

HAMLET JESSE MEDINA RUIZ
ERNEST CHAN



GENERATIVE AI

FOR TRADING
AND ASSET MANAGEMENT

WILEY

Table of Contents

[Cover](#)

[Table of Contents](#)

[Title Page](#)

[Copyright](#)

[Dedication](#)

[Preface](#)

[Acknowledgments](#)

[About the Authors](#)

[Part I: Generative AI for Trading and Asset Management: A No-code Introduction](#)

[Chapter 1: No-code Generative AI for Basic Quantitative Finance](#)

[1.1 Retrieving Historical Market Data](#)

[1.2 Computing Sharpe Ratio](#)

[1.3 Data Formatting and Analysis](#)

[1.4 Translating Matlab Codes to Python Codes](#)

[1.5 Conclusion](#)

[Chapter 2: No-code Generative AI for Trading Strategies Development](#)

[2.1 Creating Codes from a Strategy Specification](#)

[2.2 Summarizing a Trading Strategy Paper and Creating Backtest Codes from It](#)

[2.3 Searching for a Portfolio Optimization Algorithm Based on Machine Learning](#)

[2.4 Explore Options Term Structure Arbitrage Strategies](#)

2.5 Conclusion

2.6 Exercises

Appendix 2A.1 Computing Next-day's Return

Appendix 2A.2 Uploading the Fama-French Factors

Appendix 2A.3 Combining Fama-French Factors with Next-day's Returns

Chapter 3: Whirlwind Tour of ML in Asset Management

3.1 Unsupervised Learning

3.2 Supervised Learning

3.3 Deep Reinforcement Learning

3.4 Data Engineering

3.5 Feature Engineering

3.6 Conclusion

Part II: Deep Generative Models for Trading and Asset Management

Chapter 4: Understanding Generative AI

4.1 Why Generative Models

4.2 Difference with Discriminative Models

4.3 How Can We Use Them?

4.4 Illustrating Generative Models with ChatGPT

4.5 Hybrid Modeling: Combining Generative and Discriminative Models

4.6 Taxonomy of Generative Models

4.7 Conclusion

Chapter 5: Deep Autoregressive Models for Sequence Modeling

5.1 Representation Complexity

[5.2 Representation and Complexity Reduction](#)

[5.3 A Short Tour of Key Model Families](#)

[5.4 Model Fitting](#)

[5.5 Conclusions](#)

[Chapter 6: Deep Latent Variable Models](#)

[6.1 Introduction](#)

[6.2 Latent Variable Models](#)

[6.3 Examples of Traditional Latent Variable Models](#)

[6.4 Learning](#)

[6.5 Variational Autoencoder \(VAE\)](#)

[6.6 VAEs for Sequential Data and Time Series](#)

[6.7 Conclusion](#)

[Chapter 7: Flow Models](#)

[7.1 Introduction](#)

[7.2 Model Training](#)

[7.3 Linear Flows](#)

[7.4 Designing Nonlinear Flows](#)

[7.5 Coupling Flows](#)

[7.6 Autoregressive Flows](#)

[7.7 Continuous Normalizing Flows](#)

[7.8 Modeling Financial Time Series with Flow Models](#)

[7.9 Conclusion](#)

[Chapter 8: Generative Adversarial Networks](#)

[8.1 Introduction](#)

[8.2 Training](#)

[8.3 Some Theoretical Insight in GANs](#)

8.4 Why Is GAN Training Hard? Improving GAN Training Techniques

8.5 Wasserstein GAN (WGAN)

8.6 Extending GANs for Time Series

8.7 Conclusion

Chapter 9: Leveraging LLMs for Sentiment Analysis in Trading

9.1 Sentiment Analysis in Fed Press Conference Speeches Using Large Language Models

9.2 Data: Video + Market Prices

9.3 Speech-to-text Conversion

9.4 Sentiment Analysis

9.5 Experiment Results

9.6 Conclusion

Chapter 10: Efficient Inference

10.1 Introduction

10.2 Scaling Large Language Models: High Performance, High Computational Cost, and Emergent Abilities

10.3 Making FinBERT Faster

10.4 Model Quantization

10.5 Customizing Your LLM: Adapting Models to Your Needs

10.6 Conclusions

Chapter 11: Afterword

11.1 Diffusion Models

11.2 Combining Generative Model Variants

11.3 LLMs as Financial Advisors

References

Appendix

[A.1 Retrieving Adjusted Closing Prices and Computing Daily Returns](#)

[A.2 Installing Python](#)

[A.3 Plotting the Risk-free-rate over the Years](#)

[A.4 Computing the Sharpe Ratio of SPY](#)

[A.5 Matlab Code for Computing Efficient Frontier and Finding the Tangency Portfolio](#)

[Index](#)

[End User License Agreement](#)

List of Illustrations

Chapter 1

[Figure 1.1 Efficient frontier based on Python code generated by ChatGPT.](#)

Chapter 2

[Figure 2.1 Cumulative returns of Fama-French three-factor strategy.](#)

[Figure 2.2 Incorrect plot of annualized time value of put options as function o...](#)

[Figure 2.3 Incorrect plot of implied volatility of put options as function of t...](#)

[Figure 2.4 Annualized put option prices as function of time to expiration based...](#)

[Figure 2.5 Annualized call option prices as function of time to expiration base...](#)

Chapter 3

[Figure 3.1 Dendrogram of five stocks based on the correlations of their daily r...](#)

[Figure 3.2 Principal components of two correlated series.](#)

[Figure 3.3 Illustration of a sigmoid function.](#)

[Figure 3.4 Illustration of why L1 regularization can more easily get us zero we...](#)

[Figure 3.5 The Anscombe quartet. All four data sets have same means, variances,...](#)

[Figure 3.6 A typical precision-recall curve.](#)

[Figure 3.7 A typical ROC \(Receiver Operating Characteristics\) curve.](#)

[Figure 3.8 A one-hidden-layer MLP that takes a vector of features \$x\$ to classify...](#)

[Figure 3.9 A simple MLP for time-series prediction.](#)

[Figure 3.10 Tying the weights to the same values and connecting the hidden nodes...](#)

[Figure 3.11 Same as Figure 3.10, but rotated 90° counterclockwise so time procee...](#)

[Figure 3.12 Daily minimum and maximum sentiment scores that show structural brea...](#)

Chapter 4

[Figure 4.1 Model taxonomy.](#)

[Figure 4.2 Conditional probability of the first token given the prompt \$y\$: \$p\(x_1|y\)\$.](#)

[Figure 4.3 Conditional probability of the second token given the prompt \$y\$ and p...](#)

[Figure 4.4 Conditional probability of the third token given the prompt \$y\$ and pr...](#)

[Figure 4.5 Conditional probability of the fourth token given the prompt \$y\$ and p...](#)

[Figure 4.6 Conditional probability of the fourth token given the prompt \$y\$ and p...](#)

Chapter 5

[Figure 5.1 Model taxonomy: Autoregressive models.](#)

[Figure 5.2 Sample from the Binarized MNIST dataset. Larochelle and Murray \(2011\).](#)

[Figure 5.3 MADE Generation on MNIST. Left: samples from a MADE model. Right: Ne...](#)

[Figure 5.4 Visualization of a stack of causal convolutional layers. Figure 2 fr...](#)

[Figure 5.5 Visualization of a stack of dilated causal convolutional layers. Fig...](#)

[Figure 5.6 Visualizing attention.](#)

[Figure 5.7 Scaled Dot-Product Attention. Figure 2 \(left\) from Vaswani et al. \(2...](#)

[Figure 5.8 The Transformer encoder-decoder architecture, developed for machine ...](#)

[Figure 5.9 Multi-head Attention. Figure 2 \(right\) from Vaswani et al. \(2023\).](#)

[Figure 5.10 An illustration showing how the current and lagged values of the ser...](#)

[Figure 5.11 An illustration showing how the model input, comprising the time-ser...](#)

Chapter 6

[Figure 6.1 Model taxonomy: variational autoencoders.](#)

[Figure 6.2 Illustration of model selection for homoscedastic noise.](#)

[Figure 6.3 Illustration of model selection for heteroscedastic noise.](#)

[Figure 6.4 Illustration of clustering using Gaussian Mixture Models \(GMMs\). \(a\)...](#)

[Figure 6.5 Gaussian Mixture Model for market regime detection.](#)

[Figure 6.6 VAE.](#)

[Figure 6.7 Illustration of the learned data manifold for generative models with...](#)

[Figure 6.8 Illustration of the encoder-decoder architecture of Base TimeVAE. Th...](#)

[Figure 6.9 Illustration of the main components in Interpretable TimeVAE. Specia...](#)

Chapter 7

[Figure 7.1 Model taxonomy: flow models.](#)

[Figure 7.2 Illustration of the operations involved in coupling flows, including...](#)

[Figure 7.3 Illustration of unbiased samples generated by the NICE model when tra...](#)

[Figure 7.4 Illustration of unbiased samples generated by the NICE model when tra...](#)

[Figure 7.5 Samples generated by the Real-NVP model across four datasets: CIFAR-10...](#)

[Figure 7.6 Illustration of new images generated through interpolations between f...](#)

Chapter 8

[Figure 8.1 Model taxonomy: GANs.](#)

[Figure 8.2 Evolution of GAN-generated images over time, illustrating the progre...](#)

[Figure 8.3 The rightmost column displays the nearest training example to each c...](#)

[Figure 8.4 Table 1 from Goodfellow et al. \(2014\): Parzen window-based log-likel...](#)

[Figure 8.5 Illustration of one reason why training GANs can be difficult. In th...](#)

[Figure 8.6 Illustration of two different cases for the data distribution and th...](#)

[Figure 8.7 Illustration of the optimal discriminator and critic when distinguis...](#)

Chapter 9

[Figure 9.1 Fed Chair Powell discusses latest Fed rate hike.](#)

[Figure 9.2 System block diagram.](#)

[Figure 9.3 SPY Price series during Fed press conference.](#)

[Figure 9.4 Enriched price series.](#)

[Figure 9.5 Scatter plot of sentiment signal vs forward returns.](#)

[Figure 9.6 Available models and languages.](#)

[Figure 9.7 Whisper output on FED data.](#)

[Figure 9.8 Figure 3 from Devlin et al. \(2019\): Illustration of differences in p...](#)

[Figure 9.9 Figure 1 from Devlin et al. \(2019\): Illustration of the input/output...](#)

[Figure 9.10 Figure 2 from Devlin et al. \(2019\): Illustration of the BERT input r...](#)

[Figure 9.11 Time-series sentiment signal and forward returns.](#)

[Figure 9.12 Scatter plot of sentiment signal vs forward returns.](#)

Chapter 10

[Figure 10.1 Figure 1 from Kaplan et al. \(2020\): Illustration of how language mod...](#)

[Figure 10.2 Figure 4 from Wei et al. \(2023\): An illustration of how performance ...](#)

[Figure 10.3 Figure 2 from Wei et al. \(2022\): Illustration of performance, measur...](#)

[Figure 10.4 Softmax distribution.](#)

[Figure 10.5 Weights distribution.](#)

[Figure 10.6 Distribution of quantized weights.](#)

[Figure 10.7 Figure 1 from Hu et al. \(2021\): This figure illustrates the reparamete...](#)

Appendix

[Figure A.1 BIL annualized returns.](#)

[Figure A.2 Twenty-day moving average of annualized BIL returns.](#)

[Figure A.3 Efficient Frontier produced by Matlab codes.](#)

List of Tables

Chapter 3

[Table 3.1 Results of cluster-based features importance ranking based on their ...](#)

[Table 3.2 Hypothetical design matrix for three features with four samples.](#)

[Table 3.3 The confusion matrix for binary classification.](#)

[Table 3.4 Confusion matrix for three-class classification.](#)

Chapter 7

[Table 7.1 Computational complexity of the inverse and determinant of the Jacob...](#)

Chapter 9

[Table 9.1 Performance table.](#)

Chapter 10

[Table 10.1 Performance metrics of teacher vs. student models.](#)

[Table 10.2 Inference speed of teacher vs. student models.](#)

[Table 10.3 Integer range for different bit widths.](#)

[Table 10.4 Performance table for speedups because of quantization.](#)

[Table 10.5 Inference speed after quantization.](#)

Generative AI for Trading and Asset Management

Hamlet Jesse Medina Ruiz
Ernest Chan

WILEY

Copyright © 2025 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial intelligence technologies or similar technologies.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

The manufacturer's authorized representative according to the EU General Product Safety Regulation is Wiley-VCH GmbH, Boschstr. 12, 69469 Weinheim, Germany, e-mail: Product_Safety@wiley.com.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data Applied for:

Print ISBN: 9781394266975

ePDF ISBN: 9781394267002

epub ISBN: 9781394266999

Cover Image: © imaginima/Getty Images

Cover Design: Wiley

Author Photos: Courtesy of the Authors

*To my parents, Denis and Herinarco, and my grandmother,
Gloria María.*

To my family: Ben, Sarah, and Ethan

Preface

There are broadly three types of modern AI models: discriminative models, generative models, and reinforcement learning. Most quantitative asset managers are familiar with discriminative models (e.g., given yesterday's return, what is the probability of today's return being positive); some are also familiar with reinforcement learning (e.g., how can we optimize the selling price to get a better profit). But Generative AI, commonly referred to as GenAI, is a recent invention that receives a lot of buzz but is often mistaken as a synonym with Large Language Models (LLMs) or image generation. But GenAI can learn from anything, not just from text or images. In particular, it can learn from time series of asset returns, which is perhaps most relevant for asset managers.

In this book, we delve into both the applications as well as the fundamentals of GenAI. It is divided into two broad parts: (1) No-code usage of Generative AI for traders and asset managers with *little* coding experience; (2) the fundamentals of Generative AI and their applications in finance for those who are well-versed in coding and discriminative AI. As a result, readers of each category can feel free to just skim the chapters of the other part.

In the first two chapters of [Part 1](#), we will show you examples of how to use the no-code version of GenAI to do stuff that most traders and quantitative investors will encounter in their lifetime. For example, how to retrieve adjusted prices of an ETF from the internet, how to compute the most common performance metrics based on a spreadsheet full of their daily prices; how to convert a trading strategy's backtest code from Matlab to Python (using Matlab code from Ernie's book "Machine Trading"

as example); how to come up with Python code based on a strategy specification; and how to summarize a paper about a trading strategy and turn that into backtest code. What we will *not* be able to show you is “ChatGPT, just come up with a profitable trading strategy that I can use.” At this stage of GenAI development, this level of creativity hasn’t been achieved yet.

The strategies that we asked GenAI to help create include a long-short factor strategy, a VIX futures carry strategy, and a SPX options calendar spread strategy. We also asked GenAI to conduct a literature search for portfolio optimization techniques based on deep reinforcement learning.

For these two chapters, we will use the most commonly known interface for these examples: the web-based ChatGPT GPT-4o which at the time of writing was the most current version of ChatGPT, and its cousin Microsoft's Copilot. Of course, you can probably perform most of these tasks equally admirably using Google's Gemini Pro, [X.AI](#)'s Grok, or DeepSeek, but we haven't tried.

The third chapter of [Part 1](#) of this book is a whirlwind tour of machine learning techniques commonly used in asset management. They range from unsupervised learning to supervised learning and reinforcement learning. The chapter also covers techniques useful for avoiding overfitting and for model selection, such as regularization and hyperparameter optimization. It also covers various nuances in data and feature engineering that are often as important as what machine learning model to choose. It can be used as a primer for finance professionals new to AI, or as a refresher for those who are already dabbling in AI. Until around 2022 when ChatGPT was launched, this is all the AI that most asset managers would ever learn.

[Part 2](#) of the book delves into the fundamentals and technical details of GenAI. [Chapter 4](#) highlights the difference between discriminative and generative AI and introduces the major generative families: deep autoregressive models, and deep latent variable models such as variational autoencoders, flow models, and GANs. Each of these families get their own chapters in the remainder of [Part 2](#). Each chapter explores how these models were originally developed and demonstrates how to adapt them to the dynamics of financial time series, with practical notebook code examples using financial data. The book concludes with end-to-end applications, showing how these models can preprocess alternative data, generate trading signals, and be optimized for efficient inference. [Chapter 9](#) is all about application: how to leverage LLMs for sentiment analysis in trading. [Chapter 10](#) is about deploying these systems in practice—especially how to optimize these models for efficient inference. This chapter is unique, not typically found in generative modeling resources, at least at the time of writing. Efficient inference is something Hamlet worked on the last few years at his company, where scalability and cost-effectiveness were non-negotiable. Once again, Python notebooks that implement most of these techniques are provided throughout. We conclude with [Chapter 11](#), summarizing the main techniques covered in this book. We also emphasize the role of domain expertise in designing meaningful trading strategies, particularly when using LLMs as copilots, and discuss why GenAI, despite its success in other domains, may require more empirical work to achieve similar results in finance. Finally, we outline promising research directions where GenAI could significantly impact finance.

