0, s > p$ and, thus, like the AR(p) process, the partial autocorrelation function for a RNN(p) has a cut-off at p lags. The partial autocorrelation function is independent of time. Such a property can be used to identify the order of the RNN model from the estimated PACF.

2.2 Stability

We can generalize the stability constraint on AR(p) models presented in Sect. 2.3 to RNNs by considering the RNN(1) model:

$$y_t = \Phi^{-1}(L)[\epsilon_t] = (1 - \sigma(W_z L + b))^{-1}[\epsilon_t], \tag{8.13}$$

where we have set $W_y = 1$ and $b_y = 0$ without loss of generality, and dropped the superscript $()^{(1)}$ for ease of notation. Expressing this as an infinite dimensional non-linear moving average model

$$y_t = \frac{1}{1 - \sigma(W_z L + b)}[\epsilon_t] = \sum_{j=0}^{\infty} \sigma^j(W_z L + b)[\epsilon_t], \tag{8.14}$$

and the infinite sum will be stable when the $\sigma^j(\cdot)$ terms do not grow with j, i.e. $|\sigma| \leq 1$ for all values of ϕ and y_{t-1}. In particular, the choice $tanh$ satisfies the requirement on σ. For higher order models, we follow an induction argument and show first that for a RNN(2) model we obtain

$$\begin{aligned} y_t &= \frac{1}{1 - \sigma(W_z\sigma(W_z L^2 + b) + W_x L + b)}[\epsilon_t] \\ &= \sum_{j=0}^{\infty} \sigma^j(W_z\sigma(W_z L^2 + b) + W_x L + b)[\epsilon_t], \end{aligned}$$

which again is stable if $|\sigma| \leq 1$ and it follows for any model order that the stability condition holds.

It follows that lagged unit impulses of the *data* strictly decay with the order of the lag when $|\sigma| \leq 1$. Again by induction, at lag 1, the output from the hidden layer is

$$z_t = \sigma(W_z \mathbf{1} + W_x \mathbf{0} + b) = \sigma(W_z \mathbf{1} + b). \tag{8.15}$$

The absolute value of each component of the hidden variable under a unit vector impulse at lag 1 is strictly less than 1:

$$|z_t|_j = |\sigma(W_z \mathbf{1} + b)|_j < 1, \tag{8.16}$$

if $|\sigma(x)| \leq 1$ and each element of $W_z \mathbf{1} + b$ is finite. Additionally if σ is strictly monotone increasing, then $|z_t|_j$ under a lag two unit innovation is strictly less than $|z_t|_j$ under a lag one unit innovation

$$|\sigma(W_z \mathbf{1}) + b)_j| > |\sigma(W_z \sigma(W_z \mathbf{1} + b) + b)|_j. \tag{8.17}$$

The implication of this stability result is the reassuring attribute that past random disturbances decay in the model and the effect of lagged data becomes less relevant to the model output with increasing lag.

2.3 Stationarity

For completeness, we mention in passing the extension of stationarity analysis in Sect. 2.4 to RNNs. The linear univariate RNN(p), considered above, has a companion matrix of the form

$$C := \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ \phi^{-p} & -\phi^{-p+1} & \dots & -\phi^{-2} & -\phi^{-1}, \end{pmatrix} \tag{8.18}$$

and it turns out that for $\phi \neq 0$ this model is non-stationary. We can hence rule out the choice of a linear activation since this would leave us with a linear RNN. Hence, it appears that some non-linear activation is necessary for the model to be stationary, but we cannot use the Cayley–Hamilton theorem to prove stationarity.

Half-Life

Suppose that the output of the RNN is in $\mathbb{R}^d$. The half-life of the lag is the smallest number of function compositions, k, of $\tilde{\sigma}(x) := \sigma(W_z x + b)$ with itself such that the normalized jth output is

$$r_j^{(k)} = \frac{(W_y \tilde{\sigma} \circ_1 \tilde{\sigma} \circ_2 \cdots \circ_{k-1} \tilde{\sigma}(\mathbf{1}) + b_y)_j}{(W_y \tilde{\sigma}(\mathbf{1}) + b_y)_j} \leq 0.5, k \geq 2, \forall j \in \{1, \ldots, d\}. \tag{8.19}$$

Note the output has been normalized so that the lag-1 unit impulse ensures that the ratio, $r_j^{(1)} = 1$ for each j. This modified definition exists to account for the effects of the activation function and the semi-affine transformation which are not present in AR(p) model. In general, there is no guarantee that the half-life is finite but we can find parameter values for which the half-life can be found. For example, suppose for simplicity that a univariate RNN is given by $\hat{x}_t = z_{t-1}$ and

$$z_t = \sigma(z_{t-1} + x_t).$$

Then the lag-1 impulse is $\hat{x}_t = \tilde{\sigma}(\mathbf{1}) = \sigma(\mathbf{0}+\mathbf{1})$, the lag-2 impulse is $\hat{x}_t = \sigma(\sigma(\mathbf{1})+\mathbf{0}) = \tilde{\sigma} \circ \tilde{\sigma}(\mathbf{1})$, and so on. If $\sigma(x) := tanh(x)$ and we normalize over the output from the lag-1 impulse to give the values in Table 8.1.

Table 8.1 The half-life characterizes the memory decay of the architecture by measuring the number of periods before a lagged unit impulse has at least half of its effect at lag 1. The calculation of the half-life involves nested composition of the recursion relation for the hidden layer until $r_j^{(k)}$ is less than a half. The calculations are repeated for each j, hence the half-life may vary depending on the component of the output. In this example, the half-life of the univariate RNN is 9 periods

Lag k	$r^{(k)}$
1	1.000
2	0.843
3	0.744
4	0.673
5	0.620
6	0.577
7	0.543
8	0.514
9	0.489

? Multiple Choice Question 1

Which of the following statements are true:

a. An augmented Dickey–Fuller test can be applied to time series to determine whether they are covariance stationary.

b. The estimated partial autocorrelation of a covariance stationary time series can be used to identify the design sequence length in a plain recurrent neural network.
c. Plain recurrent neural networks are guaranteed to be stable, namely lagged unit impulses decay over time.
d. The Ljung–Box test is used to test whether the fitted model residual error is autocorrelated.
e. The half-life of a lag-1 unit impulse is the number of lags before the impulse has half its effect on the model output.

2.4 Generalized Recurrent Neural Networks (GRNNs)

Classical RNNs, such as those described above, treat the error as homoscedastic—that is, the error is i.i.d. We mention in passing that we can generalize RNNs to heteroscedastic models by modifying the loss function to the squared Mahalanobis length of the residual vector. Such an approach is referred to here as generalized recurrent neural networks (GRNNs) and is mentioned briefly here with the caveat that the field of machine learning in econometrics is nascent and therefore incomplete and such a methodology, while appealing from a theoretic perspective, is not yet proven in practice. Hence the purpose of this subsection is simply to illustrate how more complex models can be developed which mirror some of the developments in parametric econometrics.

In its simplest form, we solve a weighted least squares minimization problem using data, $\mathcal{D}_t$:

$$\underset{W,b}{\text{minimize}} \qquad f(W,b) + \lambda\phi(W,b), \tag{8.20}$$

$$\mathcal{L}_\Sigma(Y,\hat{Y}) := (Y-\hat{Y})^T\Sigma^{-1}(Y-\hat{Y}),\ \Sigma_{tt} = \sigma_t^2,\ \Sigma_{tt'} = \rho_{tt'}\sigma_t\sigma_t', \tag{8.21}$$

$$f(W,b) \ = \ \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}_\Sigma(y_t,\hat{y}_t), \tag{8.22}$$

where $\Sigma := \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\mathbf{T}} \mid \mathcal{X}_t]$ is the conditional covariance matrix of the residual error and $\phi(W,b)$ is a regularization penalty term.

The conditional covariance matrix of the error must be estimated. This is performed as follows using the notation $()^T$ to denote the transpose of a vector and $()'$ to denote model parameters fitted under heteroscedastic error.

1) For each $t = 1,\ldots,T$, estimate the residual error over the training set, $\boldsymbol{\epsilon}_t \in \mathbb{R}^N$, using the standard (unweighted) loss function to find the weights, $\hat{W}_t$, and biases, $\hat{b}_t$ where the error is

$$\boldsymbol{\epsilon}_t = \mathbf{y_t} - F_{\hat{W}_t, \hat{b}_t}(\mathcal{X}_t). \tag{8.23}$$

2) The sample conditional covariance matrix $\hat{\Sigma}$ is estimated accordingly:

$$\hat{\Sigma} = \frac{1}{T-1} \sum_{i=1}^{T} \boldsymbol{\epsilon}_t \boldsymbol{\epsilon}_t^T. \tag{8.24}$$

3) Perform the generalized least squares minimization using Eq. 8.20 to obtain a fitted heteroscedastic neural network model, with refined error

$$\boldsymbol{\epsilon}'_t = \mathbf{y}_t - F_{\hat{W}'_t, \hat{b}'_t}(\mathcal{X}_t). \tag{8.25}$$

The fitted GRNN $F_{\hat{W}'_t, \hat{b}'_t}$ can then be used for forecasting without any further modification. The effect of the sample covariance matrix is to adjust the importance of the observation in the training set, based on the variance of its error and the error correlation. Such an approach can be broadly viewed as a RNN analogue of how GARCH models extend AR models. Of course, GARCH models treat the error distribution as parametric and provide a recurrence relation for forecasting the conditional volatility. In contrast, GRNNs rely on the empirical error distribution and do not forecast the conditional volatility. However, a separate regression could be performed over diagonals of the empirical conditional volatility Σ by using time series cross-validation.

3 Gated Recurrent Units

The extension of RNNs to dynamical time series models rests on extending foundational concepts in time series analysis. We begin by considering a smoothed RNN with hidden state $\hat{h}_t$. Such a RNN is almost identical to a plain RNN, but with an additional scalar smoothing parameter, α, which provides the network with "long memory."