This book is not about proving mathematical theorems. Instead, we try to provide the reader with enough

foundational knowledge to help the reader adapt the techniques to your specific problems. From Hamlet's own experience solving challenging problems at his company, his own projects, and in machine learning competitions, he found that foundational understanding—going back to the mathematical formulation—often holds the key to good solutions. This is especially important when the most popular or widely accepted published solutions tend to fail on real-world datasets! So, this book is his attempt to bring these principles to traders and asset managers. At the same time, he includes many practical code examples that the reader can use to conduct their own research, improve models, and develop new solutions.

Where possible, we have used real-world datasets from the financial domain, limiting the use of standard datasets used for research publications just to explain foundational principles. Throughout the book, we also use numerous examples involving images and text data. These examples not only make the concepts easier to understand but also demonstrate techniques that have been very successful in other fields. They also have direct implications for preprocessing alternative data in finance and integrating it into trading strategies. Additionally, we offer insights into how these techniques can capture the unique characteristics of financial data. Some of the techniques may seem remote from financial applications, but the key to finding alpha where none existed before often comes from borrowing techniques from a different domain.

Part 1 is what Ernie has learned practicing AI and machine learning over the last 30 years, starting at the machine learning group at IBM T. J. Watson Lab, and ending with applying AI in the hedge fund and AI startup he founded (qtscm.com and predictnow.ai). He eats his own dog food.

Part 2 is built on Hamlet's experience at Criteo as its chief data scientist, where he worked on multiple large-scale AI applications, his experience developing his own trading strategies, his participation in achieving top 1% rankings in various machine learning competitions, both finance and non-finance-related, on platforms like Kaggle and Numerai, and his own learning and formal finance education.

Hamlet's journey into finance began, like many others, as an outsider. He started by reading any book he found interesting (fun fact: that included all of Ernie's books). He realized that, like many people entering a new field, it's common to discover that similar methods often exist in your own field but under different names, notations, and applications. Later, as he formalized his journey by completing a MicroMasters in Finance from MIT, he realized there wasn't a single resource covering all the topics he had learned or applied over the years. This book is his attempt to fill that gap.

If Part 2 of this book seems hard, it is because the concepts are truly deep and revolutionary. Read it again, again, and again. After the fourth pass, and after trying out some of the code examples, you too can apply deep learning and GenAI techniques to discover alpha that no one found before.

We are sure that by the time this book is published, we have already gained better understanding and better tools in applying AI to finance, and we will post our latest learnings in our social media accounts (x.com/echanQT, x.com/hamletjmedina, and at our blog substack.com/@gatambook). Also, as Blaise Pascal famously said, "If I had more time, I would have written a shorter letter." We hope to elucidate further some of the essential concepts here via Q&A and short tweets on our social media accounts, and via our workshops.

Acknowledgments

By Hamlet

The journey of writing this book has been deeply rewarding, but also, as my first book, quite challenging. I want to acknowledge the many people who supported me during this journey:

- To Criteo, for allowing me to work on the applied side of the business, tackling a variety of challenging, large-scale applications over many years while also supporting me to spend time on the research side in this fast-evolving field. To my colleagues there: this combination has deeply shaped the way I approach problems and find solutions.
- To my brother, Yanick Medina, currently a master's student in AI, who has been an invaluable contributor to the code in [Part 2](#) of this book. He meticulously reviewed and tested the code while providing feedback that helped us ensure a balance between code modularity and accessibility, a principle we try to keep throughout.
- To my family and friends, whose support and encouragement have been my foundation. Special thanks to my father, who taught me the beauty of mathematics from a very young age.
- To God, whose infinite mercy has made all of this possible.

By Ernest

I would like to thank Dr. Roger Hunter, CTO at QTS Capital Management (qtscm.com), for his partnership in working on many AI in finance projects throughout the years, especially in co-developing the Lifecycle of Trading Strategy Development with Machine Learning workshop (epchan.com/workshops), and the Generative AI for Asset Managers workshop (predictnow.ai/generative-ai-workshop). I am also grateful that the current CEO of QTS Capital Management (qtscm.com), Dr. Nahid Jetha, has continued to advance AI development there to the benefits of our investors.

I also would like to thank my current and former technical team members at Predictnow.ai for their contributions to bringing AI to many asset managers: Johann Abraham, Sergei Belov, Pavan Dutt, Haoyu Fan, Guillaume Goujard, Andrew Inscore, Nancy Khullar, Nancy Xin Man, Uttej Mannava, Akshay Nautiyal, Sudarshan Sawal, Jean Silva, Jai Sukumar, and Quentin Viville.

Of course, this book would not have come into existence without the support of Wiley, especially the invaluable contributions of our managing editor, Stacey Rivera, editorial assistant, Katherine Cording, copyeditor Sheryl Nelson, content specialist Bala Shanmugasundaram, and executive editor, Bill Falloon. Working with them was truly a pleasure.

About the Authors

Hamlet Jesse Medina Ruiz holds the position of chief data scientist at Criteo. He specializes in time series forecasting, machine learning, deep learning, and Generative AI. He actively explores the potential of cutting-edge AI technologies, such as Generative AI across diverse applications.

He holds an electronic engineering degree from Universidad Rafael Belloso Chacín in Venezuela, as well as two master's degrees with honors in mathematics and machine learning from the Institut Polytechnique de Paris and Université Paris-Saclay. Additionally, he earned a PhD in physics from Université Paris-Saclay. Hamlet has consistently achieved first place and top ten rankings in global machine learning contests, earning the titles of Kaggle Expert and Numerai Expert for these challenges. Recently, he also earned a MicroMaster's in finance from MIT's Sloan School of Management.

Ernest Chan (Ernie) is the founder and chief scientific officer of [Predictnow.ai](http://www.predictnow.ai) (www.predictnow.ai), which offers AI-driven adaptive optimization solutions to the finance industry and beyond. He is also the founder and nonexecutive chairperson of QTS Capital Management (www.qtscm.com), a quantitative CTA/CPO since 2011. He started his career as a machine learning researcher at IBM's T.J. Watson Research Center's language modeling group, which produced some of the best-known quant fund managers. Ernie is the acclaimed author of four previous books, *Quantitative Trading* (2nd ed.), *Algorithmic Trading*, *Machine Trading*, and *Hands-on AI Trading* all published by Wiley. More about these books and Ernie's workshops on topics in quantitative investing and machine learning can

be found at www.epchan.com. He obtained his PhD in physics from Cornell University and his BS in physics from the University of Toronto.

Part I

Generative AI for Trading and Asset Management: A No-code Introduction

Chapter 1

No-code Generative AI for Basic Quantitative Finance

In this chapter we want to demonstrate how Gen AI can be used to do the basic tasks for which quantitative traders and investors used to hire a professional programmer. We shall find out to what extent we have succeeded. In all the following examples, we have used a ChatGPT GPT-4o subscription (at US\$ 20/month as of this writing) since it is the most well-known Gen AI service. But you can try Microsoft Copilot (which has live internet access and often gives different answers than ChatGPT because of different “finetuning”), Google’s Gemini Pro (\$19.99/month), xAI’s Grok, or DeepSeek to see if they can do better.

We shall see that while we can enter English instructions into ChatGPT, its best output is often code rather than numerical answers.

Ernie has basic Python programming skills but is much more proficient in Matlab. So, the following is written from the point of view of someone who is a novice Python programmer but an expert Matlab programmer. Let’s see if ChatGPT can help such a person create useful Python code, either from English instructions, or translating from existing Matlab codes.

We will ask ChatGPT to do the following:

- a. Retrieve *adjusted* historical prices of SPY and BIL.
- b. Compute daily returns from those adjusted prices.
- c. Compute the Sharpe ratio from those daily returns.
- d. Analyze the data contained in an Excel spreadsheet.

- e. Translate Matlab codes that use quadratic optimization to compute the efficient frontier of an ETF portfolio to Python code.
- f. Use those Python codes to plot the efficient frontier.

In this and the next chapter, we will use the following boxes to denote prompts to ChatGPT, responses from ChatGPT, and our commentaries.

For user prompts, we will use the following format:

User Prompt

This is a user prompt.

For output from ChatGPT, including code, we will use the following:

ChatGPT Response

A response from ChatGPT

```
# Here is an amazing Python code example.  
import my_genai_trading_system # :)  
% Or a MATLAB code example.  
load_system('my_genai_trading_system') % :)
```

If we embed our own commentary within its response, we will enclose it with [Author Note: ...], as follows:

ChatGPT Response

```
# Here is an amazing Python code example.  
import spicy # [Author Note: This should be  
'scipy', not 'spicy'.]
```

Code, data, and relevant resources produced or required for this chapter can be found in the corresponding chapter folder within the book's repository: <https://github.com/genai-for-traders/genai-for-trading-and-am>. For this chapter, the files are located in the *chapter01* folder and includes scripts, Jupyter notebooks, input data files, and any necessary code for downloading or generating data.

1.1 Retrieving Historical Market Data

As readers of Ernie's first book (Chan, 2021) know, when we backtest a trading strategy on a stock, we must compute its daily returns. To properly compute its daily returns, we must use its closing prices adjusted for splits and dividends.

(Imagine if a stock split 1 to 2 before the market opens on day t . Without adjusting for splits, the return from day $t - 1$ to t would be about -50% or so.) So, we entered one of the most obvious prompts into ChatGPT:

User Prompt

Can you retrieve the daily closing prices of the ETF SPY for the most recent 100 days? Please make sure they are split and dividend adjusted.

ChatGPT Response

It seems that I currently don't have access to the *yfinance* library, which I typically use to retrieve financial data. Therefore, I'm unable to directly fetch the daily prices of the ETF SPY for the most recent 100 days.

However, you can easily obtain this information by using financial data services like Yahoo Finance, Google Finance, or Bloomberg. These platforms provide comprehensive historical data, including split and dividend adjustments. If you would like, I can guide you through the process of retrieving this data using these platforms.

Despite the seemingly unhelpful response as a no-code assistant, it did provide a decent Python script that may accomplish the task, which we included in [Appendix A.1](#). We will discuss that later. Let's say we use that script to download the below prices on our desktop, and upload them as a csv file back into ChatGPT:

```
2023-08-25: $438.42
2023-08-28: $441.20
2023-08-29: $447.58
...
2023-12-11: $461.99
```

We asked this question on 2023-12-12, so the retrieved prices correctly ended on 2023-12-11. They are identical to what you would see on Yahoo Finance if you had done so on the same day. It is difficult to verify that on a later date (such as when you are reading this). Adjusted prices change depending on when you retrieve them, but they should produce the same daily returns no matter how the prices look to you. So, let's now compute the daily returns.

User Prompt

Please compute daily returns based on the above prices.

ChatGPT Response

...

To compute the daily returns, we use the formula:

$$\text{Daily Return} = \left(\frac{\text{Price}_{\text{today}} - \text{Price}_{\text{previous day}}}{\text{Price}_{\text{previous day}}} \right) \times 100\%$$

I'll calculate the daily returns for the SPY ETF based on the adjusted closing prices provided:

$$1. 2023 - 08 - 28 : ((441.20 - 438.42) / 438.42) \times 100\% = 0.63\%$$

$$2. 2023 - 08 - 29 : ((447.58 - 441.20) / 441.20) \times 100\% = 1.45\%$$

... and so on for each subsequent day.

Let's calculate these returns.

Here are the daily returns for the first five days:

$$1. 2023 - 08 - 28 : ((441.20 - 438.42) / 438.42) \times 100\% = 0.63\%$$

$$2. 2023 - 08 - 29 : ((447.58 - 441.20) / 441.20) \times 100\% = 1.45\%$$

...

These percentages represent the daily return for each respective day. Would you like me to continue calculating the daily returns for the remaining days?

Correct though these formula are, the output is difficult to download and verify independently. So we ask:

User Prompt

Can you formulate the above output as a csv file with two columns?

ChatGPT Response

Certainly! Here's the data formatted as a CSV file with two columns, "Date" and "Daily Return":

```
Date,Daily Return  
2023-09-12,-0.0055  
2023-09-13,0.0012  
...  
2023-12-11,0.0039
```

We can copy and paste this output in a plain text file called SPY Daily Returns.csv, and re-open it in Excel. Spot checking a few prices around the dividend ex-date of 2023-09-15, we verify that the returns were computed correctly, based on the adjusted closing prices.

Anticipating that we will need the risk-free rate in order to compute Sharpe ratio and other performance metrics for SPY (or other trading strategies), we ask:

User Prompt

Over the same period of time as above, please retrieve the daily risk free rate in % form.

ChatGPT Response

Retrieving the daily risk-free rate over a specific period, like the one for the SPY ETF adjusted closing prices, can be a bit more complex because there isn't a direct source that provides these rates in a daily format suitable for immediate download and use in financial calculations.

(By the time you are reading this, ChatGPT will probably have integrated different data sources that will enable it to download the yield on the 3-Month U.S. Treasury Yield.)

In other words, it can't do it unless data are available from Yahoo Finance. But there is a proxy that we can use for risk free returns: the SPDR Bloomberg 1-3 Month T-Bill ETF ("BIL"). We will retrieve this by a simple modification of the Python script described in [Appendix A.1](#). We can use the codes in [Appendix A.3](#) to compute the daily returns of BIL, which we can use as the (daily) risk free rate throughout the book.

As any data scientist knows, one of the best ways to check whether the results of a computation are correct is to plot them.

User Prompt

Plot the annualized returns chart with the same x-axis. We only need YYYY as x-axis labels.

You might think that the response to such a simple prompt should be straightforward, but it was not. On the first attempt, ChatGPT incorrectly annualized the returns by multiplying the daily returns by $\sqrt{252}$ instead of 252. At another attempt it couldn't parse the date/time format YYYY-MM-DD hh:mm:ss-05:00 correctly.

User Prompt

What does the date and time format YYYY-MM-DD hh:mm:ss-05:00 mean?

The response to this prompt is correct. The last hh:mm is the timezone offset from UTC. For example, in December, YYYY-MM-DD hh:mm:ss in New York will have –05:00 at the end.

After hours (!) of prompting, it finally produced the correct plot. We reproduced the Python code accompanying the plot in [Appendix A.3](#). We recommend that you check out that code because you will see that the raw data are not quite usable. As a bonus, it gives us a bird's eye view of the US economy since 2007.

1.2 Computing Sharpe Ratio

With strenuous prompting (ChatGPT is like a mediocre and inexperienced intern), we finally downloaded data on our favorite ETFs SPY and BIL, computed the daily returns of both, and can now use these data to compute its Sharpe ratio. As we suggested previously, we want ChatGPT to display the Python code for the computation instead of generating numerical results directly.

User Prompt

Suppose a daily returns series A is stored in A.csv, and the daily risk free rate is stored in B.csv, and the dates of A and B are not identical though they overlap, how do we compute the Sharpe ratio of A? Please display the Python code.

We display the (largely correct) response obtained in [Appendix A.4](#). The Sharpe ratio computed in [Appendix A.4](#) is

for the period 2023-08-25 to 2023-12-11, which was a very bullish period. You can of course adapt the same prompts and their resulting Python codes for other performance metrics like maximum drawdown, Calmar ratio, and the like.

1.3 Data Formatting and Analysis

Not every spreadsheet that comes our way will be nicely formatted like the one downloaded from Yahoo. One of the most thankless and time-consuming task for a data analyst ... eh sorry ... scientist, is to format the input data file correctly so we can analyze it. For example, a colleague sent us the Indices.xlsx file saved in <https://github.com/genai-for-traders/genai-for-trading-and-am/chapter01/>. We have anonymized the names of the indices to "A" "B" "C" but left everything else unchanged. The file contains lots of floating detritus such as the names of these indices, the start and end dates of the data, and #N/A for those dates that did not have data. Let's see if ChatGPT can make the task of analyzing this data easier.

User Prompt

This file contains four columns of pricing data. Can you tell me the column names for these data?

[Author Note: we can attach a file to the prompt in ChatGPT.]

ChatGPT Response

Unfortunately, I encountered an error while trying to access and read the newly uploaded file. This could be due to a variety of reasons such as file format issues or compatibility problems.

That's not promising! When we tried this a couple of months back, it had no problem analyzing the same data. After several hours of trial and error, a trivial solution worked: we just logged out and back in again. This is often frustratingly the case with ChatGPT: the indeterministic and irreproducible nature of the results. Now the response is:

ChatGPT Response

The Excel file you provided contains the following column names:

1. Unnamed: 0
2. Unnamed: 1
3. [A column named with a date: 2000-01-03]
4. Unnamed: 3
5. Unnamed: 4
6. Unnamed: 5
7. Unnamed: 6
8. Unnamed: 7
9. Unnamed: 8
10. Unnamed: 9

...

Clearly, it needs a bit of human help. So

User Prompt

Use row 6, columns G to J as the column names for the pricing data.

ChatGPT Response

```
A Index  
B Index  
C Index  
D Index  
...
```

which is correct.

User Prompt

Compute the daily returns of these four indices.

ChatGPT Response

It appears there was an error in calculating the daily returns, likely due to the presence of non-numeric data in the columns.

The NaNs are tripping ChatGPT up. So

User Prompt

For those cells with NaN, you can output NaN as returns. But otherwise, display their numerical results.

After this prompt, ChatGPT attempted to compute the daily returns for several subsets of the data, and without the dates column. Finally, we prompted

User Prompt

I want the analysis to start from the first row to the last row, and output a .csv file for the results.