3.1 α-RNNs

Let us consider a univariate α-RNN(p) model in which the smoothing parameter is fixed:

$$\hat{y}_{t+1} = W_y \hat{h}_t + b_y, \tag{8.26}$$

$$\hat{h}_t = \sigma(U_h \tilde{h}_{t-1} + W_h y_t + b_h), \tag{8.27}$$

$$\tilde{h}_t = \alpha \hat{h}_{t-1} + (1-\alpha)\tilde{h}_{t-1}, \tag{8.28}$$

with the starting condition in each sequence, $\hat{h}_{t-p+1} = y_{t-p+1}$. This model augments the plain RNN by replacing $\hat{h}_{t-1}$ in the hidden layer with an exponentially smoothed hidden state $\tilde{h}_{t-1}$. The effect of the smoothing is to provide infinite memory when $\alpha \neq 1$. For the special case when $\alpha = 1$, we recover the plain RNN with short memory of length p .

We can easily study this model by simplifying the parameterization and considering the unactivated case. Setting $b_y = b_h = 0$, $U_h = W_h = \phi$ and $W_y = 1$:

$$\hat{y}_{t+1} = \hat{h}_t, \tag{8.29}$$

$$= \phi(\tilde{h}_{t-1} + y_t), \tag{8.30}$$

$$= \phi(\alpha \hat{h}_{t-1} + (1-\alpha)\tilde{h}_{t-2} + y_t). \tag{8.31}$$

Without loss of generality, consider $p = 2$ lags in the model so that $\hat{h}_{t-1} = \phi y_{t-1}$. Then

$$\hat{h}_t = \phi(\alpha\phi y_{t-1} + (1-\alpha)\tilde{h}_{t-2} + y_t) \tag{8.32}$$

and the model can be written in the simpler form

$$\hat{y}_{t+1} = \phi_1 y_t + \phi_2 y_{t-1} + \phi(1-\alpha)\tilde{h}_{t-2}, \tag{8.33}$$

with autoregressive weights $\phi_1 := \phi$ and $\phi_2 := \alpha\phi^2$. We now see, in comparison with an AR(2) model, that there is an additional term which vanishes when $\alpha = 1$ but provides infinite memory to the model since $\tilde{h}_{t-2}$ depends on y_0, the first observation in the whole time series, not just the first observation in the sequence. The α-RNN model can be trained by treating α as a hyperparameter. The choice to fix α is obviously limited to stationary time series. We can extend the model to non-stationary time series by using a dynamic version of exponential smoothing.

3.1.1 Dynamic α_t-RNNs

Dynamic exponential smoothing is a time-dependent, convex, combination of the smoothed output, $\tilde{y}_t$, and the observation y_t:

$$\tilde{y}_{t+1} = \alpha_t y_t + (1-\alpha_t)\tilde{y}_t, \tag{8.34}$$

where $\alpha_t \in [0, 1]$ denotes the dynamic smoothing factor which can be equivalently written in the one-step-ahead forecast of the form

$$\tilde{y}_{t+1} = \tilde{y}_t + \alpha_t(y_t - \tilde{y}_t). \tag{8.35}$$

Hence the smoothing can be viewed as a form of dynamic forecast error correction; When $\alpha_t = 0$, the forecast error is ignored and the smoothing merely repeats the current hidden state $\tilde{h}_t$ to the effect of the model losing its memory. When $\alpha_t = 1$, the forecast error overwrites the current hidden state $\tilde{h}_t$.

The smoothing can also be viewed a weighted sum of the lagged observations, with lower or equal weights, $\alpha_{t-s}\prod_{r=1}^{s}(1-\alpha_{t-r+1})$ at the lag $s \geq 1$ past observation, y_{t-s}:

$$\tilde{y}_{t+1} = \alpha_t y_t + \sum_{s=1}^{t-1} \alpha_{t-s} \prod_{r=1}^{s} (1-\alpha_{t-r+1}) y_{t-s} + \prod_{r=0}^{t-1} (1-\alpha_{t-r}) \tilde{y}_1, \tag{8.36}$$

where the last term is a time-dependent constant and typically we initialize the exponential smoother with $\tilde{y}_1 = y_1$. Note that for any $\alpha_{t-r+1} = 1$, the prediction $\tilde{y}_{t+1}$ will have no dependency on all lags $\{y_{t-s}\}_{s \geq r}$. The model simply forgets the observations at or beyond the rth lag.

In the special case when the smoothing is constant and equal to $1-\alpha$, then the above expression simplifies to

$$\tilde{y}_{t+1} = \alpha \Phi(L)^{-1} y_t, \tag{8.37}$$

or equivalently written as a AR(1) process in $\tilde{y}_{t+1}$:

$$\Phi(L)\tilde{y}_{t+1} = \alpha y_t, \tag{8.38}$$

for the linear operator $\Phi(z) := 1 + (\alpha - 1)z$ and where L is the lag operator.

3.2 Neural Network Exponential Smoothing

Let us suppose now that instead of smoothing the observed time series $\{y_s\}_{s \leq 1}$, we instead smooth a hidden vector $\hat{h}_t$ with $\hat{\alpha}_t \in [0,1]^H$ to give a filtered time series

$$\tilde{h}_t = \hat{\alpha}_t \circ \hat{h}_t + (1 - \hat{\alpha}_t) \circ \tilde{h}_{t-1}, \tag{8.39}$$

where $\circ$ denotes the Hadamard product between vectors. This smoothing is a vectorized form of the above classical setting, only here we note that when $(\alpha_t)_i = 1$, the ith component of the hidden variable is unmodified and the past filtered hidden variable is forgotten. On the other hand, when the $(\alpha_t)_i = 0$, the ith component of the hidden variable is obsolete, instead setting the current filtered hidden variable to its past value. The smoothing in Eq. 8.39 can be viewed then as updating long-term memory, maintaining a smoothed hidden state variable as the memory through a convex combination of the current hidden variable and the previous smoothed hidden variable.

The hidden variable is given by the semi-affine transformation:

$$\hat{h}_t = \sigma(U_h \tilde{h}_{t-1} + W_h x_t + b_h) \tag{8.40}$$

which in turns depends on the previous smoothed hidden variable. Substituting Eq. 8.40 into Eq. 8.39 gives a function of $\tilde{h}_{t-1}$ and x_t:

$$\tilde{h}_t = g(\tilde{h}_{t-1}, x_t; \alpha) := \hat{\alpha}_t \circ \sigma(U_h \tilde{h}_{t-1} + W_h x_t + b_h) + (1 - \hat{\alpha}_t) \circ \tilde{h}_{t-1}. \tag{8.41}$$

We see that when $\alpha_t = 0$, the smoothed hidden variable $\tilde{h}_t$ is not updated by the input x_t. Conversely, when $\alpha_t = 1$, we observe that the hidden variable locally behaves like a non-linear autoregressive series. Thus the smoothing parameter can be viewed as the sensitivity of the smoothed hidden state to the input x_t.

The challenge becomes how to determine dynamically how much error correction is needed. GRUs address this problem by learning $\hat{\boldsymbol{\alpha}} = F_{(W_\alpha, U_\alpha, b_\alpha)}(X)$ from the input variables with a plain RNN parameterized by weights and biases $(W_\alpha, U_\alpha, b_\alpha)$. The one-step-ahead forecast of the smoothed hidden state, $\tilde{h}_t$, is the filtered output of another plain RNN with weights and biases (W_h, U_h, b_h). Putting this together gives the following $\alpha - t$ model (simple GRU):

$$\text{smoothing} : \tilde{h}_t = \hat{\alpha}_t \circ \hat{h}_t + (1 - \hat{\alpha}_t) \circ \tilde{h}_{t-1} \tag{8.42}$$

$$\text{smoother update} : \hat{\alpha}_t = \sigma^{(1)}(U_\alpha \tilde{h}_{t-1} + W_\alpha x_t + b_\alpha) \tag{8.43}$$

$$\text{hidden state update} : \hat{h}_t = \sigma(U_h \tilde{h}_{t-1} + W_h x_t + b_h), \tag{8.44}$$

where $\sigma^{(1)}$ is a sigmoid or Heaviside function and σ is any activation function. Figure 8.2 shows the response of a α_t-RNN when the input consists of two unit impulses. For simplicity, the sequence length is assumed to be 3 (i.e., the RNN has a memory of 3 lags), the biases are set to zero, all the weights are set to one, and $\sigma(x) := tanh(x)$. Note that the weights have not been fitted here, we are merely observing the effect of smoothing on the hidden state for the simplest choice of parameter values. The RNN loses memory of the unit impulse after three lags, whereas the RNNs with smooth hidden states maintain memory of the first unit impulse even when the second unit impulse arrives. The difference between the dynamically smoothed RNN (the α_t-RNN) and α-RNN with a fixed smoothing parameter appears insignificant. Keep in mind however that the dynamical smoothing model has much more flexibility in how it controls the sensitivity of the smoothing to the unit impulses.

In the above α_t-RNN, there is no means to directly occasionally forget the memory. This is because the hidden variables update equation always depends on the previous smoothed hidden state, unless $U_h = 0$. However, it can be expected that the fitted recurrence weight $\hat{U}_h$ will not in general be zero and thus the model is without a "hard reset button."