It computed the current daily returns for all the data saved in a .csv file. For example,

User Prompt

What are the returns on 11/6/2023?

ChatGPT Response

The daily returns for the indices on November 6, 2023, are as follows:

```
A Index: 0.001378  
B Index: -0.004321  
C Index: -0.003129  
D Index: -0.000333  
...
```

User Prompt

Using these daily returns, please compute the annualized Sharpe ratio of the A Index from 2019/1/7 to 2023/7/3, assuming risk free rate of 0.

ChatGPT Response

The annualized Sharpe ratio of the A Index for the period from January 7, 2019, to July 3, 2023, assuming a risk-free rate of 0, is approximately 0.732.

Once we spot-checked a few numbers like this, it is much more convenient and efficient to use ChatGPT to extract other statistical information on all or parts of the data instead of using Excel or Python. ChatGPT can be a great friend to data analysts ... sorry ... scientists.

1.4 Translating Matlab Codes to Python Codes

Readers of Ernie's previous three books know that they are written in Matlab, for reasons explained in his third book, *Machine Trading* (Chan, 2017). Even though he subsequently added Python codes to the online code repositories for the first two books, there aren't such codes for *Machine Trading*. We will see if we can use ChatGPT to translate such Matlab codes to Python.

The first code example in *Machine Trading* (ef.m) is about computing the efficient frontier of a portfolio of ETFs and finding the tangency portfolio (the one that maximizes Sharpe ratio historically). For our readers' convenience, we have reproduced that in [Appendix A.5](#).

Now let's ask ChatGPT to translate this into Python. You can just upload ef.m and ask for translation, or you can copy and paste the entire file as text into the prompt.

User Prompt

Translate the following Matlab code to Python:

```
% ef.m
% Find efficient frontier of a set of ETFs
% Note in this program mean and sd of returns are
% row vectors,
clear;

load('inputDataOHLCDailyETF_20150417', 'stocks',
'tday', 'cl');

...
```

Author Note: see [A.5](#)

ChatGPT Response

To translate this MATLAB code to Python, we'll use libraries such as NumPy for numerical computations and SciPy for optimization (specifically, quadratic programming). The MATLAB code is used to find the efficient frontier for a set of ETFs, compute the tangency portfolio, and find the minimum variance portfolio. Here's the equivalent Python code:

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize

# Assuming 'inputDataOHLCDailyETF_20150417.csv' is
# a CSV file with columns 'stocks', 'tday', and 'cl'
# Load data (you'll need to replace this with your
# actual data loading mechanism)
data =
pd.read_csv('inputDataOHLCDailyETF_20150417.csv')
stocks = data['stocks'].unique()
cl = data.pivot(index='tday', columns='stocks',
values='cl')

# Remove 'EWZ' and 'FXI'
cl = cl.drop(columns=['EWZ', 'FXI'])

# Calculate returns
R = cl.pct_change().dropna()

# Mean and covariance of returns
mi = R.mean()
C = R.cov()

# Prepare different target mean portfolio returns
# for efficient frontier
m = np.linspace(mi.min(), mi.max(), 21)
v = np.zeros_like(m)

# Variance of portfolio to be minimized
H = 2 * C.values

# Short sale constraint
A = -np.eye(len(mi))
b = np.zeros(len(mi))

# No linear term in minimization
f = np.zeros(len(mi))

# Efficient frontier calculation
for i in range(len(m)):
```

```

Aeq = np.vstack([mi, np.ones(len(mi))])
beq = np.array([m[i], 1])

def objective(F):
    return F @ H @ F

constraints = ({'type': 'eq', 'fun': lambda F:
np.dot(Aeq[0], F) - beq[0]},{'type': 'eq', 'fun': lambda F:
np.dot(Aeq[1], F) - beq[1]})

bounds = [(0, None)] * len(mi)
result = minimize(objective, np.ones(len(mi)) / len(mi), bounds=bounds, constraints=constraints)
v[i] = result.fun

sd = np.sqrt(v)
plt.scatter(sd, m)
plt.xlabel('Standard Deviation')
plt.ylabel('Mean Return')
plt.title('Efficient Frontier')
plt.show()

# Find tangency portfolio
sharpe_ratio = m / sd
idx = np.argmax(sharpe_ratio)

# Find minimum variance portfolio
idx_min = np.argmin(sd)

# For further details like the weights of the
# tangency and minimum variance portfolios,
# use the 'idx' and 'idx_min' with the 'minimize'
# function as shown above.

```

ChatGPT goes on to respond:

ChatGPT Response

This code does the following:

Loads the ETF data and computes daily returns.
Calculates the mean and covariance of the returns. Uses quadratic programming (via `scipy.optimize.minimize`) to find the efficient frontier, tangency portfolio, and minimum variance portfolio.

Note: The `inputDataOHLCDailyETF_20150417.csv` should be replaced with the actual file name and path you have. This code assumes the data is in a long format with “stocks,” “tday” (date), and “cl” (close price) columns. Adjust the data loading and manipulation according to your actual data format. The plotting part (using `matplotlib.pyplot`) is minimal; you might want to enhance it based on your specific visualization requirements.

The first hurdle in comparing the output of the Python codes to the Matlab codes is that the Python code expects a .csv input file in the “long format,” i.e., it assumes that price data corresponding to different stocks to be stacked into rows one day at a time. For example,

```
20050512    EWC 1412  
20050512    EWG 1388  
20050512    EWJ 894  
...  
20150417    EWB 6102  
20150417    EWZ 3453  
20150417    FXI 5003
```

Assuming that we are not expert Matlab programmers, we ask ChatGPT to create Matlab code that can save the information in the input .mat file into a .csv.

User Prompt

In the Matlab code, we assumed that the input file inputDataOHLCDailyETF_20150417.mat has a cell array stocks with length S, a double array tday with length T, and a double array cl with dimension TxS. What Matlab code can we write to output a csv file that conforms to the format assumed in the input .csv file of the Python code?

ChatGPT Response

```
% Load data from .mat file
load('inputDataOHLCDailyETF_20150417.mat',
'stocks', 'tday', 'cl');
% Initialize a table with 'tday' as the first column
outputData = array2table(tday, 'VariableNames',
{'tday'});

% Add columns for each stock, with closing prices as
values
for i = 1:length(stocks)
    outputData.(stocks{i}) = cl(:, i);
end

% Reshape the data to long format
outputDataLong = stack(outputData, stocks,
'NewDataVariableName', 'cl', 'IndexVariableName',
'stocks');

% Write to CSV
writetable(outputDataLong, 'outputData.csv');
```

The output data file looks reasonable. Renaming outputData.csv to “inputDataOHLCDailyETF_20150417.csv” will let us test the Python code.

From this point on, our voyage becomes treacherous. The Python codes ChatGPT translated from Matlab didn't work at first. We will display below the tedious prompt-vs-response between the human and the machine where we coached ChatGPT to produce codes that finally work, not because we are sadistic, but because it illustrates what it takes to squeeze the correct codes out of ChatGPT.

Running the Python code results in error
“ModuleNotFoundError: No module named ‘scipy’.”
Prompting ChatGPT for help generates:

ChatGPT Response

The error “ModuleNotFoundError: No module named ‘scipy’” indicates that the SciPy library, which is essential for the optimization part of the code (specifically, the minimize function used for quadratic programming), is not installed in your Python environment.

To resolve this, you will need to install the SciPy library. If you are using a standard Python installation, you can install SciPy using pip, which is the Python package manager. Here's how you can do it:

Open your command line interface (CLI). Run the following command:

```
pip install scipy
```

which we did. Running the Python code again resulted in error:

```
NameError                                         Traceback
(most recent call last)
Cell In[1], line 50
    47     v[i] = result.fun
    49 sd = np.sqrt(v)
--> 50 plt.scatter(sd, m)
    51 plt.xlabel('Standard Deviation')
    52 plt.ylabel('Mean Return')

NameError: name 'plt' is not defined
```

Confronting ChatGPT with this error yields:

ChatGPT Response

The error you're encountering, "NameError: name 'plt' is not defined," indicates that the matplotlib.pyplot module, commonly imported as plt, has not been imported in your Python script. This module is necessary for plotting the efficient frontier.

To resolve this, you need to import matplotlib.pyplot at the beginning of your script. Here's the corrected segment of the Python code with the necessary import statement:

```
...
import matplotlib.pyplot as plt # Import matplotlib
for plotting

# [Rest of your code]
```

Nice of ChatGPT to tell us that, but why didn't it add that in the beginning? Anyway, the code now generated the [Figure 1.1](#). (If you want to save this to a .png file, remember to add `plt.tight_layout()` before `plt.savefig(plot_file_path)`.)

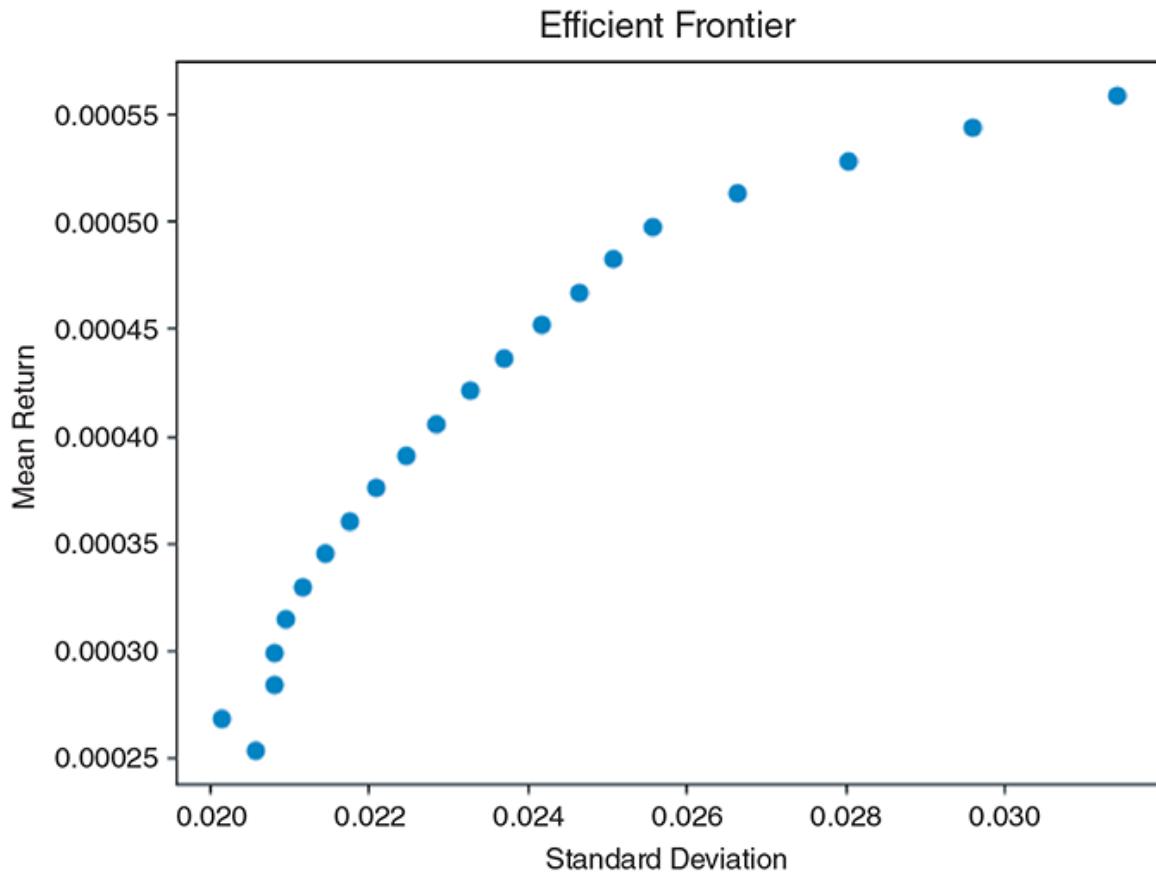


Figure 1.1 Efficient frontier based on Python code generated by ChatGPT.

Comparing this plot with the efficient frontier plotted by the Matlab code in [Appendix A.5](#), we see clearly that there is something wrong with the Python code. The scales of the x and y axes are correct. The coordinates of the minimum variance portfolio (corresponding to the green dot on the Matlab chart) also seem right. But what happened to the portfolios with the lowest two mean returns? Drilling down into the details, the mean returns vectors m_i from both programs match. The covariance matrices C from both programs also match. Following the suggestion from ChatGPT above, let's compute the weights for the tangency portfolio:

```

Aeq = np.vstack([mi, np.ones(len(mi))])
beq = np.array([m[idx], 1])

def objective(F):
    return F @ H @ F

constraints = ({'type': 'eq', 'fun': lambda F:
np.dot(Aeq[0], F) - beq[0]},{'type': 'eq', 'fun': lambda F:
np.dot(Aeq[1], F) - beq[1]})

bounds = [(0, None)] * len(mi)
result = minimize(objective, np.ones(len(mi)) /
len(mi), bounds=bounds, constraints=constraints)
tangency_portfolio_weights = result.x

```

tangency_portfolio_weights are found to be [0.19413052, 0.2978653, 0., 0.01713741, 0., 0.49086677] which are very different from $F = [0.3815, 0, 0.6040, 0.0001, 0.0143, 0]$ found by Matlab. Plugging these weights back into the array for the Sharpe ratios at different weights, we found that Matlab gave a Sharpe ratio of 0.0283 for the tangency portfolio vs Python's 0.0194. Clearly, Matlab's optimization is better. This is, however, not really the fault of ChatGPT, as a line-by-line comparison of the codes revealed nothing erroneous. In fact, we are pleasantly surprised that it was able to find an optimization package ("minimize") in scipy that reasonably mirrors the quadratic optimization function in Matlab. The fault lies squarely with Python's subpar optimization package. As Ernie has maintained in *Machine Trading* (Chan, 2017), Python is a poor cousin of Matlab—you use it at your own peril. While Matlab's codes are developed and maintained commercially by a team of full-time professionals and PhDs, Python's codes are developed and maintained by essentially a group of part-time volunteers. (A bit of grapevine gossip: the Father of Deep Learning and Nobel prize laureate, Dr. Geoff Hinton himself, was known to prefer Matlab to Python in his own research.)

1.5 Conclusion

We have asked ChatGPT to retrieve historical data from the web, compute returns and Sharpe ratio from these returns, conduct exploratory data analysis, apply quadratic optimization to compute the efficient frontier, and in general translate Matlab to Python codes. With an annoyingly large amount of coaching (i.e., prompting) and vigilant error-checking, we conclude that ChatGPT is capable of these tasks. At the very least, it can generate a first draft from which human quants can use to build better codes. This is especially helpful when the human is not an expert programmer. For example, finding the exact syntax to generate the precise chart we want in Python is tedious and time consuming for a native Matlab programmer like Ernie—a task that ChatGPT can do efficiently using English instructions and with minimal room for serious error. The keyword here is “codes”—we found it much better to ask ChatGPT to suggest codes that we can audit (see Alshahwan et al., 2024) and modify than to provide the direct numerical answers to our questions.

In the next chapter, we will be more ambitious and see if Gen AI can help us backtest or even suggest quantitative trading strategies.

Chapter 2

No-code Generative AI for Trading Strategies Development

In the previous chapter, we found that with sufficient patience and prompting, we can get ChatGPT to perform basic financial data science tasks such as retrieving, exploring, and plotting data and computing performance statistics from them. Though the instructions are all in English, the output are frequently codes (especially Python codes). We still consider that no-code usage of Gen AI.

In this chapter, we will see if no-code Gen AI can handle more challenging tasks such as turning a trading strategy specification in English into backtest codes, and even more ambitious, suggest a trading strategy based on a vague human insight. As a reminder, Ernie has basic Python programming skills, is much more proficient in MATLAB, and is a hedge fund founder. This means he has extensive experience and domain knowledge in finance. In the following examples, the author approaches the development of the examples as someone who is an expert in the finance field, but not a professional programmer or a specialist in prompt engineering (like many professional asset managers). He needs help coding in a new language. Let's see if ChatGPT can help him. Spoiler alert: ChatGPT's responses will likely improve in later releases.

As in [Chapter 1](#), we use ChatGPT GPT-4o for our tests. You can also try Microsoft's Copilot, Google's Gemini Pro, xAI's Grok, or DeepSeek to see if they can do better. We will ask ChatGPT to do the following:

- a. Create Python codes to backtest a long-short equity investment strategy that Ernie specified, which uses

Fama-French factors to predict individual stocks' returns.

- b. Create Python codes to backtest a hedged VIX futures carry strategy from a published paper.
- c. Conduct a literature search to find a portfolio optimization technique that uses deep reinforcement learning and that can adapt to regime changes.
- d. Suggest and conduct exploratory analysis of a SPX options calendar spread trading strategy.

Once again, we will use the following boxes to denote prompts to ChatGPT, responses from ChatGPT, and our commentaries.

For user prompts, we will use the following format:

User Prompt

This is a user prompt.

For output from ChatGPT, including code, we will use:

ChatGPT Response

A response from ChatGPT

```
# Here is an amazing Python code example.  
import my_genai_trading_system # :)  
  
% Or a MATLAB code example.  
load_system('my_genai_trading_system') % :)
```

If we embed our own commentary within its response, we will enclose it with [Author Note: ...], for example:

ChatGPT Response

```
# Here is an amazing Python code example.  
import spicy # [Author Note: This should be  
'scipy', not 'spicy'.]
```

Once again, code, data, and relevant resources produced or required for this chapter can be found in the corresponding chapter folder within the book's repository:

<https://github.com/genai-for-traders/genai-for-trading-and-am>. For this chapter, the files are located in the *chapter02* folder and include scripts, Jupyter notebooks, input data files, and any necessary code for downloading or generating data.

2.1 Creating Codes from a Strategy Specification

In *Machine Trading* (Chan, 2017), Ernie created a toy strategy that used the Fama-French three-factor model to predict stocks' returns and backtested the performance of a trading strategy based on those predictions. Let's ask ChatGPT to create all the Python codes necessary for the backtest by just specifying the strategy in English. (Recall that Chan, 2017 only has Matlab codes.)

User Prompt

In the following dialogue, pretend you are a quantitative finance expert.

ChatGPT Response

... [Author Note: The response is a lot of unimportant babble. Some research suggests it results in better responses if we first prime ChatGPT on a particular application area. See Kıcıman et al. (2023).]

User Prompt

Fama-French factors are often used to explain the returns of a portfolio of stocks. What are the Fama-French factors?

ChatGPT Response

The Fama-French factors are a set of three factors (and later expanded to five) introduced by Eugene F. Fama and Kenneth R. French. They are used in finance to explain the returns of a stock portfolio, going beyond the traditional Capital Asset Pricing Model (CAPM), which uses only market risk (beta) to explain returns. Here's a breakdown of these factors:

Original Three Factors

Market Risk ($R_m - R_f$):

This is the same as in CAPM, representing the excess return of a portfolio over the risk-free rate, capturing the market risk. R_m stands for the return of the market portfolio, and R_f is the risk-free rate.

Size Factor ($SMB - Small\ Minus\ Big$):

SMB stands for “Small [market capitalization] Minus Big”. It captures the historical excess returns of small-cap stocks over big-cap stocks. The factor is based on the observation that, over long periods, small-cap stocks tend to outperform large-cap stocks.

Value Factor ($HML - High\ Minus\ Low$):

HML stands for “High [book-to-market ratio] Minus Low”. This factor captures the excess returns of value stocks (with high book-to-market ratios) over growth stocks (with low book-to-market ratios). Value stocks are those that are typically undervalued by the market but have strong fundamentals.

...

So far so good—at least ChatGPT regurgitates the correct information that you can find in any quantitative finance

textbook (e.g., Ruppert and Matteson, 2015) or even Wikipedia. Now let's get it to produce codes.

User Prompt

Let's focus only on the original Fama-French three factors. Instead of using them to explain contemporaneous returns of stocks in a portfolio, let's use them to predict the next day's returns of these stocks. Can you display the mathematical setup for this prediction?

ChatGPT Response

Certainly! To use the original Fama-French three factors for predicting the next day's returns of stocks in a portfolio, we set up a predictive linear regression model. In this model, the future stock returns are the dependent variable, and the current values of the Fama-French factors are the independent variables. The mathematical setup for this predictive model is as follows:

Let's denote: [Author Note: We did some light editing of the LaTeX code to make it more readable.]

$R_{i,t+1}$: The return of stock i in the next day (day $(t + 1)$).

$R_{m,t} - R_{f,t}$: The excess return of the market over the risk free rate in the current day (t).

SMB_t : The small minus big factor in the current day.

HML_t : The high minus low factor in the current day.

$(\beta_{i0}, \beta_{i1}, \beta_{i2}, \beta_{i3})$: Coefficients to be estimated for stock i .

$\epsilon_{i,t+1}$: The error term for stock i in month [Author Note: This should be day, not month] $t + 1$.

The linear regression model for each stock i can be represented as:

$$R_{i,t+1} = \beta_{i0} + \beta_{i1}(R_{m,t} - R_{f,t}) + \beta_{i2}SMB_t + \beta_{i3}HML_t + \epsilon_{i,t+1}$$

In this model:

β_{i0} , is the intercept term for stock i .

$\beta_{i1}, \beta_{i2}, \beta_{i3}$ are the coefficients measuring the sensitivities of stock i 's returns to the market excess return, SMB, and HML factors, respectively.

$\epsilon_{i,t+1}$ captures the residual effects not explained by the model.

This setup allows for the prediction of next day's returns for each stock in the portfolio based on the current values of the Fama-French factors. It's important to note that while this model uses historical relationships to make predictions, the accuracy of these predictions can vary due to market volatility, economic changes, and other factors not captured by the model.

User Prompt

Correct. Can you turn this mathematical model into Python code?

To which ChatGPT responded with Python codes that represent the linear regression of a single stock's returns against the Fama-French factors. Obviously, that isn't complete. Also, as a prerequisite, we will need data. Assume that the prices are contained in the file "CRSP_mid.csv," which contains data formatted as follows:

```
tday      syms   mid
20070103  ABT    8.867780
20070103  HON    37.689365
20070103  AA     25.932974
20070103  BEAM   45.437196
20070103  AEP    31.218061
...
20131231  GGP    20.055000
20131231  MHK    148.904995
20131231  ADS    262.945005
20131231  FB     54.655000
20131231  TSCO   NaN
[1309166 rows x 3 columns]
```

While we cannot share this data due to licensing restrictions, you can construct a similar file from any daily stock

survivorship-bias-free data, such as that from Sharadar. (See a list of other data vendors in Chan [2017].)

* * *

Exercise: Ask ChatGPT to help you convert whatever data you purchased into the “long” format shown in [Section 1.4](#).

* * *

We asked ChatGPT for the code for computing the future 1-day return, and it responded with something that looked like the code in [Appendix 2A.1](#) at the end of this chapter. The original code wasn’t perfect, but if you know a bit of Python, it is easy to fix it up.

Next, we need to upload the Fama-French data file, which is available from

https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/da ta_library.html as “F-F_Research_Data_Factors_daily.CSV.”

We again prompted ChatGPT for the codes, and the response is passable. After we edited it lightly, the code is displayed in [Appendix 2A.2](#). We also prompted for the codes for merging the labels (future 1-day return) and the features (Fama-French factors) into one dataframe for use as our training data. Those are shown in [Appendix 2A.3](#).

Now we are ready to train a regression model, or more precisely, train regression models for all the stocks. We started with the following:

User Prompt

The linear regression model you proposed above is correct, but it is for one stock only. Can you display the codes for all the stocks in the data?

ChatGPT Response

...

```
# Assuming 'Stock_ID' and '1_Day_Returns' are
# columns in your next month
# returns data
# Train a model for each stock
for stock_id in combined_data['Stock_ID'].unique():
    stock_data =
    combined_data[combined_data['Stock_ID'] == stock_id]
    X = stock_data[['Mkt-RF', 'SMB', 'HML']] # Replace with actual Fama-French column names
    y = stock_data['1_Day>Returns']
    model = LinearRegression().fit(X, y)
    # Save or use the model for each stock
```

The code looks correct. But unfortunately, it can't handle any NaNs in the data, and financial data (or indeed any data) is full of such missing values. So, we try again.

User Prompt

Correct, but what if the X and y contains NaN?

ChatGPT Response

as most machine learning algorithms, including linear regression in scikit-learn, cannot handle NaN values directly. To address this, you have a few options: ...

The statement that “most machine learning algorithms cannot handle NaN” is *wrong*. Certainly most if not all algorithms in Matlab’s Statistics and Machine Learning Toolbox can handle NaN. We want to catch ChatGPT in its lie:

User Prompt

Can LightGBM handle NaNs in the input?

ChatGPT Response

Yes, LightGBM, a gradient boosting framework that uses tree-based learning algorithms, can handle NaN (Not a Number) values in its input data. ...

Which directly contradicts ChatGPT's own statement above!
Self-consistency is not a forte of Gen AI.

User Prompt

Is there any Python package for linear regression that can handle NaN in the input data?

ChatGPT Response

... Statsmodels, another statistical modeling library in Python, can handle missing data using the `missing='drop'` parameter in its model-fitting functions. This tells the model to automatically drop observations with NaNs.

```
import statsmodels.api as sm  
model = sm.OLS(y, X, missing='drop').fit()
```

Note that this approach drops any row with NaNs, similar to using `dropna()` in pandas, but it's done internally within the model.

So, we tried replacing the line

```
model = LinearRegression().fit(X, y)
```

with the sm.OLS line (after running pip install statsmodel). Well, that didn't work either, because sm.OLS is interpreting the missing='drop' option as weights. ChatGPT has no idea why, and suggested we remove all rows with NaN before running the LinearRegression in scikit-learn.

```
import pandas as pd
from sklearn.linear_model import LinearRegression

# Assuming 'combined_data' is your DataFrame
some_minimum_threshold=100 # Define a threshold as per
your analysis

for stock_id in combined_data['Stock_ID'].unique():
    stock_data =
combined_data[combined_data['Stock_ID'] == stock_id]

    # Remove rows where any of the columns have NaN
values
    stock_data = stock_data.dropna(subset=
['1_Day_Returns', 'Mkt-RF', 'SMB', 'HML'])

    X = stock_data[['Mkt-RF', 'SMB', 'HML']]
    y = stock_data['1_Day_Returns']

    # Ensure that X and y have enough data points
    if len(X) > some_minimum_threshold: # Define a
threshold as per your analysis
        model = LinearRegression().fit(X, y)

        print({
            'Stock_ID': stock_id,
            'Intercept': model.intercept_,
            'Beta_Market': model.coef_[0],
            'Beta_SMB': model.coef_[1],
            'Beta_HML': model.coef_[2]
        })
```

That seems to work. But we don't want to fit a regression model using all our data—we need to leave some data for out-of-sample testing of our trading strategy. So let's use the first half for training:

User Prompt

The program you created above using Linear Regression to predict 1-day return is correct, but can you modify it so that it uses only the first half of the data for fitting?

ChatGPT Response

...

```
# Split the data into two halves
halfway_point = len(combined_data) // 2
training_data = combined_data.iloc[:halfway_point]
#...
for stock_id in training_data['Stock_ID'].unique():
    # Filter training data for the current stock
    stock_data =
    training_data[training_data['Stock_ID'] == stock_id]
#...
```

The changes look correct and are easy enough to incorporate in our original program. (See the final code fragment in the `#%%Train` model cell in `ff_1day.py`, available for download from our GitHub.)

We will need to make predictions using the previous trained (fitted) model. Let's ask ChatGPT how to do that.

User Prompt

With the fitted regression model above, how do you make predictions using input “features”?

ChatGPT responded with some very cumbersome codes that explicitly multiplies the regression coefficients such as “Beta_Market” with the input features contained in X. So, we asked it to make them more elegant.

User Prompt

Instead of explicitly using the names of the model coefficients to make these predictions, can you call some predict function generic to a regression model object?

ChatGPT Response

Certainly! To use a generic predict function that is common to regression model objects, you would typically save each fitted model object for each stock during the training process, and then use the predict method of these model objects to make predictions. This approach is more flexible and aligns with standard practices in model deployment.

Here's how you can adjust the code to include saving the fitted model objects and then using their predict methods:

Step 1: Save Each Fitted Model

While training the models, save each fitted model object into a dictionary where the keys are the stock IDs:

```
from sklearn.linear_model import LinearRegression

# Dictionary to store the model for each stock
fitted_models = {}

for stock_id in training_data['Stock_ID'].unique():
    # ... [Data preparation steps]

    # Fit the model
    model = LinearRegression().fit(X, y)

    # Store the fitted model
    fitted_models[stock_id] = model
```

Step 2: Predict Using the Saved Models

Use the predict method of each saved model to make predictions on new data:

```
def predict_returns(fitted_models, features):
    """
        Make predictions for each stock using the fitted
        models.

        :param fitted_models: Dictionary of fitted model
        objects for each stock
        :param features: DataFrame with new input
        features for prediction
        :return: A DataFrame with predicted returns for
        each stock
    """
    predictions = []

    for stock_id, model in fitted_models.items():
        # Select the features for the current stock
        # stock_features =
        features.loc[features['Stock_ID'] == stock_id,
        ['Market_Excess_Return', 'SMB', 'HML']]
        # Modify this to
        stock_features =
        features.loc[features['Stock_ID'] == stock_id,
        ['Mkt-RF', 'SMB', 'HML']]
        # Check if there are features to predict on
        if not stock_features.empty:
            # Make prediction
            predicted_return =
            model.predict(stock_features)

            # Store the predictions
            for i, prediction in
            enumerate(predicted_return):
                predictions.append({'Stock_ID':
                stock_id,
                    'Predicted_Return': prediction,
                    'Date': stock_features.index[i]
                })

    return pd.DataFrame(predictions)

# Example usage
```

```
# features = pd.DataFrame(...) # Replace with your  
new input features DataFrame  
predicted_returns = predict_returns(fitted_models,  
training_data)  
print(predicted_returns)
```

In this updated code:

- The `predict_returns` function iterates through each fitted model in `fitted_models`.
- It uses the `predict` method on the subset of features corresponding to each stock.
- The function compiles the predictions into a `DataFrame`.

Ensure that the `features` `DataFrame` is structured correctly with the Fama-French factors and stock IDs. This approach allows for more flexibility and is more aligned with typical model deployment practices.

Granted, we would only know to ask this question if we have prior knowledge that fitted models from *scikit-learn* or *statsmodels* come with a `predict` function, so clearly this won't work for someone completely unfamiliar with Python and its data science libraries. Furthermore, though this code looks useful because it helps us save the fitted models somewhere as well, it isn't quite correct.

A fundamental problem with the `predict_returns` function is that it is expecting the input `stock_features` to have an index, "`stock_id`." This may be appropriate for "cross-sectional" features (see Nautiyal and Chan [2021]), but certainly not for "time-series" features. When we told ChatGPT that these are time-series features that do not have `stock_id` as index, it thinks we must be predicting the returns of the portfolio as a whole, not the returns of the individual stocks in that portfolio. So, more prompting is needed to clarify.

User Prompt

Maybe it is common to use time-series factors to predict portfolio returns, but here, we have trained our models to predict returns for each stock. Modify your function accordingly.

ChatGPT Response

...

```
import pandas as pd
import numpy as np

def predict_returns(fitted_models, features_today):
    """
        Predict returns for each stock for a single day
        using their respective fitted models.

        :param fitted_models: Dictionary of fitted model
        objects for each stock
        :param features_today: DataFrame or Series with
        input features for a single day
        :return: A Series with predicted returns for
        each stock
    """
    predictions = {}

    # Ensure features_today is in the correct format
    # (2D array)
    if isinstance(features_today, pd.Series):
        features_today =
    features_today.values.reshape(1, -1)
    elif isinstance(features_today, pd.DataFrame):
        features_today = features_today.values

    # Iterate over each model and predict the return
    # for each stock
    for stock_id, model in fitted_models.items():
        # Predict the return for the current stock
        # using the common features
        predicted_return =
    model.predict(features_today)

        # If the model returns a list/array, take
        # the first element as the prediction

        if isinstance(predicted_return, (list,
    np.ndarray)):
            predictions[stock_id] =
    predicted_return[0]
```

```
        else:  
            predictions[stock_id] = predicted_return  
  
    return pd.Series(predictions)
```

Ultimately, none of the codes produced by ChatGPT quite worked. Ernie had to use his limited Python skills to mix-and-match different codes to produce the final function.

```
# Predicting returns

import pandas as pd
import numpy as np

def predict_returns(fitted_models, features_today):
    """
        Predict returns for each stock for a single day
        using their respective fitted models.