GRUs also have the capacity to entirely reset the memory by adding an additional reset variable:

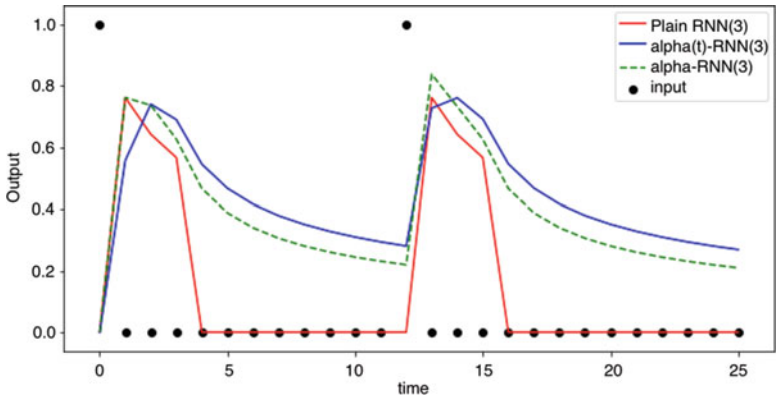


Fig. 8.2 An illustrative example of the response of an α_t-RNN and comparison with a plain RNN and a RNN with an exponentially smoothed hidden state, under a constant α (α-RNN). The RNN(3) model loses memory of the unit impulse after three lags, whereas the α-$RNN(3)$ models maintain memory of the first unit impulse even when the second unit impulse arrives. The difference between the α_t-RNN (the toy GRU) and the α-RNN appears insignificant. Keep in mind however that the dynamical smoothing model has much more flexibility in how it controls the sensitivity of the smoothing to the unit impulses

$$\text{smoothing} : \tilde{h}_t = \hat{\alpha}_t \circ \hat{h}_t + (1 - \hat{\alpha}_t) \circ \tilde{h}_{t-1} \tag{8.45}$$

$$\text{smoother update} : \hat{\alpha}_t = \sigma^{(1)}(U_\alpha \tilde{h}_{t-1} + W_\alpha x_t + b_\alpha) \tag{8.46}$$

$$\text{hidden state update} : \hat{h}_t = \sigma(U_h \hat{r}_t \circ \tilde{h}_{t-1} + W_h x_t + b_h) \tag{8.47}$$

$$\text{reset update} : \hat{r}_t = \sigma^{(1)}(U_r \tilde{h}_{t-1} + W_r x_t + b_r). \tag{8.48}$$

The effect of introducing a reset, or switch, $\hat{r}_t$, is to forget the dependence of $\hat{h}_t$ on the smoothed hidden state. Effectively, we turn the update for $\hat{h}_t$ from a plain RNN to a FFN and entirely neglect the recurrence. The recurrence in the update of $\hat{h}_t$ is thus dynamic. It may appear that the combination of a reset and adaptive smoothing is redundant. But remember that $\hat{\alpha}_t$ effects the level of error correction in the update of the smoothed hidden state, $\tilde{h}_t$, whereas $\hat{r}_t$ adjusts the level of recurrence in the unsmoothed hidden state $\hat{h}_t$. Put differently, $\hat{\alpha}_t$ by itself cannot disable the memory in the smoothed hidden state (internal memory), whereas $\hat{r}_t$ in combination with $\hat{\alpha}_t$ can. More precisely, when $\alpha_t = 1$ and $\hat{r}_t = 0$, $\tilde{h}_t = \hat{h}_t = \sigma(W_h x_t + b_h)$ which is reset to the latest input, x_t, and the GRU is just a FFNN. Also, when $\alpha_t = 1$ and $\hat{r}_t > 0$, a GRU acts like a plain RNN. Thus a GRU can be seen as a more general architecture which is capable of being a FFNN or a plain RNN under certain parameter values.

These additional layers (or cells) enable a GRU to learn extremely complex long-term temporal dynamics that a plain RNN is not capable of. The price to pay for this flexibility is the additional complexity of the model. Clearly, one must choose whether to opt for a simpler model, such as an α_t-RNN, or use a GRU. Lastly, we

comment in passing that in the GRU, as in a RNN, there is a final feedforward layer to transform the (smoothed) hidden state to a response:

$$\hat{y}_t = W_Y \tilde{h}_t + b_Y. \tag{8.49}$$

3.3 Long Short-Term Memory (LSTM)

The GRU provides a gating mechanism for propagating a smoothed hidden state—a long-term memory—which can be overridden and even turn the GRU into a plain RNN (with short memory) or even a memoryless FFN. More complex models using hidden units with varying connections within the memory unit have been proposed in the engineering literature with empirical success (Hochreiter and Schmidhuber 1997; Gers et al. 2001; Zheng et al. 2017). LSTMs are similar to GRUs but have a separate (cell) memory, C_t, in addition to a hidden state h_t. LSTMs also do not require that the memory updates are a convex combination. Hence they are more general than exponential smoothing. The mathematical description of LSTMs is rarely given in an intuitive form, but the model can be found in, for example, Hochreiter and Schmidhuber (1997).

The cell memory is updated by the following expression involving a forget gate, $\hat{\alpha}_t$, an input gate $\hat{z}_t$, and a cell gate $\hat{c}_t$

$$c_t = \hat{\alpha}_t \circ c_{t-1} + \hat{z}_t \circ \hat{c}_t. \tag{8.50}$$

In the language of LSTMs, the triple $(\hat{\alpha}_t, \hat{r}_t, \hat{z}_t)$ are, respectively, referred to as the forget gate, output gate, and input gate. Our change of terminology is deliberate and designed to provide more intuition and continuity with GRUs and econometrics. We note that in the special case when $\hat{z}_t = 1 - \hat{\alpha}_t$ we obtain a similar exponential smoothing expression to that used in the GRU. Beyond that, the role of the input gate appears superfluous and difficult to reason with using time series analysis. Likely it merely arose from a contextual engineering model; however, it is tempting to speculate how the additional variable provides the LSTM with a more elaborate representation of complex temporal dynamics.

When the forget gate, $\hat{\alpha}_t = 0$, then the cell memory depends solely on the cell memory gate update $\hat{c}_t$. By the term $\hat{\alpha}_t \circ c_{t-1}$, the cell memory has long-term memory which is only forgotten beyond lag s if $\hat{\alpha}_{t-s} = 0$. Thus the cell memory has an adaptive autoregressive structure.

The extra "memory," treated as a hidden state and separate from the cell memory, is nothing more than a Hadamard product:

$$h_t = \hat{r}_t \circ tanh(c)_t, \tag{8.51}$$

which is reset if $\hat{r}_t = 0$. If $\hat{r}_t = 1$, then the cell memory directly determines the hidden state.

Thus the reset gate can entirely override the effect of the cell memory's autoregressive structure, without erasing it. In contrast, the GRU has one memory, which serves as the hidden state, and it is directly affected by the reset gate.

The reset, forget, input, and cell memory gates are updated by plain RNNs all depending on the hidden state h_t.

$$\text{Reset gate} : \hat{r}_t = \sigma(U_r h_{t-1} + W_r x_t + b_r) \tag{8.52}$$

$$\text{Forget gate} : \hat{\alpha}_t = \sigma(U_\alpha h_{t-1} + W_\alpha x_t + b_\alpha) \tag{8.53}$$

$$\text{Input gate} : \hat{z}_t = \sigma(U_z h_{t-1} + W_z x_t + b_z) \tag{8.54}$$

$$\text{Cell memory gate} : \hat{c}_t = \tanh(U_c h_{t-1} + W_c x_t + b_c). \tag{8.55}$$

Like the GRU, the LSTM can function as a short memory, plain RNN; just set $\alpha_t = 0$ in Eq. 8.50. However, the LSTM can also function as a coupling of FFNs; just set $\hat{r}_t = 0$ so that $h_t = 0$ and hence there is no recurrence structure in any of the gates. Both GRUs and LSTMs, even if the nomenclature does not suggest it, can model long- and short-term autoregressive memory. The GRU couple these through a smoothed hidden state variable. The LSTM separates out the long memory, stored in the cellular memory, but uses a copy of it, which may additionally be reset. Strictly speaking, the cellular memory has long-short autoregressive memory structure, so it would be misleading in the context of time series analysis to strictly discern the two memories as long and short (as the nomenclature suggests). The latter can be thought of as a truncated version of the former.

? Multiple Choice Question 2

Which of the following statements are true:

a. A gated recurrent unit uses dynamic exponential smoothing to propagate a hidden state with infinite memory.
b. The gated recurrent unit requires that the data is covariance stationary.
c. Gated recurrent units are unconditionally stable, for any choice of activation functions and weights.
d. A GRU only has one memory, the hidden state, whereas a LSTM has an additional, cellular, memory.

4 Python Notebook Examples

The following Python examples demonstrate the application of RNNs and GRUs to financial time series prediction.

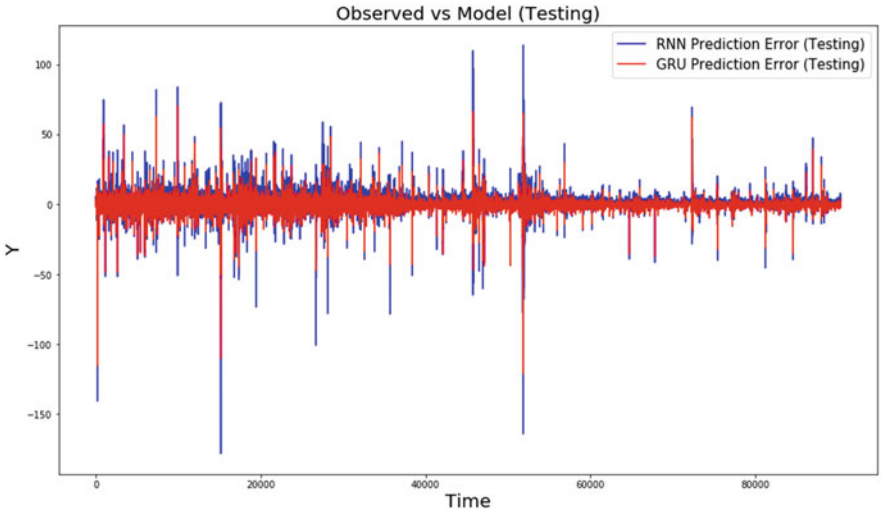


Fig. 8.3 A comparison of out-of-sample forecasting errors produced by a RNN and GRU trained on minute snapshots of Coinbase mid-prices

4.1 Bitcoin Prediction

`ML_in_Finance-RNNs-Bitcoin.ipynb` provides an example of how `TensorFlow` can be used to train and test RNNs for time series prediction. The example dataset is for predicting minute head mid-prices from minute snapshots of the USD value of Coinbase over 2018.

Statistical methods for stationarity and autocorrelation shall be used to characterize the data, identify the sequence length needed in the RNN, and to diagnose the model error. Here we accept the Null as the p-value is larger than 0.01 in absolute value and thus we cannot reject the ADF test at the 99% confidence level. Since plain RNNs are not suited to non-stationary time series modeling, we can use a GRU or LSTM to model non-stationarity data, since these models exhibit dynamic autocorrelation structure. Figure 8.3 compares the out-of-sample forecasting errors produced by a RNN and GRU. See the notebook for further details of the architecture and experiment.