        :param fitted_models: Dictionary of fitted model
        objects for each stock
        :param features_today: DataFrame or Series with
        input features for a single day
        :return: A Series with predicted returns for each
        stock
    """
    predictions = {}

    # Ensure features_today is in the correct format
    # (2D array)
    features_today = features_today[['Mkt-RF', 'SMB',
    'HML']]
    features_today = features_today.values.reshape(1,
    -1)

    # Iterate over each model and predict the return
    # for each stock
    for stock_id, model in fitted_models.items():
        # Predict the return for the current stock
        # using the common features
        predicted_return =
model.predict(features_today)
        # If the model returns a list/array, take the
        # first element as the prediction
        predictions[stock_id] = predicted_return[0]

    return pd.Series(predictions)
```

If you run this code as

```
predicted_returns = predict_returns(fitted_models,  
features_today)
```

where features today is a pd.Series that looks like

```
Mkt-RF      -0.04  
SMB         0.08  
HML        -0.11  
RF          0.022
```

and fitted_models is an output of the training codes above, the output will look like this:

```
0033A      0.044187  
3511B      0.057673  
3FMCC     -0.139659  
3FNMA     -0.112233  
5933B      0.045648  
...  
XTO       0.016649  
YHOO     -0.012752  
YUM       0.015054  
ZION     -0.039987  
ZMH      -0.007693  
Length: 553, dtype: float64
```

Now we need to incorporate these predictions in a backtest program for the trading strategy. But let's test if ChatGPT understands what a trading strategy is first.

User Prompt

Imagine a trading strategy that buys the top 50 stocks with the highest predicted returns and short the bottom 50, on a daily basis, and hold those positions for the next 30 days. What is the maximum number of positions we can hold for a particular stock at any one time?

ChatGPT Response

Since you are trading daily and holding each position for 30 days, it means that on any given day, you could have up to 30 overlapping positions for the same stock, each initiated on a different day over the past 30 days. This scenario assumes that the stock consistently appears either in the top 50 or bottom 50 each day for 30 consecutive days.

Impressive! ChatGPT is finally showing some conceptual intelligence that we felt wasn't quite apparent until now. However, when we asked it to convert this trading strategy to backtest code (with 1-day holding period for simplicity and to match the existing Matlab code), it produced versions that are wrong in one aspect or another. In one version it is using all the columns in the input data as features instead of just selecting the three Fama-French factors. In another version, it forgot that the input data has multiple rows of the same date that correspond to different *stock_Id*'s. So, we had to ask the following:

User Prompt

This doesn't fix the issue. The problem is that the for loop goes over every row in the data, assuming that each row refers to data of a different date. But that's not true. Currently it will repeatedly calculate portfolio_returns for the same date many times as it iterates through the rows of the data.

To which it regrettably responded with more erroneous codes.

In a third version, ChatGPT thinks it is smart and looks at the data 1-day ahead of the current date to look for the value 31_Day_Returns. But that is unnecessary because we created 1_Day_Returns as the return in the next day already. This is another case where unless you understand Python, you won't be able to catch the error.

After many days (!) of prompting and manual adjustments, we finally arrived at the following working version.

```
# Backtest long-short trading strategy

import pandas as pd
import warnings
warnings.filterwarnings('ignore') # Without suppression
of warnings, the program will freeze in Jupyter
Notebook due to too much output

# Function to execute the trading strategy
def execute_strategy(data, fitted_models,
holding_period=1):
    portfolio_returns = []

    # Get unique dates from the data
    unique_dates = data['Date'].unique()

    for day in range(len(unique_dates) - holding_period):
        # Get the current date
        current_date = unique_dates[day]

        # Filter data for the current date
        data_today = data[data['Date'] == current_date]

        # Get features for this day (assuming the
        features_are_the_same_for_all_stocks_on_a_given_day)
        features_today = data_today[['Mkt-RF', 'SMB',
'HML', 'RF']].iloc[0]

        # Predict returns using the predict_returns
        function
        predicted_returns =
predict_returns(fitted_models, features_today)

        # Implement your strategy: select top 50 and
        bottom 50 stocks
        top_50 = predicted_returns.nlargest(50)
        bottom_50 = predicted_returns.nsmallest(50)

        # Calculate returns for the selected stocks on
        a current date
```

```

        current_date = unique_dates[day]
        current_data = data[data['Date'] ==
current_date]

        #print(current_date)

        long_returns =
current_data[current_data['Stock_ID'].isin(top_50.index
)]['1_Day_Returns'].mean()
        short_returns = -
current_data[current_data['Stock_ID'].isin(bottom_50.in
dex)]['1_Day_Returns'].mean()

        # Daily portfolio return
        daily_portfolio_return = (long_returns +
short_returns) / 2

        portfolio_returns.append(daily_portfolio_return)

    return portfolio_returns

```

To execute this function, ChatGPT helpfully suggested (correctly)

```

portfolio_returns=execute_strategy(training_data,
fitted_models)

```

It produces a list, not a pd.Series, which isn't ideal because it does not include timestamps. We shan't be picky, so we said:

User Prompt

Correct! Please compute the Sharpe ratio.

ChatGPT Response

```
import numpy as np
# Calculate Sharpe Ratio
def calculate_sharpe_ratio(portfolio_returns,
risk_free_rate=0.02):
    excess_returns = np.array(portfolio_returns) -
risk_free_rate / 252
    sharpe_ratio = np.mean(excess_returns) /
np.std(excess_returns) * np.sqrt(252)
    return sharpe_ratio
```

This brings up an important point that Ernie discussed in *Quantitative Trading* (Chan, 2021). Let's see if ChatGPT understands this nuance.

User Prompt

In the calculate_sharpe_ratio function defined above, the risk-free rate is defaulted to 0.02. For a long-short dollar-neutral portfolio, should we deduct risk-free rate from the returns in a Sharpe ratio calculation? In other words, is excess returns the raw returns minus the risk-free rate?

ChatGPT Response

In the context of a long-short dollar-neutral portfolio, the consideration of the risk-free rate in the calculation of the Sharpe ratio can be a bit nuanced. Here's how to think about it:

Excess Returns: Typically, the Sharpe ratio is calculated using excess returns, which are the portfolio's returns minus the risk-free rate. This is to assess how much additional return the investment is providing over a risk-free alternative (like Treasury bills).

Dollar-Neutral Portfolio: In a dollar-neutral portfolio, you're equally long and short in dollar terms.

Theoretically, such a portfolio might be seen as self-financing, implying that its performance could be assessed without considering the risk-free rate. The rationale is that any interest you would earn on the long side (from investing money) might be offset by the interest you would pay on the short side (from borrowing money).

Practical Considerations: Practically, whether or not to include the risk-free rate in the Sharpe ratio calculation for a dollar-neutral portfolio depends on several factors:

Strategy and Leverage: If your strategy involves leverage or financing costs, these should be factored into the return calculation. **Benchmarking:** Consider what is standard practice in your benchmark group or among your peers.

Investor Expectations: Consider what investors in your fund would expect. They might compare your fund's performance to other investments, which typically are assessed using excess returns over the risk-free rate.

Risk-Free Rate in Sharpe Ratio:

Include Risk-Free Rate: If you want to maintain consistency with broader market practices and make your Sharpe ratio comparable to others, you might include the

risk-free rate. Exclude Risk-Free Rate: If you believe that your portfolio's characteristics (like being dollar-neutral) make the risk-free rate less relevant, you could exclude it. In summary, whether to use raw returns or excess returns (returns minus the risk-free rate) in the Sharpe ratio calculation for a long-short dollar-neutral portfolio depends on the specific characteristics of your portfolio and your performance measurement goals. Both approaches have rationales, but it's important to be consistent and clear about the method you choose, especially when communicating with stakeholders or investors.

Impressive! It got most of the nuance right. For a long-short portfolio such as ours, we do not typically subtract risk-free rate (R_f) from the raw returns in Sharpe ratio calculations for the self-financing reason ChatGPT described. In other words, investors with this portfolio can still earn margin interest on the cash deposited in their accounts, and as long as you don't include such margin interest in your P&L, you also don't need to subtract R_f from your returns. But if you are running a hedge fund and this portfolio is its sole holding, the fund's returns (ignoring management and performance fees) will already have included this margin interest, and so in reporting its Sharpe ratio, you must subtract R_f . During the post-COVID era of high interest rates that reached more than 5%, this is an important consideration. Since we are not including R_f as credit interest in our portfolio, we can also set R_f to 0 in our following Sharpe ratio calculation.

```

# Calculate Sharpe Ratio
def calculate_sharpe_ratio(portfolio_returns,
risk_free_rate=0.0):
    excess_returns = np.array(portfolio_returns) -
risk_free_rate / 252
    sharpe_ratio = np.mean(excess_returns) /
np.std(excess_returns) * np.sqrt(252)
    return sharpe_ratio

# Backtest the strategy
sharpe_ratio =
calculate_sharpe_ratio(portfolio_returns)
print("Sharpe Ratio:", sharpe_ratio)

```

which produces 6.5, much higher than the 3.7 that we obtained from Matlab, on the training set. Which is right?

The reason the Python program seemingly outperformed the Matlab program is that we have skipped all stocks with insufficient data in the Python program, whereas in the Matlab program, we have made a prediction of 0 returns for those stocks—which wasn't correct.

Let's see how it performs on the test set.

User Prompt

Let's say I have dataframe combined_data. I want to split it into 2 two halves: training_data and test_data using halfway_point = len(combined_data) // 2 training_data = combined_data.iloc[:halfway_point]. How should I compute test_data?

ChatGPT Response

```
test_data = combined_data.iloc[halfway_point:]
```

which is correct. Running the backtest on the test data resulted in a Sharpe ratio of 0.55, which is much better than the negative Sharpe from the Matlab program. The complete process can be downloaded from `ff_1day.py` on our GitHub. Now let's display the cumulative return ("equity" or "Growth of \$1") curve shown in [Figure 2.1](#).

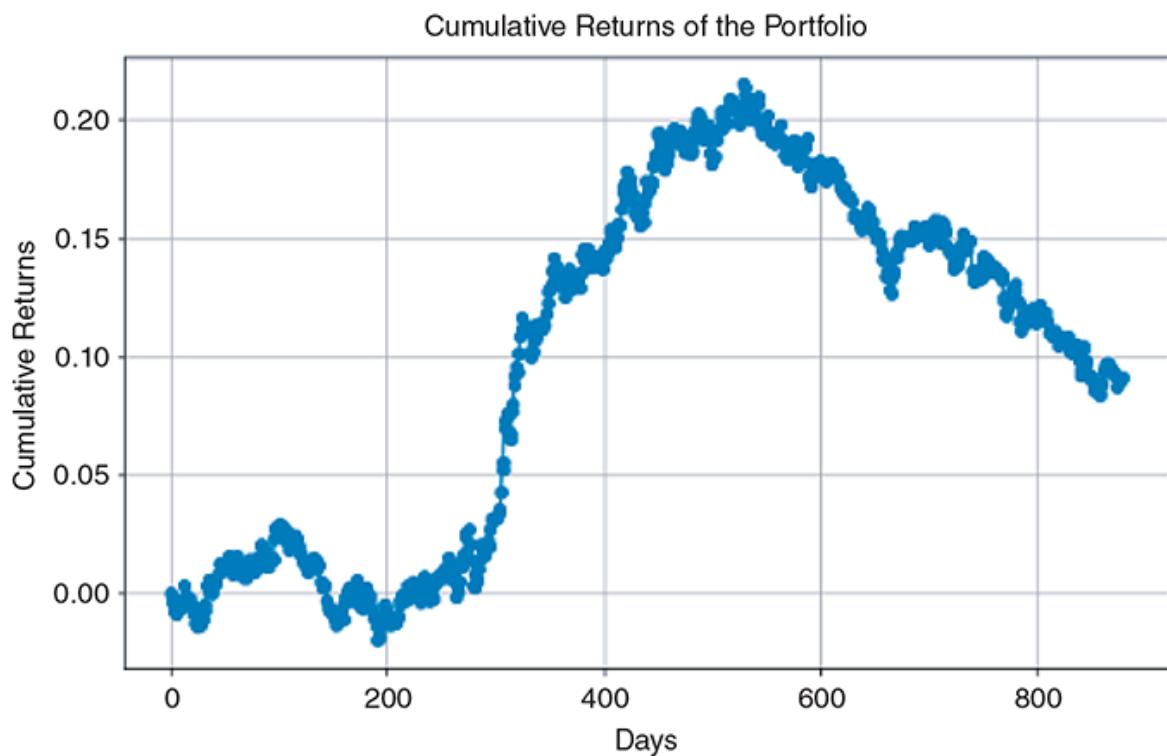


Figure 2.1 Cumulative returns of Fama-French three-factor strategy.

User Prompt

Suppose the daily returns of a portfolio is contained in a list `portfolio_returns`. Please display the cumulative returns curve.

ChatGPT Response

```
import matplotlib.pyplot as plt
import numpy as np

# Example portfolio_returns list (replace this with
# your actual returns)
portfolio_returns = [0.01, -0.005, 0.007, -0.003,
0.002, 0.01, -0.004]

# Calculate cumulative returns
cumulative_returns = np.cumprod(1 +
np.array(portfolio_returns)) - 1

# Plotting the cumulative returns curve
plt.figure(figsize=(10, 6))
plt.plot(cumulative_returns, marker='o',
linestyle='--')
plt.title('Cumulative Returns of the Portfolio')
plt.xlabel('Days')
plt.ylabel('Cumulative Returns')
plt.grid(True)
plt.show()
```

Remember, this is all on the test set. If you are wondering, the drawdown started around 2012-09, and it had been relentless until the end of the data 2023-12. This doesn't mean Fama-French factors can't explain returns any more. It just means factor loadings are no longer persistent from one day to the next. As a side note, we did not try to make the plots generated by ChatGPT look pretty since the goal was just to show the raw outputs it produced. Sure, we could have asked it to create better-looking figures, but that wasn't the point of this exercise. By the way, if you ask ChatGPT to improve the style of plots in Python using Matplotlib or Seaborn, it is actually really good at it.

2.2 Summarizing a Trading Strategy Paper and Creating Backtest Codes from It

Ernie's second book, *Algorithmic Trading* (Chan, 2013, p.143), describes an interday momentum trading strategy involving the VIX futures. The strategy is based on a paper by Simon and Campasano (2014). Let's see if ChatGPT can read and understand that paper and produce backtest codes for it.

User Prompt

Pretend you are a quantitative finance expert. Please summarize the trading strategies described in this paper.

[**Author Note:** Remember we can upload a paper to ChatGPT-4o.]

ChatGPT Response

...

the paper explores relative value spread trades, designed to profit from deviations in the EH [Author Note: Expectations Hypothesis]

...

The paper's results suggest that the VIX futures market presents opportunities for strategic trades based on its term structure anomalies.

That summary is too high level. So, we asked it to be more specific.

User Prompt

More specifically, generate pseudocodes for the relative value spread trades, or any other trades based on term structure anomalies.

ChatGPT Response

The paper describes a beta-neutral trading strategy (BN) that involves maintaining a long position in constant-maturity VIX futures with a long-maturity tenor (L) and a simultaneous short position in constant-maturity VIX futures with a short-maturity tenor (S). The goal of this strategy is to ensure that the net position is VIX-neutral. Here's a simplified pseudocode for the beta-neutral trading strategy:

```
# Define the beta-neutral (BN) trading strategy
Define BN_Trading_Strategy(long_maturity_tenor,
short_maturity_tenor, constant_maturity_betas):
    # Initialize weights for long and short
    # positions
    weight_long = (1 / beta_long) / ((1 / beta_long)
+ (1 / beta_short))
    weight_short = 1 - weight_long

    # Calculate daily payoff
    daily_payoff = weight_long * return_long -
weight_short * return_short

    return daily_payoff

# Example usage
long_maturity_tenor = L # Specify the long-maturity
tenor
short_maturity_tenor = S # Specify the short-
maturity tenor
constant_maturity_betas = Table_4_data # Retrieve
constant-maturity betas from table 4

# Calculate the daily payoff for the beta-neutral
# trading strategy
daily_payoff =
BN_Trading_Strategy(long_maturity_tenor,
short_maturity_tenor, constant_maturity_betas)
```

In this pseudocode:

long_maturity_tenor and short_maturity_tenor are the tenors for the long and short VIX futures positions, respectively. constant_maturity_betas represents the betas for each maturity, which can be retrieved from a specified table (Table 4 in the paper). The weights for the long (weight_long) and short (weight_short) positions are calculated based on these betas. The daily_payoff is computed as the weighted difference between the returns of the long and short positions.

This actually *isn't* the trading strategy the paper proposes. The strategy proposed isn't a calendar spread strategy. It is a strategy to buy or short VIX futures based on whether it is in backwardation or contango and then hedge its position by the ES future with the same sign. Instead, we want ChatGPT to focus on the following paragraph in the paper:

The simulations examine the profitability of shorting or buying VIX futures contracts when the VIX futures basis is in contango or backwardation, respectively, and the efficiency of hedging market risk with mini-S&P futures positions. The simulations then examine the impact of modifying entry rules and then introducing exit rules based on risk management strategies. All trading simulations examine the profitability of shorting or buying the front VIX futures contract that has at least ten business days until settlement when the magnitude of the daily roll is greater in magnitude than .10 VIX futures point (\$100) per day.¹³ The daily roll is defined as the difference between the front VIX futures price and the VIX, divided by the number of business days until the VIX futures contract settles, and measures potential profits assuming that the basis declines linearly until settlement.”

Let's see how we can prompt it to do so.

User Prompt

In your pseudocodes above, you missed an important trading rule. See especially the paragraph starting with "The simulations examine the profitability of shorting or buying VIX futures contracts when the VIX futures basis is in contango or backwardation,..."

[Author Note: We continued with the rest of the paragraph just quoted.]"

ChatGPT Response

...