4.2 Predicting from the Limit Order Book

The dataset is tick-by-tick, top of the limit order book, data such as mid-prices and volume weighted mid-prices (VWAP) collected from ZN futures. This dataset is heavily truncated for demonstration purposes and consists of 1033492 observations. The data has also been labeled to indicate whether the prices up-tick (1), remain

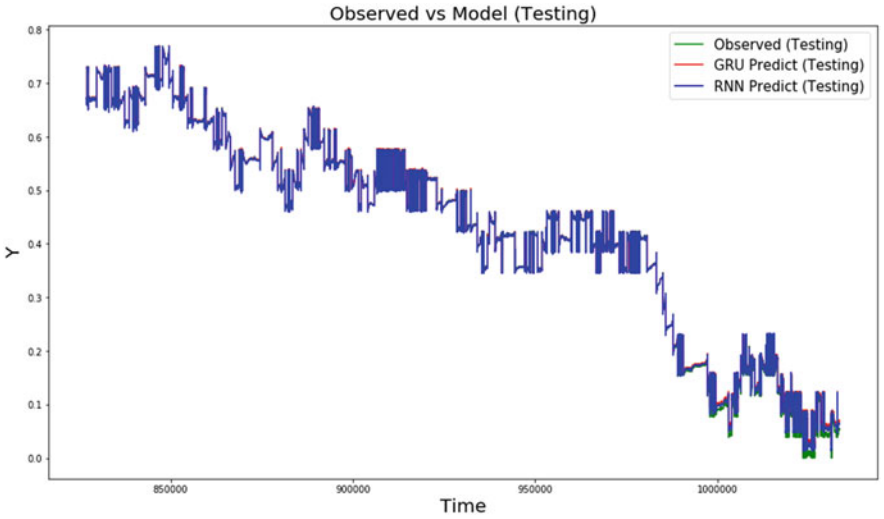


Fig. 8.4 A comparison of out-of-sample forecasting errors produced by a plain RNN and GRU trained on tick-by-tick smart prices of ZN futures

the same, or down-tick (-1) over the next tick. For demonstration purposes, the timestamps have been removed. In the simple forecasting experiment, we predict VWAPs (a.k.a. "smart prices") from historical smart prices. Note that a classification experiment is also possible but not shown here.

The ADF test is performed over the first 200k observations as it is computationally intensive to apply it to the entire dataset. The ADF test statistic is -3.9706 and the p-value is smaller in absolute value than 0.01 and we thus reject the Null of the ADF test at the 99% confidence level in favor of the data being stationary (i.e., there are no unit roots). The Ljung–Box test is used to identify the number of lags needed in the model. A comparison of out-of-sample VWAP prices produced by a plain RNN and GRU is shown in Fig. 8.4. Because the data is stationary, we observe little advantage in using a GRU over a plain RNN. See `ML_in_Finance-RNNs-HFT.ipynb` for further details of the network architectures and experiment.

5 Convolutional Neural Networks

Convolutional neural networks (CNNs) are feedforward neural networks that can exploit local spatial structures in the input data. Flattening high-dimensional time series, such as limit order book depth histories, would require a very large number of weights in a feedforward architecture. CNNs attempt to reduce the network size by exploiting data locality (Fig. 8.5).

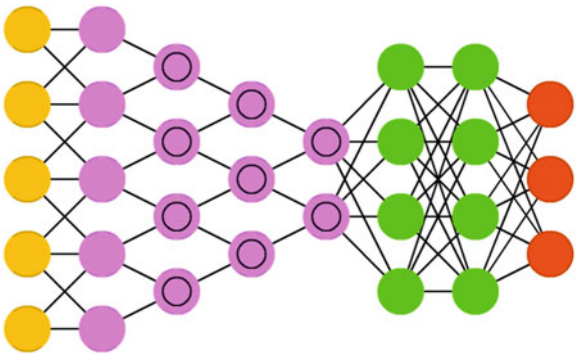


Fig. 8.5 Convolutional neural networks. Source: Van Veen, F. & Leijnen, S. (2019), "The Neural Network Zoo", Retrieved from https://www.asimovinstitute.org/neural-network-zoo

Deep CNNs, with multiple consecutive convolutions followed by non-linear functions, have shown to be immensely successful in image processing (Krizhevsky et al. 2012). We can view convolutions as spatial filters that are designed to select a specific pattern in the data, for example, straight lines in an image. For this reason, convolution is frequently used for image processing, such as for smoothing, sharpening, and edge detection of images. Of course, in financial modeling, we typically have different spatial structures, such as the limit order book depths or the implied volatility surface of derivatives. However, the CNN has established its place in time series analysis too.

5.1 Weighted Moving Average Smoothers

A common technique in time series analysis and signal processing is to filter the time series. We have already seen exponential smoothing as a special case of a class of smoothers known as "weighted moving average (WMA)" smoothers. WMA smoothers take the form

$$\tilde{x}_t = \frac{1}{\sum_{i\in\mathcal{I}} w_i} \sum_{i\in\mathcal{I}} w_i x_{t-i}, \tag{8.56}$$

where $\tilde{x}_t$ is the local mean of the time series. The weights are specified to emphasize or deemphasize particular observations of x_{t-i} in the span $|\mathcal{I}|$. Examples of well-known smoothers include the Hanning smoother $h(3)$:

$$\tilde{x}_t = (x_{t-1} + 2x_t + x_{t+1})/4. \tag{8.57}$$

Such smoothers have the effect of reducing noise in the time series. The moving average filter is a simple low pass finite impulse response (FIR) filter commonly used for regulating an array of sampled data. It takes $|\mathcal{I}|$ samples of input at a time and takes the weighted average of those to produce a single output point. As the

length of the filter increases, the smoothness of the output increases, whereas the sharp modulations in the data are smoothed out.

The moving average filter is in fact a convolution using a very simple filter kernel. More generally, we can write a univariate time series prediction problem as a convolution with a filter as follows. First, the discrete convolution gives the relation between x_i and x_j:

$$x_{t-i} = \sum_{j=0}^{t-1} \delta_{ij} x_{t-j}, \ i \in \{0, \ldots, t-1\} \tag{8.58}$$

where we have used the Kronecker delta δ. The kernel filtered time series is a convolution

$$\tilde{x}_{t-i} = \sum_{j \in J} K_{j+k+1} x_{t-i-j}, i \in \{k+1, \ldots, p-k\}, \tag{8.59}$$

where $J := \{-k, \ldots, k\}$ so that the span of the filter $|J| = 2k+1$, where k is taken as a small integer, and the kernel is K. For simplicity, the ends of the sequence are assumed to be unfiltered but for notational reasons we set $\tilde{x}_{t-i} = x_{t-i}$ for $i \in \{1, \ldots, k, p-k+1, \ldots, p\}$. Then the filtered $AR(p)$ model is

$$\hat{x}_t = \mu + \sum_{i=1}^{p} \phi_i \tilde{x}_{t-i} \tag{8.60}$$

$$= \mu + (\phi_1 L + \phi_2 L^2 + \cdots + \phi_p L^p)[\tilde{x}_t] \tag{8.61}$$

$$= \mu + [L, L^2, \ldots, L^p]\boldsymbol{\phi}[\tilde{x}_t], \tag{8.62}$$

with coefficients $\boldsymbol{\phi} := [\phi_i, \ldots, \phi_p]$. Note that there is no look-ahead bias because we do not filter the last k values of the observed data $\{x_s\}_{s=1}^{t}$. We have just written our first toy 1D CNN consisting of a feedforward output layer and a non-activated hidden layer with one unit (i.e., kernel):

$$\hat{x}_t = W_y z_t + b_y, z_t = [\tilde{x}_{t-1}, \ldots, \tilde{x}_{t-p}]^T, \ W_y = \boldsymbol{\phi}^T, \ b_y = \mu, \tag{8.63}$$

where $\tilde{x}_{t-i}$ is the ith output from a convolution of the p length input sequence with a kernel consisting of $2k+1$ weights. These weights are fixed over time and hence the CNN is only suited to prediction from stationary time series. Note also, in contrast to a RNN, that the size of the weight matrix W_y increases with the number of lags in the model.

The univariate CNN predictor with p lags and H activated hidden units (kernels) is

$$\hat{x}_t = W_y \text{vec}(z_t) + b_y \tag{8.64}$$

$$[z_t]_{i,m} = \sigma(\sum_{j \in J} K_{m,j+k+1} x_{t-i-j} + [b_h]_m) \tag{8.65}$$

$$= \sigma(K * x_t + b_h), \tag{8.66}$$

where $m \in \{1, \ldots, H\}$ denotes the index of the kernel and the kernel matrix $K \in \mathbb{R}^{H \times 2k+1}$, hidden bias vector $b_h \in \mathbb{R}^H$ and output matrix $W_y \in \mathbb{R}^{1 \times pH}$.

Dimension Reduction

Since the size of W_y increases with both the number of lags and the number of kernels, it may be preferable to reduce the dimensionality of the weights with an additional layer and hence avoid over-fitting. We will return to this concept later, but one might view it as an alternative to auto-shrinkage or dropout.

Non-sequential Models

Convolutional neural networks are not limited to sequential models. One might, for example, sample the past lags non-uniformly so that $I = \{2^i\}_{i=1}^p$ then the maximum lag in the model is 2^p. Such a non-sequential model allows a large maximum lag without capturing all the intermediate lags. We will also return to non-sequential models in the section on dilated convolution.

Stationarity

A univariate CNN predictor, with one kernel and no activation, can be written in the canonical form

$$\hat{x}_t = \mu + (1 - \Phi(L))[K * x_t] = \mu + K * (1 - \Phi(L))[x_t] \tag{8.67}$$

$$= \mu + (\tilde{\phi}_1 L + \ldots \tilde{\phi}_p L^p)[x_t] \tag{8.68}$$

$$:= \mu + (1 - \tilde{\Phi}(L))[x_t], \tag{8.69}$$

where, by the linearity of $\Phi(L)$ in x_t, the convolution commutes and thus we can write $\tilde{\phi} := K * \phi$. Finding the roots of the characteristic equation

$$\tilde{\Phi}(z) = 0, \tag{8.70}$$

it follows that the CNN is strictly stationary and ergodic if all the roots lie outside the unit circle in the complex plane, $|\lambda_i| > 1,\ i \in \{1, \ldots, p\}$. As before, we would compute the eigenvalues of the companion matrix to find the roots. Provided that $\tilde{\Phi}(L)^{-1}$ forms a divergent sequence in the noise process $\{\epsilon_s\}_{s=1}^t$ then the model is *stable*.

5.2 2D Convolution

2D convolution involves applying a small kernel matrix (a.k.a. a filter), $K \in \mathbb{R}^{2k+1 \times 2k+1}$, over the input matrix (called an image), $X \in \mathbb{R}^{m \times n}$, to give a filtered image, $Y \in \mathbb{R}^{m-2k \times n-2k}$. In the context of convolutional neural networks, the elements of the filtered image are referred to as the *feature map* values and are calculated according to the following formula:

$$y_{i,j} = [K * X]_{i,j} = \sum_{p,q=-k}^{k} K_{k+1+p,k+1+q} x_{i+p+1,j+q+1},$$
$$i \in \{1, \ldots, m\}, j \in \{1, \ldots .n\}. \tag{8.71}$$

It is instructive to consider the following example to illustrate the 2D convolution with a small kernel matrix.

Example 8.2 2D Convolution

Consider the 4×4 input, 3×3 kernel, and 2×2 output matrices $X = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 3 \\ 2 & 0 & 1 & 0 \\ 0 & 2 & 1 & 0 \end{bmatrix}$, $K = \begin{bmatrix} 0 & -1 & 1 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \end{bmatrix}$, $Y = \begin{bmatrix} 2 & 1 \\ -2 & 5 \end{bmatrix}$. The calculation of the outputs for the case when $i = j = 1$ is

$$\begin{aligned} y_{i,j} = [K * X]_{i,j} &= \sum_{p,q=-k}^{k} K_{k+1+p,k+1+q} x_{i+p+1,j+q+1}, \\ & i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\} \\ &= 0 \cdot 1 + -1 \cdot 0 + 1 \cdot 0 + 0 \cdot 0 + 1 \cdot 0 + 0 \cdot 0 + 1 \cdot 2 + -1 \cdot 0 + 0 \cdot 1 \\ &= 2 \end{aligned}$$

We leave it as an exercise for the reader to compute the output for the remaining values of i and j.

As in the example above, when we perform convolution over the 4×4 image with a 3×3 kernel, we get a 2×2 feature map. This is because there are only 4 unique positions where we can place our filter inside this image.

As convolutional neural networks were designed for image processing, it is common to represent the color values of the pixels with c color channels. For

example, RGB values are represented with three channels. The general form of the convolution layer map for a $m \times n \times c$ input tensor and outputs $m \times n \times H$ (with stride 1 and padding) is

$$\theta : \mathbb{R}^{m\times n\times c} \to \mathbb{R}^{m\times n\times H}.$$

Writing

$$f = \begin{pmatrix} f_1 \\ \vdots \\ f_c \end{pmatrix} \tag{8.72}$$

we can then write the layer map as

$$\theta(f) = K * f + \mathbf{b}, \tag{8.73}$$

where $K \in \mathbb{R}^{[(2k+1)\times(2k+1)]\times H\times c}$, $\mathbf{b} \in \mathbb{R}^{m\times n\times H}$ given by $\mathbf{b} := \mathbf{1}_{m\times n} \otimes b$ and $\mathbf{1}_{m\times n}$ is a $m \times n$ matrix with all elements being 1.

In component form, the operation (8.73) is

$$[\theta(f)]_j = \sum_{i=1}^{c} [K]_{i,j} * [f]_i + \mathbf{b}_j, \quad j \in \{1, \ldots, H\}, \tag{8.74}$$

where $[\cdot]_{i,j}$ contracts the 4-tensor to a 2-tensor by indexing the ith third component and jth fourth component of the tensor and for any $g \in \mathbb{R}^{m\times n}$ and $H \in \mathbb{R}^{(2k+1)\times(2k+1)}$

$$[H * g]_{i,j} = \sum_{p,q=-k}^{k} H_{k+1+p,k+1+q} g_{i+p,j+q}, \; i \in \{1, \ldots, m\}, \; j \in \{1, \ldots .n\}. \tag{8.75}$$

By analogy to a fully connected feedforward architecture, the weights in the layer are given by the kernel tensor, K, and the biases, b are H-vectors. Instead of a semi-affine transformation, the layer is given by an activated convolution $\sigma(\boldsymbol{\theta}(\mathbf{f}))$.

Furthermore, we note that not all neurons in the two consecutive layers are connected to each other. In fact, only the neurons which correspond to inputs within a $2k + 1 \times 2k + 1$ square connect to the same output neuron. Thus the filter size controls the *receptive field* of each output. We note, therefore, that some neurons share the same weights. Both of these properties result in far fewer parameters to learn than a fully connected feedforward architecture.

Padding is needed to extend the size of the image f so that the filtered image has the same dimensions as the original image. Specifically padding means how to choose $f_{i+p,j+q}$ when $(i + p, j + q)$ is outside of $\{1, \ldots, m\}$ or $\{1, \ldots, n\}$. The following three choices are often used

$$f_{i+p,j+q} = \begin{cases} 0, & \text{zero padding}, \\ f_{(i+p) \pmod m,(s+q) \pmod n}, & \text{periodic padding}, \\ f_{|i-1+p|,|j-1+q|}, & \text{reflected padding}, \end{cases} \tag{8.76}$$

if

$$i + p \notin \{1, \ldots, m\} \text{ or } j + q \notin \{1, \ldots, n\}. \tag{8.77}$$

Here $d \pmod m \in \{1, \cdots, m\}$ means the remainder when d is divided by m.

The operation in Eq. 8.75 is also called a convolution with stride 1. Informally, we performed the convolution by sliding the image area by a unit increment. A common choice in CNNs is to take $s = 2$. Given an integer $s \geq 1$, a convolution with stride s for $f \in \mathbb{R}^{m \times n}$ is defined as

$$[K *_s f]_{i,j} = \sum_{p,q=-k}^{k} K_{p,q} f_{s(i-1)+1+p,s(j-1)+1+q}, \; i \in \{1, \ldots, \lceil \frac{m}{s} \rceil\},$$
$$j \in \{1, \ldots, \lceil \frac{n}{s} \rceil\}. \tag{8.78}$$

Here $\lceil \frac{m}{s} \rceil$ denotes the smallest integer greater than $\frac{m}{s}$.

5.3 Pooling

Data with high spatial structure often results in observations which have similar values within a neighborhood. Such a characteristic leads to redundancy in data representation and motivates the use of data reduction techniques such as pooling. In addition to a convolution layer, a pooling layer is a map:

$$\bar{R}_{\ell}^{\ell+1} : \mathbb{R}^{m_\ell \times n_\ell} \to \mathbb{R}^{m_{\ell+1} \times n_{\ell+1}}. \tag{8.79}$$

One popular pooling is the so-called average pooling R_{avr} which can be a convolution with stride 2 or bigger using the kernel K in the form of

$$K = \frac{1}{9} \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{8.80}$$

Non-linear pooling operator is also used, for example, the $(2k + 1) \times (2k + 1)$ max-pooling operator with stride s as follows:

$$[R_{\max}(f)]_{i,j} = \max_{-k \leq p,q \leq k} \{f_{s(i-1)+1+p,s(j-1)+1+q}\}. \tag{8.81}$$

5.4 Dilated Convolution

In addition to image processing, CNNs have also been successfully applied to time series. WaveNet, for example, is a CNN developed for audio processing (van den Oord et al. 2016).

Time series often displays long-term correlations. Moreover, the dependent variable(s) may exhibit non-linear dependence on the lagged predictors. The WaveNet architecture is a non-linear p-autoregression of the form

$$y_t = \sum_{i=1}^{p} \phi_i(x_{t-i}) + \epsilon_t \tag{8.82}$$

where the coefficient functions $\phi_i, i \in \{1, \ldots, p\}$ are data-dependent and optimized through the convolutional network.

To enable the network to learn these long-term, non-linear, dependencies Borovykh et al. (2017) use stacked layers of dilated convolutions. A dilated convolution effectively allows the network to operate on a coarser scale than with a normal convolution. This is similar to pooling or strided convolutions, but here the output has the same size as the input (van den Oord et al. 2016).

In a dilated convolution the filter is applied to every dth element in the input vector, allowing the model to efficiently learn connections between far-apart data points. For an architecture with L layers of dilated convolutions $\ell \in \{1, \ldots, L\}$, a dilated convolution outputs a stack of "feature maps" given by

$$[K^{(\ell)} *_{d^{(\ell)}} f^{(\ell-1)}]_i = \sum_{p=-k}^{k} K_p^{(\ell)} f^{(\ell-1)}_{d^{(\ell)}(i-1)+1+p}, \; i \in \{1, \ldots, \lceil \frac{m}{d^{(\ell)}} \rceil\}, \tag{8.83}$$

where d is the dilation factor and we can choose the dilations to increase by a factor of two: $d^{(\ell)} = 2^{\ell-1}$. The filters for each layer, $K^{(\ell)}$, are chosen to be of size $1 \times (2k+1) = 1 \times 2$.

An example of a three-layer dilated convolutional network is shown in Fig. 8.6. Using the dilated convolutions instead of regular ones allows the output y to be influenced by more nodes in the input. The input of the network is given by the time series X. In each subsequent layer we apply the dilated convolution, followed by a non-linearity, giving the output feature maps $f^{(\ell)}, \ell \in \{1, \ldots, L\}$.

Since we are interested in forecasting the subsequent values of the time series, we will train the model so that this output is the forecasted time series $\hat{Y} = \{\hat{y}_t\}_{t=1}^{N}$.

The receptive field of a neuron was defined as the set of elements in its input that modifies the output value of that neuron. Now, we define the receptive field r of the model to be the number of neurons in the input in the first layer, i.e. the time series, that can modify the output in the final layer, i.e. the forecasted time series. This then depends on the number of layers L and the filter size $2k+1$, and is given by

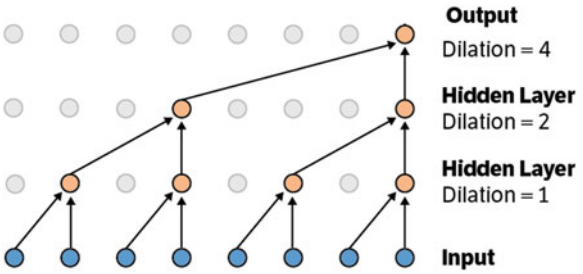


Fig. 8.6 A dilated convolutional neural network with three layers. The receptive field is given by $r = 8$, i.e. one output value is influenced by eight input neurons. Source: van den Oord et al. (2016)

$$r := 2^{L-1}(2k + 1). \tag{8.84}$$

In Fig. 8.6, the receptive field is given by $r = 8$, one output value is influenced by eight input neurons.