```

# Define the beta-neutral trading strategy with
contango/backwardation consideration
Define
BN_Trading_Strategy_With_Contango_Backwardation(long
_maturity_tenor, short_maturity_tenor,
constant_maturity_betas, VIX_term_structure_state):
    # Determine trade direction based on the state
    # of the VIX futures term structure
    if VIX_term_structure_state == "contango":
        # In contango, buy long-maturity and sell
        short-maturity VIX futures
        trade_direction = "buy_long_sell_short"
    elif VIX_term_structure_state ==
"backwardation":
        # In backwardation, sell long-maturity and
        buy short-maturity VIX futures
        trade_direction = "sell_long_buy_short"
    else:
        return "No trade"

    # Calculate weights for long and short positions
    weight_long = (1 / beta_long) / ((1 / beta_long)
+ (1 / beta_short))
    weight_short = 1 - weight_long

    # Calculate daily payoff based on trade
    direction
    if trade_direction == "buy_long_sell_short":
        daily_payoff = weight_long * return_long -
weight_short * return_short
    elif trade_direction == "sell_long_buy_short":
        daily_payoff = weight_short * return_short -
weight_long * return_long

    return daily_payoff

# Example usage
VIX_term_structure_state =
Determine_VIX_Term_Structure_State() # Function to
determine if VIX is in contango or backwardation
daily_payoff =

```

```
BN_Trading_Strategy_With_Contango_Backwardation(long  
_maturity_tenor, short_maturity_tenor,  
constant_maturity_betas, VIX_term_structure_state)
```

In this revised pseudocode, the function `Determine_VIX_Term_Structure_State()` is assumed to analyze the VIX term structure and return either “contango” or “backwardation”. The trading strategy then adjusts its positions accordingly, either buying long and selling short in a contango state or the reverse in a backwardation state.

It still isn’t the strategy described in the paragraph, but at least it mentioned the term structure criterion. Does ChatGPT know how the authors of the paper define backwardation or contango of the VIX futures? We shall find out.

User Prompt

Based on the paper, provide definition of the function `Determine_VIX_Term_Structure_State()`. That is, define backwardation or contango based on the paper’s definition.

ChatGPT Response

Based on the information in the paper, the Determine_VIX_Term_Structure_State() function can be defined as follows:

Contango: The VIX futures curve is in a state of contango when it is upward sloping. This means that the futures prices are higher than the spot price of the VIX index, and they increase as the time to maturity increases. In a contango market, the long end of the futures curve is less volatile, and the price ranges (difference between high and low readings) and the volatility of prices decrease with increasing time to maturity.

Backwardation: The VIX futures curve is in a state of backwardation when it is downward sloping. This occurs when the futures prices are lower than the spot price of the VIX index, and they decrease as the time to maturity increases. The long end of the curve in backwardation would be more responsive to changes in the spot price.

Here's how the Determine_VIX_Term_Structure_State() function could be defined in pseudocode:

```

# Define a function to determine the state of the
VIX term structure
Define
Determine_VIX_Term_Structure_State(VIX_spot_price,
VIX_futures_prices):
    # Compare spot price with futures prices for
    different maturities
    is_upward_sloping = True
    for each maturity in VIX_futures_prices:
        if VIX_futures_prices[maturity] <
VIX_spot_price:
            is_upward_sloping = False
            break

    # Determine the term structure state based on
    the slope
    if is_upward_sloping:
        return "contango"
    else:
        return "backwardation"

# Example usage
VIX_spot_price = Current_VIX_Spot_Price() # Function to get the current VIX spot price
VIX_futures_prices = Get_VIX_Futures_Prices() # Function to get VIX futures prices for different maturities

term_structure_state =
Determine_VIX_Term_Structure_State(VIX_spot_price,
VIX_futures_prices)

```

In this pseudocode:

The `VIX_spot_price` is the current spot price of the VIX index. `VIX_futures_prices` is a dictionary (or similar data structure) that holds futures prices for various maturities. The function compares the spot price with futures prices for each maturity to check if the curve is upward or

downward sloping, thus determining if the term structure is in contango or backwardation.

It got the definition of backwardation and contango right. But still, we are missing trading rules that incorporate them.

User Prompt

That is correct definition of backwardation and contango. But note the paragraph starting with “The simulations examine....” It has a version of the strategy embedded there. Please summarize that version and provide pseudocodes.

ChatGPT Response

Thank you for providing the detailed paragraph. Based on this information, the revised trading strategy involves:

Determining the State of the VIX Futures Basis: The strategy involves shorting VIX futures contracts when the VIX futures basis is in contango and buying them when in backwardation.

Entry Rules: Trades are entered when the front VIX futures contract, which has at least ten business days until settlement, shows a daily roll greater in magnitude than 0.10 VIX futures points (\$100) per day.

Daily Roll Calculation: The daily roll is calculated as the difference between the front VIX futures price and the VIX, divided by the number of business days until the VIX futures contract settles. This measures potential profits assuming that the basis declines linearly until settlement.

Exit Rules: Based on risk management strategies, the strategy may also involve specific exit rules, though these are not detailed in the paragraph you provided.

Alternative Thresholds: Footnote 13 indicates that simulations were also run with $+/- 0.15$ daily roll thresholds, resulting in fewer but more profitable trades.

Here's a pseudocode representation of this strategy:

```
# Define the trading strategy based on VIX futures
basis and daily roll
Define Trading_Strategy(VIX_spot_price,
front_VIX_futures_price, days_until_settlement,
daily_roll_threshold = 0.10):
    # Calculate the daily roll
    daily_roll = (front_VIX_futures_price -
VIX_spot_price) / days_until_settlement

    # Determine the state of the VIX futures basis
    if daily_roll > daily_roll_threshold:
        basis_state = "contango"
    elif daily_roll < -daily_roll_threshold:
        basis_state = "backwardation"
    else:
        return "No trade"

    # Determine the trade action based on the basis
state
    if basis_state == "contango":
        trade_action = "short VIX futures"
    elif basis_state == "backwardation":
        trade_action = "buy VIX futures"

    return trade_action

# Example usage
VIX_spot_price = Current_VIX_Spot_Price()  # Function to get the current VIX spot price
front_VIX_futures_price =
Get_Front_VIX_Futures_Price()  # Function to get the price of the front VIX futures contract
days_until_settlement =
Get_Days_Until_Settlement(front_VIX_futures_contract)
    # Function to get the number of business days until settlement of the front VIX futures contract

trade_action = Trading_Strategy(VIX_spot_price,
front_VIX_futures_price, days_until_settlement)
```

Getting closer. However, it is still strange that it completely ignored the explicit requirement (much discussed in the paper) to hedge the VIX future with E-mini future.

User Prompt

Apparently you have forgotten the specification "hedging market risk with mini-S&P futures positions". How would that enter your strategy specification and pseudocode?

ChatGPT Response

Incorporating the hedging of market risk with mini-S&P futures into the trading strategy requires adding a component to the strategy that adjusts the position in mini-S&P futures based on the market exposure of the VIX futures position. The idea here is to hedge the market risk associated with the VIX futures trade by taking an offsetting position in mini-S&P futures.

Here's how this could be integrated into the strategy:

Assess Market Exposure: After determining the trade action for VIX futures (either shorting or buying), assess the market exposure this position creates. For instance, a long position in VIX futures could be considered as having negative market exposure (since VIX often increases when the market falls), and a short position in VIX futures as positive market exposure.

Hedge with Mini-S&P Futures: Based on the assessed market exposure from the VIX futures position, take an offsetting position in mini-S&P futures. If the VIX futures position has negative market exposure, a long position in mini-S&P futures would hedge this risk, and vice versa.

Here is the revised pseudocode with market risk hedging:

```

# Define the trading strategy with market risk
# hedging using mini-S&P futures
Define Trading_Strategy_With_Hedging(VIX_spot_price,
front_VIX_futures_price, days_until_settlement,
S_and_P_position_size, daily_roll_threshold = 0.10):
    # Calculate the daily roll
    daily_roll = (front_VIX_futures_price -
VIX_spot_price) / days_until_settlement

    # Determine the state of the VIX futures basis
    if daily_roll > daily_roll_threshold:
        basis_state = "contango"
    elif daily_roll < -daily_roll_threshold:
        basis_state = "backwardation"
    else:
        return "No trade"

    # Determine the trade action based on the basis
    # state
    if basis_state == "contango":
        VIX_trade_action = "short VIX futures"
        S_and_P_trade_action = "long mini-S&P
futures" # Hedge against positive market exposure
    elif basis_state == "backwardation":
        VIX_trade_action = "buy VIX futures"
        S_and_P_trade_action = "short mini-S&P
futures" # Hedge against negative market exposure

    # Adjust mini-S&P futures position size to hedge
    # market risk

    Adjust_Mini_SP_Futures_Position(S_and_P_position_
size, S_and_P_trade_action)

    return VIX_trade_action, S_and_P_trade_action

# Example usage
VIX_spot_price = Current_VIX_Spot_Price()
front_VIX_futures_price =
Get_Front_VIX_Futures_Price()
days_until_settlement =

```

```
Get_Days_Until_Settlement(front_VIX_futures_contract  
)  
S_and_P_position_size = Determine_Position_Size() #  
Function to determine the size of the mini-S&P  
futures position  
  
VIX_trade_action, S_and_P_trade_action =  
Trading_Strategy_With_Hedging(VIX_spot_price,  
front_VIX_futures_price, days_until_settlement,  
S_and_P_position_size)
```

In this pseudocode, `Adjust_Mini_SP_Futures_Position` is a function that adjusts the position in mini-S&P futures based on the calculated trade action and position size. This function would take into account the specifics of the hedge, such as the degree of leverage or the precise sizing of the mini-S&P position relative to the VIX position, to effectively balance the market risk.

After basically spoon-feeding information to ChatGPT (treating it as an inexperienced and mediocre intern), it finally added the hedge. But wait, whenever there is a hedge, we need to specify a hedge ratio. Doesn't ChatGPT know basic finance?

User Prompt

This is correct. Does the paper specify how we should determine the hedge ratio between the VIX and ES futures?

ChatGPT Response

I have thoroughly reviewed the paper “VIX Futures Term Structure and the Expectations Hypothesis” to find specific information regarding the determination of the hedge ratio between VIX and mini-S&P (ES) futures. However, the sections of the paper I was able to access did not provide explicit details or formulas for calculating this hedge ratio.

This mediocre intern is becoming even more annoying. But as mentors, we shall not lose patience.

User Prompt

See this paragraph starting with “The hedge ratio—the number of mini-S&P futures contracts to buy or sell per VIX futures position—is constructed using the out of sample parameter estimates from [equation 3](#).” Are you able to summarize the method of calculating the hedge ratio and display the pseudocodes?

ChatGPT Response

..., the hedge ratio (HR) is calculated using the following formula:

$$HR_t = \frac{\beta_1 \times 1000 + \beta_2 \times TTS_{t-1} \times 1000}{0.01 \times ES_{t-1} \times 50}$$

where:

- HR_t is the hedge ratio at time t .
- β_1 and β_2 are parameters estimated from [equation 3](#) (not provided in your quote).
- TTS_{t-1} is the lagged value of some variable (not specified in your quote) at time $(t - 1)$.
- ES_{t-1} is the lagged value of the mini-S&P futures contract at time $(t - 1)$.
- The value of a one-point VIX futures price change is \$1,000.
- The gain or loss for a one percent mini-S&P futures price change is calculated as 0.01 times the lagged value of the mini-S&P futures contract times \$50, which is the value of one mini-S&P futures point.

The average hedge ratio is roughly one mini-S&P futures contract per VIX futures contract, with a range from 1/2 to 2 contracts, influenced mainly by fluctuations in the number of business days to settlement of the relevant VIX futures contracts and the level of S&P futures prices.

Here is the pseudocode for this hedge ratio calculation:

```

# Define a function to calculate the hedge ratio
Define Calculate_Hedge_Ratio(beta1, beta2,
TTS_lagged, ES_lagged):
    # Calculate the value of a one-point VIX futures
    # price change
    VIX_price_change_value = 1000

    # Calculate the gain or loss for a one percent
    # mini-S&P futures price change
    mini_SP_price_change_gain_loss = 0.01 *
    ES_lagged * 50

    # Calculate the hedge ratio
    HR = (beta1 * VIX_price_change_value + beta2 * 
    TTS_lagged * VIX_price_change_value) /
    mini_SP_price_change_gain_loss

    return HR

# Example usage
beta1 = Parameter_Estimate1 # Estimated from
equation 3
beta2 = Parameter_Estimate2 # Estimated from
equation 3
TTS_lagged = Lagged_Value_TTS # Lagged value of the
TTS variable
ES_lagged = Lagged_Value_ES # Lagged value of the
mini-S&P futures contract

hedge_ratio = Calculate_Hedge_Ratio(beta1, beta2,
TTS_lagged, ES_lagged)

```

In this pseudocode, `Parameter_Estimate1` and `Parameter_Estimate2` represent the out of sample parameter estimates β_1 and β_2 from [equation 3](#). `Lagged_Value_TTS` and `Lagged_Value_ES` are the lagged values of the TTS variable and the mini-S&P futures contract, respectively. The function `Calculate_Hedge_Ratio`

computes the hedge ratio based on these inputs and the formula provided.

Apparently, ChatGPT couldn't remember the entire paper, hence it doesn't know what β_1 and β_2 are, nor does it know what TTS_{t-1} is. To avoid prolonging the agony, let's just quote what the paper said.

"The hedge ratios are constructed from regressions of VIX futures price changes on a constant and on contemporaneous percentage changes of the front mini-S&P 500 futures contract both alone and multiplied by the number of days that the VIX futures contract is from settlement, as shown below.

$$\Delta VIX_t^F = \beta_0 + \beta_1 * SPRET_t + \beta_2 * SPRET_t * TTS_t + \mu_t \quad (3)''$$

That's the mysterious [equation 3](#) that ChatGPT can't remember. Let's leave that aside and move on to a bigger issue: what about the exit rule? The paper specified that

"These simulations also assume that trades are exited at the end of 5 business days...."

Let's see if ChatGPT spotted that!

User Prompt

In the trading strategy specification Define Trading_Strategy_With_Hedging(VIX_spot_price, front_VIX_futures_price, days_until_settlement, S_and_P_position_size, daily_roll_threshold = 0.10), it appears that you only specified when we should enter a trade. What is the exit rule?

ChatGPT Response

The exit rule for the trading strategy involving VIX futures and hedging with mini-S&P futures, as described in your earlier quote from the paper, isn't explicitly detailed in the information provided...

User Prompt

That isn't true! See the paragraph that contains the sentence, "These simulations also assume that trades are exited at the end of"

ChatGPT Response

...

```
# Define the trading strategy with market risk
# hedging and a fixed duration exit rule
Define
Trading_Strategy_With_Hedging_And_Fixed_Duration_Exi
t(VIX_spot_price, front_VIX_futures_price,
days_until_settlement, S_and_P_position_size,
daily_roll_threshold = 0.10, holding_period = 5):
    # Entry rules as before
    ...

    # Exit rule based on a fixed holding period of 5
    business days
    Define
    Fixed_Duration_Exit_Rule(current_holding_days,
holding_period):
        if current_holding_days >= holding_period:
            return "exit"
        else:
            return "hold"

    # Monitor the holding period and decide whether
    to exit
    current_holding_days = Count_Holding_Days() # Function to count the number of business days since
    the trade was entered
    exit_decision =
    Fixed_Duration_Exit_Rule(current_holding_days,
holding_period)

    return exit_decision

# Example usage
...
exit_decision =
Trading_Strategy_With_Hedging_And_Fixed_Duration_Exi
t(...)
```

In this pseudocode:

holding_period is set to 5 business days, as specified in your sentence. The function Fixed_Duration_Exit_Rule checks whether the current holding period has reached or exceeded the specified duration of 5 business days. Count_Holding_Days is a hypothetical function that counts the number of business days since the trade was initiated.

ChatGPT finally found the relevant sentence in question! Now let's convert to Python codes.

User Prompt

Good. Now convert the pseudocodes to a complete Python program to backtest the complete trading strategy.

ChatGPT Response

```
import pandas as pd
import numpy as np
from datetime import timedelta

# Function to simulate getting historical data
# (Replace with actual data source)
def get_historical_data():
    # Dummy data for illustration
    data = pd.DataFrame({
        'date': pd.date_range(start='2020-01-01',
periods=100, freq='B'),
        'VIX_spot': np.random.uniform(20, 30, 100),
        'VIX_futures': np.random.uniform(20, 30,
100),
        'ES_futures': np.random.uniform(3000, 4000,
100)
    })
    return data

# Function to calculate the daily roll
def calculate_daily_roll(VIX_futures, VIX_spot,
days_until_settlement):
    return (VIX_futures - VIX_spot) /
days_until_settlement

# Trading strategy implementation
def trading_strategy(data,
daily_roll_threshold=0.10, holding_period=5):
    positions = []
    entry_days_remaining = None

    for index, row in data.iterrows():
        if entry_days_remaining is None:
            # Entry logic
            daily_roll =
calculate_daily_roll(row['VIX_futures'],
row['VIX_spot'], days_until_settlement=10)
            if abs(daily_roll) >
```

```

daily_roll_threshold:
    position = 'short' if daily_roll > 0
else 'long'
    entry_days_remaining =
holding_period
else:
    # Holding the position
    entry_days_remaining -= 1
    if entry_days_remaining <= 0:
        # Exit logic after holding period
        positions.append(position)
        position = None
        entry_days_remaining = None

return positions

# Main function to run the backtest
def backtest_strategy():
    data = get_historical_data()
    positions = trading_strategy(data)
    print("Trade positions:", positions)

# Run the backtest
backtest_strategy()

```

This script includes:

A dummy function `get_historical_data` to simulate historical data. You should replace this with actual data retrieval from a reliable source. Functions `calculate_daily_roll` and `trading_strategy` implement the entry and exit logic based on the daily roll and the fixed holding period. The `backtest_strategy` function runs the backtest and prints the trade positions. Please note:

This script is highly simplified and doesn't include real data sources, transaction costs, slippage, risk management features, or performance metrics.

The hedge ratio and hedging with mini-S&P futures are not included in this script. You would need to integrate

this based on the specific hedge ratio formula and market data. For a realistic backtesting environment, consider using a backtesting framework like backtrader or zipline. This script is a starting point, and you would need to expand and refine it for more comprehensive and realistic backtesting.

Strangely, ChatGPT didn't translate the complete pseudocodes it generated earlier into the corresponding Python. This is quite inexcusable.

User Prompt

Why don't you include the hedging positions with E-mini futures, since you already have the pseudocodes for Define_Calculate_Hedge_Ratio? Add that into the Python codes.

It responded by adding the calculate_hedge_ratio function definition and so forth. The final code is correct, and we saved it in VX_ES.py on our GitHub.

Overall, we find that ChatGPT is an exceptionally poor reader and content summarizer when technical details are involved. It cannot remember many details from a typical finance paper of about 40 pages. Even when presented with specific quotes from the paper, it took a great deal of human intelligence, expertise, and prompting to get it to generate pseudo backtest codes from a paper. This is far from the AGI paradise (or dystopia) that we were promised. Count us as unimpressed.

2.3 Searching for a Portfolio Optimization Algorithm Based on Machine Learning

Enough about spoon-feeding ChatGPT specific strategies that we want to backtest. How about if we use it to find these algorithms? One of us has developed a machine-learning-based portfolio optimization method called CPO (Chan et al., 2023). The CPO method uses a large number of input features spanning different markets, including fundamental, macro-economic factors, technical, and even microstructure data. A machine learning algorithm is then trained on these data in addition to many historical hypothetical capital allocations, in order to determine which capital allocation will give the best Sharpe ratio for some forward period at any time.

However, this isn't the only way machine learning can be applied to portfolio optimization. A more "conventional" approach is to apply deep reinforcement learning (DRL), with the current portfolio allocation as the current "state," future returns as the "reward," reallocation decisions as "actions" that lead to the next state. The typical input to DRL are just returns or prices—we haven't seen any examples of using fundamental and other big data input, especially one using time-series features. So, we ask Copilot for such references. (We tried ChatGPT first, but it did not provide satisfactory responses.)

Again, Kıcıman et al. (2023) said that we should warm up and orient LLMs to a specific domain to get better results.

User Prompt

Pretend you are a quantitative finance and deep learning expert in the following.

Copilot Response

[Author Note: some babble].

User Prompt

We intend to apply deep reinforcement learning to the portfolio optimization problem. What types of input have researchers used for this problem?

Copilot Response

Applying deep reinforcement learning (DRL) to portfolio optimization is a cutting-edge approach that combines the complexity of financial markets with advanced AI techniques. In this context, researchers have used various types of inputs to inform the DRL models. Here are some common types of inputs:

1. **Price-based Data:** This is the most common input type and includes historical prices, returns, and other derived statistics like moving averages, volatility (measured as standard deviation of returns), and momentum indicators. This data can be at different frequencies (daily, hourly, minute-level, etc.).
2. **Volume Data:** Trading volume and its derivatives can provide insights into market sentiment and liquidity, which are important factors in portfolio optimization.
3. **Fundamental Data:** For equities, this might include company financials like earnings, revenue, debt levels, P/E ratios, etc. For other asset classes, relevant fundamental data would be used (e.g., economic indicators for forex or commodities).
4. **Alternative Data:** This can range from satellite imagery for predicting crop yields in commodity markets to social media sentiment analysis for gauging market sentiment.
5. **Market Indicators:** Various market indices, inter-market spreads, and macroeconomic indicators (like interest rates, inflation rates, GDP growth rates) can be important for understanding the broader market context.
6. **Technical Indicators:** Indicators like the Relative Strength Index (RSI), Bollinger Bands, MACD (Moving

Average Convergence Divergence), and others are often used by traders and can be useful inputs for a DRL model.

7. **Risk Metrics:** Measures like Value at Risk (VaR), Conditional Value at Risk (CVaR), or Sharpe Ratio can also be inputs to guide the DRL algorithm toward not just return maximization but also risk-adjusted return optimization.
8. **News and Event Data:** Real-time news feeds, earnings reports, and other event data can have a significant impact on market dynamics and, hence, on portfolio performance.
9. **Order Book Data:** In high-frequency trading scenarios, order book data (like bid-ask spreads, order depth, etc.) can be crucial.
10. **Macro Signals:** Signals from larger economic trends or geopolitical events can also be factored into more sophisticated models.

...

That looks encouraging—it mentions a large variety of input, exactly what we have in mind. So, let's drill down.

User Prompt

Can you give me references to research that used fundamental data as input for a DRL solution to portfolio optimization?

Copilot Response

...

One study, “Deep Reinforcement Learning for Portfolio Optimization Using Latent Feature State Space (LFSS) Module,” explores the use of reinforcement learning in a financial context... .

...

Another relevant study is titled “A New Deep Reinforcement Learning Model for Dynamic Portfolio Optimization.”...

...

We actually know of one specific paper (Cong et al., 2021) that uses DRL with fundamental features, although it is more of a stock-selection algorithm than a general portfolio optimization one. It is still disappointing that ChatGPT didn't mention it. Let's see what these other papers are about.

User Prompt

In the Deep Reinforcement Learning for Portfolio Optimization Using Latent Feature State Space (LFSS) Module you cited, what kind of input data was used?

Copilot Response

...

The primary focus was on price and return data, which is typically derived from Open, High, Low, Close (OHLC) data.

...

That's not what we wanted!

User Prompt

In the new deep reinforcement learning model for dynamic portfolio optimization paper you cited, what kind of input data was used?

Copilot Response

The paper “A New Deep Reinforcement Learning Model for Dynamic Portfolio Optimization” did not provide specific details in the sections I accessed about the exact types of input data used ...

Well, that's quite disappointing. What's the use of AI if we have to read the original paper to find that information?

User Prompt

So, neither of the two papers you cited actually used fundamental data as input, yet you claimed that fundamental data is one type of data used for DRL. What evidence do you have on which to base that claim?

Copilot Response

You are correct in noting that the two papers I cited earlier did not specifically mention the use of fundamental data as input in their deep reinforcement learning models for portfolio optimization. However, the claim that fundamental data is used in such models is based on broader trends and practices in the field of quantitative finance and deep learning.

...

In summary, while the specific papers I cited did not use fundamental data, it remains an important type of input for DRL-based portfolio optimization. I appreciate your diligence and thank you for pointing out the oversight!

For further exploration, you can refer to research papers that explicitly focus on fundamental data integration in DRL models for portfolio management.

User Prompt

Yes, so which research papers should I explore that cover that exact topic?

Copilot Response

Certainly! Here are some research papers that specifically explore the integration of fundamental data into deep reinforcement learning (DRL) models for portfolio optimization:

“Deep Reinforcement Learning for Portfolio Optimization Using Latent Feature State Space (LFSS) Module”

Abstract: This paper presents a DRL framework for autonomous portfolio optimization. It includes a Latent Feature State Space (LFSS) Module for filtering and feature extraction from financial data. The study confirms robust results and performance advantages over several benchmarks.

“Deep Reinforcement Learning in Portfolio Management”

Abstract: This paper implements two state-of-the-art continuous reinforcement learning algorithms (Deep Deterministic Policy Gradient and Proximal Policy Optimization) in portfolio management. While it doesn't explicitly focus on fundamental data, it provides insights into DRL techniques for portfolio optimization.

“A New Deep Reinforcement Learning Model for Dynamic Portfolio Optimization” **Abstract:** This model integrates data processing methods, deep learning, and reinforcement learning to enhance investor decision-making. While it doesn't directly use fundamental data, it explores dynamic portfolio optimization.

“Deep Reinforcement Learning for Optimizing Finance Portfolio Management” **Abstract:** This study examines how DRL can be adopted to maximize portfolio management in the financial sector. While it covers various aspects, including technical indicators, it provides insights into DRL applications.

...

It is great that Copilot finally cited some papers relevant to our query, but when we followed up by asking what specific features these papers used, it wasn't able to answer. In fact, you can verify that these papers did not use any fundamental data or time-series features at all. Once again, count us as unimpressed.

2.4 Explore Options Term Structure Arbitrage Strategies

In the ideal world of the famous Black-Scholes options pricing model that every quant finance student is forced to learn, volatility is constant, and prices can be described by Gaussian Brownian Motion (i.e., a random walk of log prices). None of these assumptions is true in the real world. That is a good thing for traders—if these assumptions were correct, implied volatilities would have been constant for any maturity, and so would these options' annualized time values. In other words, the term structure of options prices would be flat, and term structure arbitrage would have been impossible. By term structure arbitrage, we meant the possibility of earning options premium by buying and shorting options of different maturities without taking directional risk in the underlying. The simplest term structure arbitrage strategy is calendar spreads trading where all the options involved have the same strike price.

Let's see if ChatGPT can help us construct a term structure arbitrage strategy. Note that this task is more challenging than any of the other tasks we have asked ChatGPT to perform so far. Here we haven't specified a trading strategy and are asking ChatGPT to construct one based only on conceptual hints.

User Prompt

Pretend you are a quantitative finance and especially an options trading expert in the following.

User Prompt

Please summarize different trading strategies involving term structure arbitrage in options

ChatGPT Response

Term structure arbitrage in options trading involves exploiting inefficiencies in the pricing of options across different expiration dates. This type of arbitrage takes advantage of discrepancies in the implied volatility curve or the term structure of interest rates, which can affect option prices differently depending on their time to expiration. Here are several strategies that traders might use to exploit these inefficiencies:

1. Calendar Spreads (Horizontal Spreads)

- **Description:** Involves buying and selling two options of the same type (either calls or puts) and strike, but with different expiration dates. The trader might sell a short-term option and buy a long-term option, betting on the volatility difference over time.
- **Goal:** Profit from the difference in time decay or changes in the volatility term structure.

2. Diagonal Spreads

- **Description:** Similar to calendar spreads but involves options with different strikes and expiration dates. It's a combination of a vertical and a calendar spread.
- **Goal:** To take advantage of differences in volatility and time decay between two options.

3. Butterfly Spreads in Time

- **Description:** This strategy involves setting up a butterfly spread (buying one in-the-money option, selling two at-the-money options, and buying one out-of-the-money option) across different expiration dates.

- **Goal:** To profit from the convergence of the term structure toward a normal state if it is currently in an abnormal state.

4. Variance Swaps

- **Description:** Entering into a variance swap contract allows traders to speculate or hedge against the future variability of an asset's price, which can be related to the differences in the implied volatility term structure over time.
- **Goal:** To profit from the difference between the realized variance of an asset's returns and the implied variance from the term structure.

5. Interest Rate Options Arbitrage

- **Description:** Utilizing options on interest rate futures or directly on bonds, traders can exploit inefficiencies in the term structure of interest rates.
- **Goal:** To profit from the mispricing of options based on expected changes in interest rates affecting different maturities differently.

6. Volatility Surface Arbitrage

- **Description:** This involves analyzing the entire volatility surface (across both strike prices and expiration dates) to identify mispricings.
- **Goal:** To exploit discrepancies in implied volatility for options with different strikes and maturities.

7. Dynamic Hedging Strategies

- **Description:** Involves frequently adjusting the composition of a portfolio of options with different expirations to maintain a delta-neutral or gamma-neutral position.

- **Goal:** To profit from changes in the term structure of volatility or interest rates while managing risk.

Implementation Considerations

- **Market Analysis:** Requires deep understanding of the factors influencing option prices, including volatility, interest rates, and their term structures.
- **Risk Management:** Strategies involve significant risks, including the potential for loss if the term structure moves against the position or if the anticipated arbitrage opportunity does not materialize.
- **Execution:** Efficient execution and low transaction costs are crucial due to the potentially small margins of arbitrage opportunities.

These strategies are sophisticated and carry a high level of risk, requiring a solid understanding of options pricing models, market dynamics, and rigorous risk management practices. Traders employ these strategies in markets where they have identified specific inefficiencies or expect certain shifts in the term structure of volatility or interest rates.

It is good that ChatGPT describes a plethora of possible strategies, the simplest of them uses just calendar spreads. So, let's learn more about it.

User Prompt

Elaborate on the calendar spread strategy.

ChatGPT Response

The calendar spread strategy, also known as a time spread or horizontal spread, is a versatile options trading strategy that involves simultaneously entering into two options positions of the same type (either two calls or two puts), same strike price, but with different expiration dates. The trader typically sells a short-term option (the front month) and buys a long-term option (the back month) on the same underlying asset. This strategy is designed to take advantage of differences in time decay (theta) and potentially implied volatility between the two options.

How It Works

- **Short-Term Option (Sell)**: By selling a short-term option, the trader collects premium due to the option's higher theta, or time decay, as options lose value faster as they approach expiration.
- **Long-Term Option (Buy)**: By buying a long-term option, the trader pays for the option which has a slower rate of time decay. This position benefits from any increase in implied volatility or favorable movements in the underlying asset's price over time.

Objective

The primary goal of a calendar spread is to profit from the differential in time decay between the short-dated and longer-dated options. The ideal scenario is for the short-term option to expire worthless or nearly so, allowing the trader to keep the premium collected, while still holding the longer-term option that retains significant time value.

...

This doesn't sound like a viable strategy at all, since a long-dated option will typically have a higher time value than a short-dated one. If we don't expect any directional movement of the underlying, then after the short-dated option expires, we will just be losing (time) value of the long-dated option all the way to its expiration. Maybe if we keep rolling forward the short-dated option to the next short-term expiration period this will work, but ChatGPT didn't suggest that.

How about the opposite strategy of shorting the long-dated option and buying the short-dated one? In this case, we have to keep rolling over (buying) the short-dated long position forward as a hedge to the long-dated short position.

In either strategy, we need to compare the annualized premium (price) of the short-dated options with the annualized premium of shorting the long-dated option, assuming there is no change in underlying price nor volatility.

User Prompt

You suggested that we should short a short-term option and buy a long-term option. Let's assume this is a put on the SPX. Can you plot the annualized time values of these two puts as a function of their time-to-expiration?

ChatGPT Response

... [Author Note: See plot in [Figure 2.2.](#)]

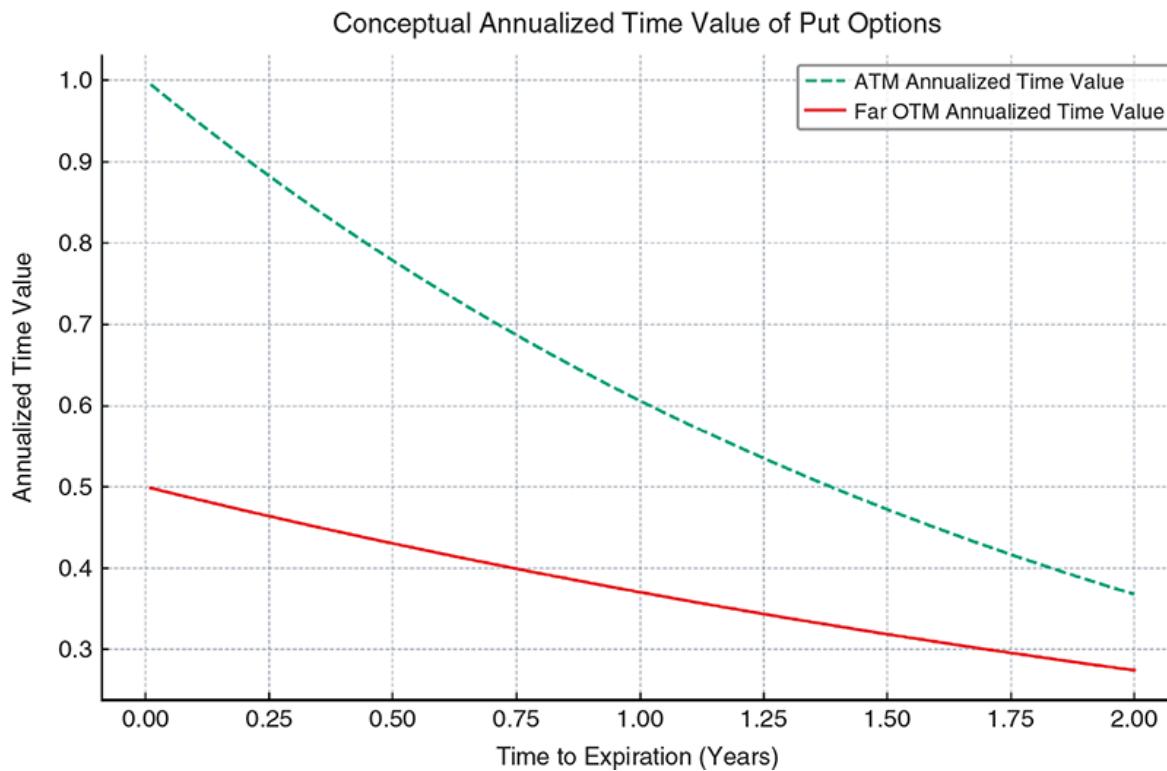


Figure 2.2 Incorrect plot of annualized time value of put options as function of time to expiration.

User Prompt

Can you plot the same for a far out-of-the-money put?

The plot in [Figure 2.2](#) isn't correct—we compared these curves with actual option prices we observed from live market data, and they are wildly off the mark for OTM options. Let's explore further.

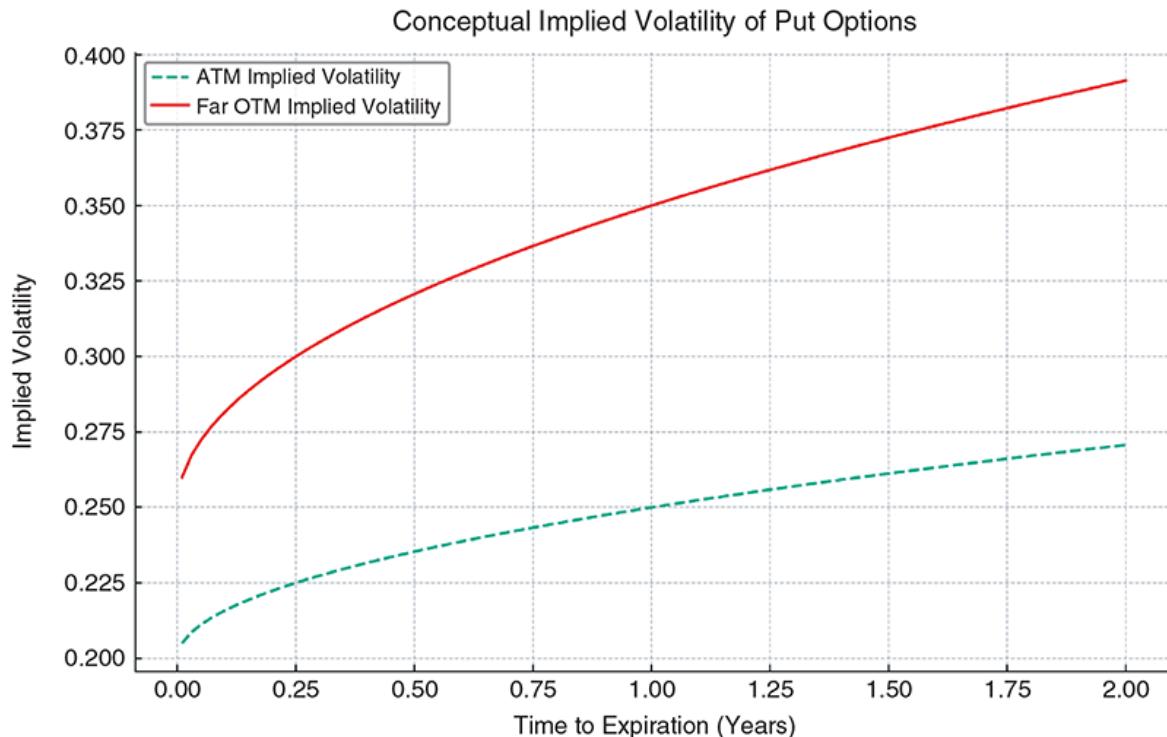
User Prompt

Please plot the same for implied volatilities this time.

ChatGPT Response

...

[Figure 2.3](#) shows that implied volatility behaves opposite to options prices! That's quite counterintuitive—is it hallucinating? As we have learned, it is better to ask ChatGPT for codes than the output of some unknown models it uses internally. To produce code in this case, we need an options pricing model, but Black-Scholes model won't do.



[Figure 2.3](#) Incorrect plot of implied volatility of put options as function of time to expiration.

User Prompt

Do you know of a theoretical option pricing model that exhibits non-trivial volatility surface with such characteristics as the “volatility smile” and the fact that short-dated options have different implied volatility than long-dated options of the same strike, while short-dated options have more negative theta than long-dated ones?

ChatGPT Response

Yes, the phenomena you're describing, including the volatility smile and the observed patterns in implied volatility and theta across different expiration dates, are addressed by several advanced option pricing models that go beyond the Black-Scholes framework. These models incorporate features to account for market imperfections and observed behaviors that the Black-Scholes model, with its simplifying assumptions (e.g., constant volatility, log-normal distribution of stock prices), cannot fully capture. Here are a few theoretical models known for their ability to reflect a more complex and realistic volatility surface:

...

2. Stochastic Volatility Models Examples: Heston model, SABR model. Key Features: In stochastic volatility models, the volatility of the underlying asset is itself a random process. These models can capture the volatility smile because they allow for changes in volatility over time, independent of the underlying asset's price movements. Characteristics: These models can explain various empirical observations, including the volatility smile and changes in implied volatilities across different expiration dates. They also allow for correlation between the asset price and its volatility, which can affect the pricing of short-dated versus long-dated options.

...

ChatGPT listed four types of options pricing model: local volatility models that use options prices to calibrate the volatility surface at different expirations and strikes; stochastic volatility models (such as Heston) that use a few

parameters to produce the general shape of the volatility surface; jump-diffusion model that allows for jumps in the price of the underlying in addition to Gaussian Brownian Motion; and the Levy Process Models that model the movement of the underlying asset as a Levy process. The Heston model seems the simplest and most appropriate if we just want to get a conceptual understanding of the term structure without getting an exact match with real data.

User Prompt

Let's pick the Heston model. Can you elaborate on that?

ChatGPT Response

The Heston model is a popular stochastic volatility model for pricing options, introduced by Steven Heston in 1993. It addresses some of the limitations of the Black-Scholes model, particularly its assumption of constant volatility. The Heston model allows the volatility of the underlying asset to be stochastic, meaning it can change over time according to a specific stochastic process. This feature enables the model to more accurately capture market phenomena such as the volatility smile.

Key Features of the Heston Model

1. **Stochastic Volatility:** The model describes the evolution of the underlying asset's price and its variance (volatility squared) using two stochastic differential equations (SDEs). The first SDE models the asset price dynamics similarly to Black-Scholes but incorporates a stochastic volatility term. The second SDE models the dynamics of the variance itself.
2. **Mean Reversion:** The variance process in the Heston model is mean-reverting, meaning it tends to return to a long-term average level over time. This is consistent with observed market behavior, where volatility spikes are usually temporary.
3. **Correlation Between Asset Returns and Volatility:** The model allows for a correlation between the asset returns and volatility changes. This correlation is crucial for accurately capturing the leverage effect, where asset prices and their volatility tend to be inversely related.

Mathematical Formulation

The dynamics of the underlying asset price (S) and its variance (ν) in the Heston model are described by the following SDEs:

$$\begin{aligned} dS_t &= \mu S_t dt + \sqrt{\nu_t} S_t dW_t^S \\ d\nu_t &= \kappa (\theta - \nu_t) dt + \sigma \sqrt{\nu_t} dW_t^\nu \end{aligned}$$

where:

- S_t is the asset price at time t .
- ν_t is the variance of the asset price at time t .
- μ is the risk-free rate of return. [Author Note: This isn't correct! It should be the deterministic (i.e., non-stochastic) instantaneous drift of asset price returns! See Gatheral (2006).]
- κ is the rate of mean reversion of the variance.
- θ is the long-term variance mean level.
- σ is the volatility of the volatility (vol of vol), which determines the volatility of the variance process.
- dW_t^S and dW_t^ν are two Wiener processes for the asset price and variance, respectively, with a correlation coefficient ρ .

Pricing Options with the Heston Model

The Heston model allows for the derivation of a semi-analytical formula for the price of European-style options. The option pricing formula involves complex integrations due to the model's stochastic nature but can be efficiently computed using numerical methods such as the fast Fourier transform (FFT).

Applications and Limitations

- **Applications:** The Heston model is widely used in financial markets for pricing European options, capturing the volatility surface more accurately than the Black-Scholes model, and is also extended to price American options and exotic options under certain conditions.
- **Limitations:** Despite its advanced features, the Heston model can be challenging to calibrate to market data due to the complexity of its dynamics and the sensitivity of its parameters. Moreover, like all models, it makes simplifications and assumptions that may not hold under all market conditions.

The Heston model's ability to incorporate stochastic volatility and mean reversion makes it a powerful tool for understanding and pricing options in a more realistic manner, reflecting the dynamic nature of financial markets.

User Prompt

Please cite any publications that show the term structure of implied volatility of an option as function of time-to-expiration.

ChatGPT Response

...

Books and Papers

1. *Option Volatility and Pricing* by Sheldon Natenberg

Natenberg - This book is a widely regarded resource in the field of options trading. Natenberg provides detailed explanations of the behavior of implied volatility and its impact on options pricing, including discussions on the term structure of volatility.

2. *Dynamic Hedging* by Nassim Nicholas Taleb

Taleb's work focuses on the practical aspects of managing option positions, including insights into the term structure of volatility and how it can be exploited through dynamic hedging strategies.

3. *The Volatility Surface: A Practitioner's Guide* by Jim Gatheral

Gatheral - Gatheral provides an in-depth analysis of the volatility surface, which includes discussions on the term structure of implied volatility and how it varies with time to expiration and strike price.

...

We pick Jim Gatheral's book (Gatheral, 2006) as our reference as it seems to cover the relevant topic with lots of charts. Unfortunately, none of which is exactly what we want: a comparison of the term structure of ATM and OTM option prices based on the Heston model. We need to instantiate a Heston model to do that. But first, we need to know what inputs are reasonable.

User Prompt

What are reasonable estimates for μ when the asset in question is the SPX?

ChatGPT Response

The parameter μ in financial models typically represents the expected return or drift rate of the underlying asset over time. For the S&P 500 Index (SPX), reasonable estimates for μ should reflect the long-term average return of the index, accounting for dividends, inflation, and the overall growth of the economy.

Historical Perspective on μ for the SPX

The S&P 500 has historically returned about 7% to 10% annually after adjusting for inflation. This range accounts for dividends reinvested and represents a long-term perspective over multiple decades. The exact figure can vary depending on the specific time period considered, how dividends are treated, and the inflation adjustment method.

...

User Prompt

What are reasonable estimates for the rate of mean reversion of the variance when the asset in question is the SPX?

ChatGPT Response

The rate of mean reversion κ in the Heston model is a key parameter that describes how quickly the variance of the asset returns reverts to its long-term mean θ . For the S&P 500 Index (SPX), as with other financial assets, the specific value of κ can vary based on market conditions, the time period analyzed, and the calibration methodology used. However, empirical studies and market practice provide some guidance on reasonable estimates for this parameter.

Reasonable Estimates for κ

- **Typical Range:** For equity indices like the SPX, κ values are often found in the range of 1 to 3 per year. This range suggests that the variance tends to revert to its long-term mean within a few months to a year.

...

User Prompt

What are reasonable estimates for long-term variance mean level for SPX?

ChatGPT Response

The long-term variance mean level θ in the Heston model represents the level to which the variance of the asset returns is expected to revert over time. For the S&P 500 Index (SPX), as with other assets, this parameter is crucial for modeling the long-term behavior of market volatility. Estimating θ involves analyzing historical volatility data and calibrating the model to current market prices of options.

Reasonable Estimates for θ

Given that θ is the long-term mean of the variance, not volatility, it's important to distinguish between the two when discussing estimates. Variance ν_t is the square of volatility Θ [Author Note: This is wrong. σ is the volatility of volatility here, not the volatility], so when practitioners refer to volatility levels, they're often discussing Θ [Author Note: same error as above], whereas θ pertains to ν_t .

- **Converting Volatility to Variance:** To think about θ in terms of familiar volatility levels, you first need to square the annualized volatility. For example, if the long-term volatility expectation for the SPX is around 20% (or 0.20 in decimal form), the corresponding variance θ would be $(0.20^2 = 0.04)$.
- **Typical Range for SPX:** Historically, the annualized volatility of the SPX has fluctuated, but a range of 15% to 30% (0.15 to 0.30 in decimal form) is common for estimating long-term volatility, depending on the time period and market conditions. Squaring these values gives a variance range of approximately 0.0225 to 0.09 for θ .

User Prompt

What are reasonable estimates for σ for SPX?

ChatGPT Response

In the context of the Heston model, σ represents the volatility of volatility (vol of vol), which quantifies the variability in the implied volatility of the underlying asset. This parameter is crucial for capturing the dynamics of the volatility surface, including the curvature and behavior of the implied volatility skew. For the S&P 500 Index (SPX), reasonable estimates for σ can vary based on historical market conditions, the calibration period, and the specific model calibration approach. [Author Note: In this description, it got right the parameters σ , which as described stands for the volatility of volatility.]

Reasonable Estimates for σ

- **Typical Range:** For the SPX, reasonable estimates for σ (volatility of volatility) often fall within the range of 0.1 to 0.4 per annum. This range reflects the observed variability in the market's implied volatility but can vary depending on the specific market environment and historical period analyzed.

...

User Prompt

What are reasonable estimates for correlation coefficient ρ for SPX?

ChatGPT Response

The correlation coefficient ρ in the context of the Heston model represents the correlation between the asset's returns and its volatility. For the S&P 500 Index (SPX), as well as other assets, this correlation tends to be negative, reflecting the common market observation that asset prices often move inversely to their volatility. This phenomenon is sometimes referred to as the "leverage effect," where market downturns are typically associated with increases in volatility.

Reasonable Estimates for ρ - Typical Range: For the SPX, reasonable estimates for the correlation coefficient ρ often fall within the range of (-0.5) to (-0.8). This range is consistent with empirical studies that have examined the relationship between stock returns and volatility.

We will generally pick the middle value of a given range of just provided estimates, except that we use the low end of the estimate of θ (long-term variance of SPX) as volatility level has dropped over the years, but a high-end estimate of σ (volatility of volatility of SPX) as that has increased over the years (see Chan (2013), [Figure 5.10](#)). We also pick the high-end estimate of μ (long-term return of SPX) because it has trended higher in recent years.

User Prompt

Let's assume $\kappa = 2$, $\theta = 0.15^2$, $\sigma = 0.4$, $\rho = -0.65$, $\mu = 0.1$, initial variance $\nu_0 = 0.15^2$. Display the Matlab code for computing the ATM option price using its Financial Toolbox.

ChatGPT Response

To compute the price of an At-The-Money (ATM) option on the SPX using the Heston model in MATLAB, you would typically use functions provided by MATLAB's Financial Toolbox. The Heston model parameters you've specified are:

- Rate of mean reversion, $\kappa = 2$
- Long-term variance mean level, $\theta = 0.15^2$
- Volatility of volatility, $\sigma = 0.4$
- Correlation coefficient, $\rho = -0.65$
- Expected return or drift rate, $mu = 0.1$
- Initial variance, $\nu_0 = 0.15^2$

Here's a conceptual MATLAB script to calculate the price of an ATM option using the Heston model. Note that MATLAB's Financial Toolbox may require specific functions for option pricing, such as `optByHestonNI` (using numerical integration) to compute European option prices.