? Multiple Choice Question 3

Which of the following statements are true:

a. CNNs apply a collection of different, but equal width, filters to the data before using a feedforward network for regression or classification.
b. CNNs are sparse networks, exploiting locality of the data, to reduce the number of weights.
c. A dilated CNN is appropriate for multi-scale time series analysis—it captures a hierarchy of patterns at different resolutions (i.e., dependencies on past lags at different frequencies, e.g. days, weeks, months)
d. The number of layers in a CNN is automatically determined during training.

5.5 *Python Notebooks*

`ML_in_Finance-1D-CNNs.ipynb` demonstrates the application of 1D CNNs to predict the next element in a uniform sequence of integers. The CNN uses a sequence length of 50 and 4 kernels each of width 5. See Exercise 8.7 for a programming challenge involving applying this 1D CNN for time series to the HFT dataset described in the previous section on RNNs.

For completeness, `ML_in_Finance-2D-CNNs.ipynb` demonstrates the application of a 2D CNN to image data from the MNIST dataset. Such an architecture might be appropriate for learning volatility surfaces but is not demonstrated here.

6 Autoencoders

An autoencoder is a *self-supervised* deep learner which trains the architecture to approximate the identity function, $Y = F(Y)$, via a bottleneck structure. This means we fit a model $\hat{Y} = F_{W,b}(Y)$ which aims to very efficiently concentrate the information required to recreate Y. Put differently, an autoencoder is a form of compression that creates a much more cost-effective representation of Y.

Its output layer has the same number of nodes as the input layer, and the cost function is some measure of the reconstruction error, $Y - \hat{Y}$. Autoencoders are often used for the purpose of dimensionality reduction and noise reduction. A simple autoencoder that implements dimensionality reduction is a feedforward autoencoder with at least one layer that has a smaller number of nodes, which functions as a bottleneck. After training the neural network using back-propagation, it is separated into two parts: the layers up to the bottleneck are used as an encoder, and the remaining layers are used as a decoder. In the simplest case, there is only one hidden layer (the bottleneck), and the layers in the network are fully connected. The compression capacity of autoencoders motivates their application in finance as a non-parametric, non-linear, analogue of the heavily used principal component analysis (PCA). It has been well known since the pioneering work of Baldi and Hornik (1989) that autoencoders are closely related to PCA. We follow Plaut (2018) and begin with a brief review of PCA and then show how exactly linear autoencoders enable PCA.

Example 8.3 A Simple Autoencoder

For example, under a L_2-loss function, we wish to solve

$$\underset{W,B}{\text{minimize}} \, ||F_{W,b}(X) - Y||_F^2$$

subject to a regularization penalty on the weights and offsets. An autoencoder with two layers can be written as a feedforward network:

$$\begin{aligned} Z^{(1)} &= f^{(1)}(W^{(1)}Y + b^{1)}), \\ \hat{Y} &= f^{(2)}(W^{(2)}Z^{(1)} + b^{(2)}), \end{aligned}$$

where $Z^{(1)}$ is a low-dimensional representation of Y. We find the weights and biases so that the number of rows of $W^{(1)}$ equals the number of columns of $W^{(2)}$ and the number of rows is much smaller than the columns, which looks like the architecture in Fig. 8.7.

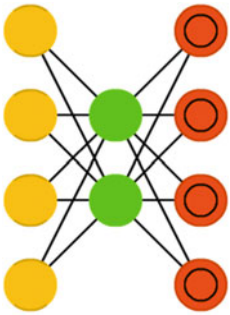


Fig. 8.7 Autoencoders. Source: Van Veen, F. & Leijnen, S. (2019), "The Neural Network Zoo", Retrieved from https://www.asimovinstitute.org/neural-network-zoo

6.1 Linear Autoencoders

In the case that no non-linear activation function is used, $\mathbf{x}_i = W^{(1)}\mathbf{y}_i + b^{(1)}$ and $\hat{\mathbf{y}}_i = W^{(2)}\mathbf{x}_i + \mathbf{b}^{(2)}$. If the cost function is the total squared difference between output and input, then training the autoencoder on the input data matrix Y solves

$$\min_{W^{(1)},\mathbf{b}^{(1)},W^{(2)},\mathbf{b}^{(2)}} \left\| \mathrm{Y} - \left(\mathrm{W}^{(2)} \left(W^{(1)}\mathrm{Y} + \mathbf{b}^{(1)}\mathbb{1}_N^T \right) + \mathbf{b}^{(2)}\mathbb{1}_N^T \right) \right\|_F^2. \tag{8.85}$$

If we set the partial derivative with respect to $\mathbf{b}_2$ to zero and insert the solution into (8.85), then the problem becomes

$$\min_{W^{(1)},W^{(2)}} \left\| \mathrm{Y}_0 - W^{(2)}W^{(1)}\mathrm{Y}_0 \right\|_F^2$$

Thus, for any $\mathbf{b}_1$, the optimal $\mathbf{b}_2$ is such that the problem becomes independent of $\mathbf{b}_1$ and of $\bar{\mathbf{y}}$. Therefore, we may focus only on the weights $W^{(1)}$, $W^{(2)}$.

Linear autoencoders give orthogonal projections, even though the columns of the weight matrices are not orthogonal. To see this, set the gradients to zero, $W^{(1)}$ is the left Moore–Penrose pseudoinverse of $W^{(2)}$ (and $W^{(2)}$ is the right pseudoinverse of $W^{(1)}$):

$$W^{(1)} = (W^{(2)})^{\dagger} = \left(W^{(2)T} W^{(2)} \right)^{-1} (W^{(2)})^T$$

The minimization with respect to a single matrix is

$$\min_{W^{(2)} \in \mathbb{R}^{n \times m}} \left\| \mathrm{Y}_0 - W^{(2)}(W^{(2)})^{\dagger}\mathrm{Y}_0 \right\|_F^2 \tag{8.86}$$

The matrix $W^{(2)}(W^{(2)})^{\dagger} = W^{(2)} \left((W^{(2)})^T W^{(2)} \right)^{-1} (W^{(2)})^T$ is the orthogonal projection operator onto the column space of $W^{(2)}$ when its columns are not

necessarily orthonormal. This problem is very similar to (6.52), but *without* the orthonormality constraint.

It can be shown that $W^{(2)}$ is a minimizer of Eq. 8.86 if and only if its column space is spanned by the first m loading vectors of Y.

The linear autoencoder is said to apply PCA to the input data in the sense that its output is a projection of the data onto the low-dimensional principal subspace. However, unlike actual PCA, the coordinates of the output of the bottleneck are *correlated* and are *not sorted in descending order of variance*. The solutions for reduction to different dimensions are not nested: when reducing the data from dimension n to dimension m_1, the first m_2 vectors ($m_2 < m_1$) are not an optimal solution to reduction from dimension n to m_2, which therefore requires training an entirely new autoencoder.

6.2 Equivalence of Linear Autoencoders and PCA

Theorem *The first m loading vectors of Y are the first m left singular vectors of the matrix $W^{(2)}$ which minimizes (8.86).* □

A sketch of the proof now follows. We train the linear autoencoder on the original dataset Y and then compute the first m left singular vectors of $W^{(2)} \in \mathbb{R}^{n\times m}$, where typically $m << N$. The loading vectors may also be recovered from the weights of the hidden layer, $W^{(1)}$, by a singular value decomposition, If $W^{(2)} = \mathrm{U}\Sigma\mathrm{V}^T$, which we assume is full-rank, then

$$W^{(1)} = (W^{(2)})^\dagger = \mathrm{V}\Sigma^\dagger\mathrm{U}^T$$

and

$$\mathrm{W}_2\mathrm{W}_2^\dagger = \mathrm{U}\Sigma\mathrm{V}^T\mathrm{V}\Sigma^\dagger\mathrm{U}^T = \mathrm{U}\Sigma\Sigma^\dagger\mathrm{U}^T = \mathrm{U}_m\mathrm{U}_m^T, \tag{8.87}$$

where we used the fact that $\left(\mathrm{V}^T\right)^\dagger = \mathrm{V}$ and that $\Sigma^\dagger \in \mathbb{R}^{m\times n}$ is a matrix whose diagonal elements are $\frac{1}{\sigma_j}$ (assuming $\sigma_j \neq 0$, and 0 otherwise).

The matrix $\Sigma\Sigma^\dagger$ is a diagonal matrix whose first m diagonal elements are equal to one and the other $n - m$ elements are equal to zero. The matrix $\mathrm{U}_m \in \mathbb{R}^{n\times m}$ is a matrix whose columns are the first m left singular vectors of W_2. Thus, the first m left singular vectors of $(W^{(1)})^T \in \mathbb{R}^{n\times m}$ are also equal to the first m loading vectors of Y.

A common application of PCA is in fixed income modeling—the principal components are used to characterize the daily movement of the yield curve. Because the components explain most of the variability in the curve, investors can hedge their exposures with only a few instruments from different sectors (Litterman and

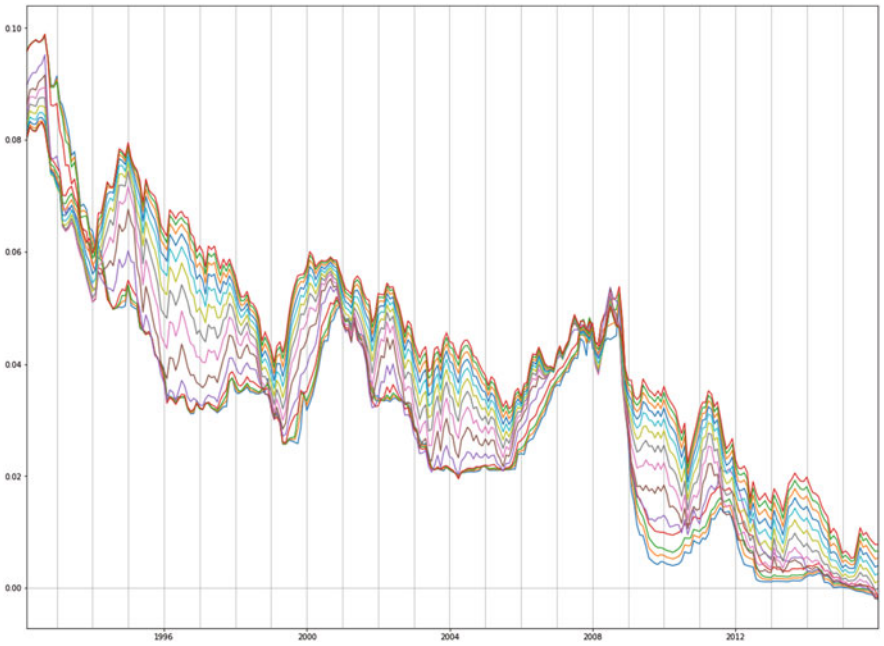
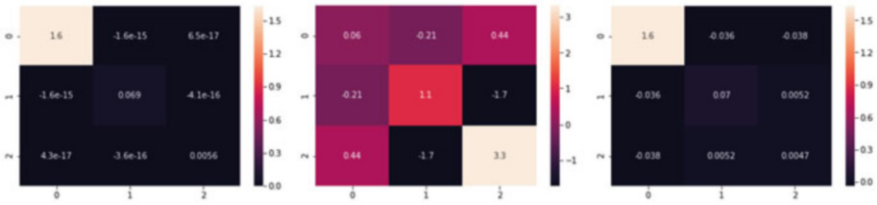


Fig. 8.8 This figure shows the yield curve over time, each line corresponds to a different maturity in the term structure of interest rates

Scheinkman 1991). Figure 8.8 shows the yield curve over a 25-year period, where each line corresponds to each of the maturities in the term structure of fixed income securities.

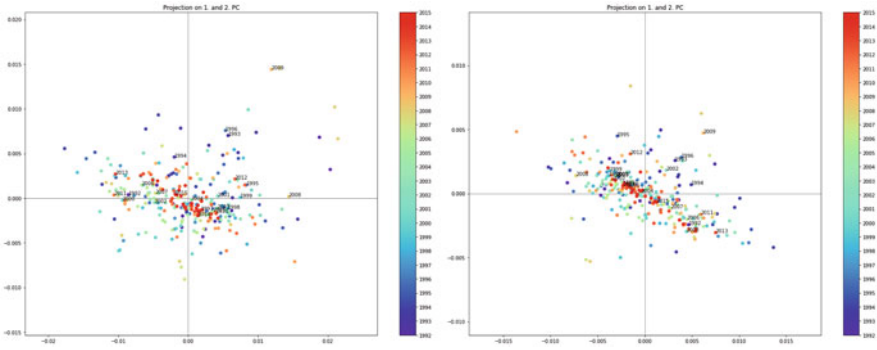
We can illustrate the comparison by finding the principal components of the sample covariance matrix from the time series of the yield curve, as shown in Fig. 8.9a. The eigenvalues are the diagonal of the transformed matrix, are all positive, and arranged in descending order. In this case we have plotted the first $m = 3$ components from a high-dimensional dataset where $n > m$. The percentage of variance attributed to these components is 95.6%, 4.07%, 0.34%, respectively. Figure 8.9c shows the decomposition of the sample covariance matrix using the left singular vectors of the autoencoder weights and observed to be similar to Fig. 8.9a. The percentage of variance attributed to the components is 95.63%, 4.10%, 0.27%. For completeness, Fig. 8.9b shows the transformation using $W^{(2)}$, which results in correlated values.

Performing PCA on the daily change of the yield curve leads to more interpretable components: the first eigenvalue can be attributed to parallel shift of the curve, the second to twist, and the third to curvature (a.k.a. butterfly). Figure 8.10 compares the first two principal components of ΔY_0 using either the $m = 3$ loading vectors, P_m or the $m = 3$ singular vectors, U_m. For the purposes of interpreting the behavior of the yield curve over time, both give similar results. Periods in which



(a) (b) (c)

Fig. 8.9 The covariance matrix of the data in the transformed coordinates, according to (**a**) the loading vectors computed by applying SVD to the entire dataset, (**b**) the weights of the linear autoencoder, and (**c**) the left singular vectors of the autoencoder weights. (**a**) $\mathrm{P}_m^T \mathrm{Y}_0 \mathrm{Y}_0^T \mathrm{P}_m$, (**b**) $(W^{(2)})^T \mathrm{Y}_0 \mathrm{Y}_0^T W^{(2)}$, (**c**) $\mathrm{U}_m^T \mathrm{Y}_0 \mathrm{Y}_0^T \mathrm{U}_m$



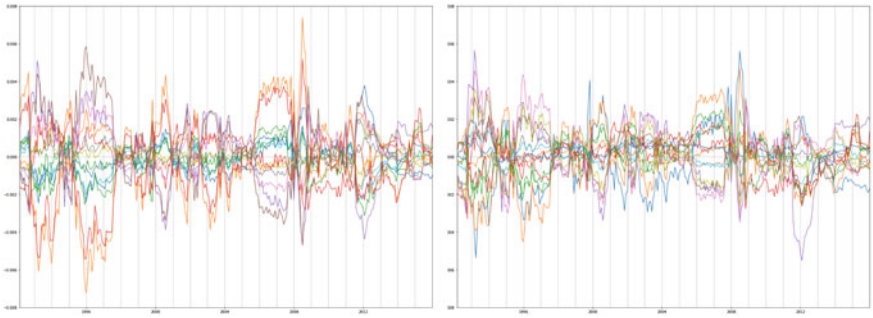
(a) (b)

Fig. 8.10 The first two principal components of ΔY_0, projected using P_m are shown in (**a**). The first two approximated principal components (up to a sign change) using U_m. The first principal component is represented by the x-axis and the second by the y-axis. (**a**) $\mathrm{P}_m^T \Delta Y_0$. (**b**) $\mathrm{U}_m^T \Delta Y_0$

the yield curve is dominated by parallel shift exhibit a large absolute first principal component compared to the second component. And conversely, periods exhibiting a large second component compared to the first indicates that the curve movement is dominated by twist. The latter phenomenal often occurs when the curve moves from an upward sloping to a download sloping regime. In both cases, we note that the period following the financial crisis, 2009, exhibits a relatively large amount of shift and twist when compared to other years.

6.3 Deep Autoencoders

As we saw in the previous chapter, merely adding more layers to the linear autoencoder does not change the properties of the autoencoder—it remains a linear



(a) (b)

Fig. 8.11 The reconstruction error in Y is shown for (**a**) the linear encoder and (**b**) a deep autoencoder, with two $tanh$ activated layers for each of the encoder and decoders

autoencoder and if there are L layers in the encoder, then the first m singular values of $W^{(1)}W^{(2)}\dots W^{(L)}$ will correspond to the loading vectors. With non-linear activation, the autoencoder can no longer resolve the loading vectors. However, the addition of a more expressive, non-linear, model is used to reduce the reconstruction error for a given compression dimension m. Figure 8.11 compares the reconstruction error in Y using the linear autoencoder and a deep autoencoder, with two $tanh$ activated layers in each of the encoder and decoder.

Recently, the application of deep autoencoders to statistical equity factor models has been demonstrated by Heaton et al. (2017). The authors compress the asset returns in a portfolio to give a small set of deep factors which explain the variability in the portfolio returns more reliably than PCA or fundamental equity factors. Such a representation provides a general portfolio selection process which relies on encoding of the asset return histories into deep factors and then decoding, to predict the asset returns. One practical challenge with this approach, and indeed all statistical factor models, is their lack of investability and hedgeability. For ReLU activated autoencoders, deep factors can be interpreted as compositions of financial put and call options on linear combinations of the assets represented. As such, the authors speculate that deep factors could be potentially investable and hence hedgeable.

See the notebook `ML_in_Finance-Autoencoders.ipynb` for an implementation of the methodology and results presented in this section.

7 Summary

In this chapter we have seen how different neural network architectures can be used to exploit the structure in the data, resulting in fewer weights, and broadening their application as a wider class of models than regression and classification.

- We characterize RNNs as non-linear autoregressive models with geometrically decaying lagged coefficients. RNNs can be shown to exhibit unconditional stability under certain constraints on the activation function. In particular, a tanh activation will lead to a stable architecture;
- We combine hypothesis tests from classical time series analysis to guide the experimental design and diagnose the RNN output. In particular, we can sample the partial autocorrelation to determine the sequence length if the data is stationarity and we can check for autocorrelation in the model error to determine if the model has under-fitted.
- Gated recurrent units and long short-term memory architectures give a dynamic autoregressive model with variable memory. Adaptive exponential smoothing is used to propagate a hidden variable with potentially infinite memory. These architectures have the ability to behave as plain RNNs or even as feedforward architectures, i.e. they behave as static non-linear autoregressive models or linear regression as a special case.
- CNNs filter the data and then exploit data locality, either spatial, temporal, or even spatio-temporal, to efficiently represent the input data. When applied to time series, CNNs are non-linear autoregressive models which can be designed to capture multiple scales in the data using dilated convolution.
- Principal component analysis is one of the most powerful techniques for dimension reduction and uses orthogonal projection to decorrelate the features.
- The first m singular values of the weight matrix in a linear autoencoder are the m loading vectors used as an orthogonal basis for projection.
- We can combine these different architectures together to build powerful regressions and compression methods. For example, we might use a GRU-autoencoder to compress non-stationary time series where as we might use a CNN autoencoder to compress spatial data.

8 Exercises

Exercise 8.1*
Calculate the half-life of the following univariate RNN

$$\begin{aligned}\hat{x}_t &= W_y z_{t-1} + b_y,\\ z_{t-1} &= \tanh(W_z z_{t-2} + W_x x_{t-1}),\end{aligned}$$

where $W_y = 1$, $W_z = W_x = 0.5$, $b_h = 0.1$ and $b_y = 0$.

Question 8.2: Recurrent Neural Networks
- State the assumptions needed to apply plain recurrent neural networks to time series data.
- Show that a linear RNN(p) model with bias terms in both the output layer and the hidden layer can be written in the form

$$\hat{y}_t = \mu + \sum_{i=1}^{p} \phi_i y_{t-i}$$

and state the form of the coefficients $\{\phi_i\}$.
– State the conditions on the activation function and weights in a plain RNN under which the model is stable? (i.e., lags do not grow)

Exercise 8.3*
Using Jensen's inequality, calculate the lower bound on the partial autocovariance function of the following zero-mean RNN(1) process:

$$y_t = \sigma(\phi y_{t-1}) + u_t,$$

for some monotonically increasing, positive and convex activation function, $\sigma(x)$ and positive constant ϕ. Note that Jensen's inequality states that $\mathbb{E}[g(X)] \geq g(\mathbb{E}[X])$ for any convex function g of a random variable X.

Exercise 8.4*
Show that the discrete convolution of the input sequence $X = \{3, 1, 2\}$ and the filter $F = \{3, 2, 1\}$ given by $Y = X * F$ where

$$y_i = X * F_i = \sum_{j=-\infty}^{\infty} x_j F_{i-j}$$

is $Y = \{9, 9, 11, 5, 2\}$.

Exercise 8.5*
Show that the discrete convolution $\hat{x}_t = F * x_t$ defines a univariate $AR(p)$ if a p-width filter is defined as $F_j := \phi^j$ for some constant parameter ϕ.

*8.1 Programming Related Questions**

Exercise 8.6***
Modify the RNN notebook to predict Coindesk prices using a univariate RNN applied to the data `coindesk.csv`. Then complete the following tasks

a. Determine whether the data is stationary by applying the augmented Dickey–Fuller test.
b. Estimate the partial autocorrelation and determine the optimum lag at the 99% confidence level. Note that you will not be able to draw conclusions if your data is not stationary. Choose the sequence length to be equal to this optimum lag.
c. Evaluate the MSE in-sample and out-of-sample as you vary the number of hidden neurons. What do you conclude about the level of over-fitting?

d. Apply L_1 regularization to reduce the variance.
e. How does the out-of-sample performance of a plain RNN compare with that of a GRU?
f. Determine whether the model error is white noise or is auto-correlated by applying the Ljung–Box test.

Exercise 8.7***
Modify the CNN 1D time series notebook to predict high-frequency mid-prices with a single hidden layer CNN, using the data `HFT.csv`. Then complete the following tasks

a. Confirm that the data is stationary by applying the augmented Dickey–Fuller test.
b. Estimate the partial autocorrelation and determine the optimum lag at the 99% confidence level.
c. Evaluate the MSE in-sample and out-of-sample using 4 filters. What do you conclude about the level of over-fitting as you vary the number of filters?
d. Apply L_1 regularization to reduce the variance.
e. Determine whether the model error is white noise or is auto-correlated by applying the Ljung–Box test.

Hint: You should also review the HFT RNN notebook before you begin this exercise.

Appendix

Answers to Multiple choice questions

Question 1
Answer: 1,2,4,5. An augmented Dickey–Fuller test can be applied to time series to determine whether they are covariance stationary.

The estimated partial autocorrelation of a covariance stationary time series can be used to identify the design sequence length in a plain RNN because the network has a fixed partial autocorrelation matrix.

Plain recurrent neural networks are not guaranteed to be stable—the stability constraint restricts the choice of activation in the hidden state update.

Once the model is fitted, the Ljung–Box test is used to test whether the residual error is auto-correlated. A well-specified model should exhibit white noise error both in and out-of-sample.

The half-life of a lag-1 unit impulse is the number of lags before the impulse has half its effect on the model output.

Question 2
Answer: 1,4. A gated recurrent unit uses dynamic exponential smoothing to propagate a hidden state with infinite memory. However, there is no requirement for

covariance stationarity of the data in order to fit a GRU, or LSTM. This is because the later are dynamic models with a time-dependent partial autocorrelation structure.

Gated recurrent units are conditionally stable—the choice of activation in the hidden state update is especially important. For example, a tanh function for the hidden state update satisfies the stability constraint. A GRU only has one memory, the hidden state, whereas a LSTM indeed has an additional, cellular, memory.

Question 3

Answer: 1,2,3.

CNNs apply a collection of different, but equal width, filters to the data. Each filter is a unit in the CNN hidden layer and is activated before using a feedforward network for regression or classification. CNNs are sparse networks, exploiting locality of the data, to reduce the number of weights. CNNs are especially relevant for spatial, temporal, or even spatio-temporal datasets (e.g., implied volatility surfaces). A dilated CNN, such as the WaveNet architecture, is appropriate for multi-scale time series analysis—it captures a hierarchy of patterns at different resolutions (i.e., dependencies on past lags at different frequencies, e.g., days, weeks, months). The number of layers in a CNN must be determined manually during training.

Python Notebooks

The notebooks provided in the accompanying source code repository implement many of the techniques presented in this chapter including RNNs, GRUs, LSTMs, CNNs, and autoencoders. Example datasets include 1-minute snapshots of Coinbase prices and a HFT dataset. Further details of the notebooks are included in the `README.md` file.

References

Baldi, P., & Hornik, K. (1989, January). Neural networks and principal component analysis: Learning from examples without local minima. *Neural Netw., 2*(1), 53–58.

Borovykh, A., Bohte, S., & Oosterlee, C. W. (2017, Mar). Conditional time series forecasting with convolutional neural networks. *arXiv e-prints*, arXiv:1703.04691.

Elman, J. L. (1991, Sep). Distributed representations, simple recurrent networks, and grammatical structure. *Machine Learning, 7*(2), 195–225.

Gers, F. A., Eck, D., & Schmidhuber, J. (2001). *Applying LSTM to time series predictable through time-window approaches* (pp. 669–676). Berlin, Heidelberg: Springer Berlin Heidelberg.

Graves, A. (2012). *Supervised sequence labelling with recurrent neural networks*. Studies in Computational intelligence. Heidelberg, New York: Springer.

Heaton, J. B., Polson, N. G., & Witte, J. H. (2017). Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry, 33*(1), 3–12.

Hochreiter, S., & Schmidhuber, J. (1997, November). Long short-term memory. *Neural Comput., 9*(8), 1735–1780.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

Litterman, R. B., & Scheinkman, J. (1991). Common factors affecting bond returns. *The Journal of Fixed Income, 1*(1), 54–61.

Plaut, E. (2018, Apr). From principal subspaces to principal components with linear autoencoders. *arXiv e-prints*, arXiv:1804.10253.

van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., et al. (2016). WaveNet: A generative model for raw audio. *CoRR, abs/1609.03499*.

Zheng, J., Xu, C., Zhang, Z., & Li, X. (2017, March). Electric load forecasting in smart grids using long-short-term-memory based recurrent neural network. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)* (pp. 1–6).

Part III
Sequential Data with Decision-Making

Chapter 9
Introduction to Reinforcement Learning

This chapter introduces Markov Decision Processes and the classical methods of dynamic programming, before building familiarity with the ideas of reinforcement learning and other approximate methods for solving MDPs. After describing Bellman optimality and iterative value and policy updates before moving to Q-learning, the chapter quickly advances towards a more engineering style exposition of the topic, covering key computational concepts such as greediness, batch learning, and Q-learning. Through a number of mini-case studies, the chapter provides insight into how RL is applied to optimization problems in asset management and trading.

1 Introduction

In the previous chapters, we dealt with supervised and unsupervised learning. Recall that supervised learning involves training an agent to produce an output given an input, where a teacher provides some training examples of input–output pairs. The task of the agent is to generalize from these examples, that is to find a function that produces outputs given inputs which are consistent with examples provided by the teacher. In unsupervised learning, the task is again to generalize, i.e. provide some outputs given inputs; however, there is no teacher to provide examples of a "ground truth."

In this chapter, we address a different type of learning where an agent should learn to *act* optimally, given a certain goal, in a setting of sequential decision-making given a state of its environment. The latter serves as an input, while the agent's actions are outputs. Acting optimally given a goal is mathematically formulated as a problem of maximization of a certain objective function. Problems of such sort belong to the area of machine learning known as of reinforcement learning (RL). Such an area of machine learning is tremendously important in trading and investment management.

M. F. Dixon et al., *Machine Learning in Finance*,
https://doi.org/10.1007/978-3-030-41068-1_9

Chapter Objectives

By the end of this chapter, the reader should expect to accomplish the following:

- Gain familiarity with Markov Decision Processes;
- Understand the Bellman equation and classical methods of dynamic programming;
- Gain familiarity with the ideas of reinforcement learning and other approximate methods of solving MDPs;
- Understand the difference between off-policy and on-policy learning algorithms; and
- Gain insight into how RL is applied to optimization problems in asset management and trading.

The notebooks provided in the accompanying source code repository accompany many of the examples in this chapter. See Appendix "Python Notebooks" for further details.

The task of finding an optimal mapping of inputs into outputs given an objective function looks superfluously similar to tasks of both supervised and unsupervised learning. Indeed, in all these cases, and in a sense in all problems of machine learning in general, the objective is always formulated as a sample-based problem of mapping some inputs into some outputs given some criteria for optimality of such mapping. This can be generally viewed as a special case of optimization or alternatively as a special case of function approximation. There are however at least three distinct differences between the problem setting in RL and the settings of both supervised learning and unsupervised learning.

The first difference is the presence and role of a teacher. In RL, like supervised learning and unlike unsupervised learning, there is a teacher. However, a feedback provided by the teacher to an agent is different from a feedback from a teacher in supervised learning. In the latter case, a teacher gives correct outputs for a given training dataset. The role of a supervised learning algorithm is to generalize from such explicit examples, i.e. to provide a function that maps any inputs to outputs, including inputs not encountered in the training set.

In reinforcement learning , a teacher provides only a *partial feedback* for actions taken by an agent. Such a partial feedback is given in terms of *rewards* that the agent receives upon taking a certain action. Rewards have numerical values, therefore a higher reward for a particular action generally implies that this particular action taken by the agent is better than other actions that would produce lower rewards. However, there is no *explicit* information from a teacher on what action is the best or is "right" to produce the highest reward possible. Therefore, a teacher in this case provides only a partial feedback to an agent during training. The goal of a RL agent is to maximize a total cumulative reward over a sequence of steps.