```

% Heston model parameters
kappa = 2;
theta = 0.15^2;
sigma = 0.4;
rho = -0.65;
v0 = 0.15^2;
r = 0.1; % Risk-free rate, assuming it's the same as
mu for simplicity [Author Note: this is wrong! r
should be 0.03.]

% Define the asset path
S0 = 100; % Example initial SPX level, adjust as
needed [Author Note: should be about 5,000 as of
February, 2024.]
K = S0; % ATM option
T = 1; % Time to maturity in years

% Define option type
OptType = {'call', 'put'}; % Choose as needed

% Pricing the option using Heston model
for i = 1:length(OptType)
    price = optByHestonNI(kappa, theta, sigma, rho,
v0, r, T, S0, K, 'OptSpec', OptType{i});
    fprintf('%s option price using Heston model:
%f\n', OptType{i}, price);
end

```

This script uses the `optByHestonNI` function to calculate option prices based on the Heston model parameters. The `OptSpec` option allows you to specify the type of option ('call' or 'put'). Adjust `S0`, `K`, and `T` as needed to reflect the current price of the SPX, the strike price for the ATM option, and the desired time to maturity, respectively.

Please ensure you have the Financial Toolbox installed and consult the MATLAB documentation for the latest functions and syntax, as function names and parameters

can change between releases. This example assumes a simplified scenario for illustrative purposes.

We don't completely trust the codes generated, but we are glad that ChatGPT pointed to the Matlab Financial Instruments Toolbox (though not the Financial Toolbox that it suggested) function *optByHestonNI*. We will reference the official documentation at

[https://www.mathworks.com/help/fininst/optbyhestonni.html?
s_tid=doc_ta#d126e204312](https://www.mathworks.com/help/fininst/optbyhestonni.html?s_tid=doc_ta#d126e204312) to clean up this script. Our final version is saved in GitHub as HestonModelPutPlot.m. The plot is displayed in [Figure 2.4](#). The one for calls is saved in HestonModelCallPlot.m and displayed in [Figure 2.5](#).

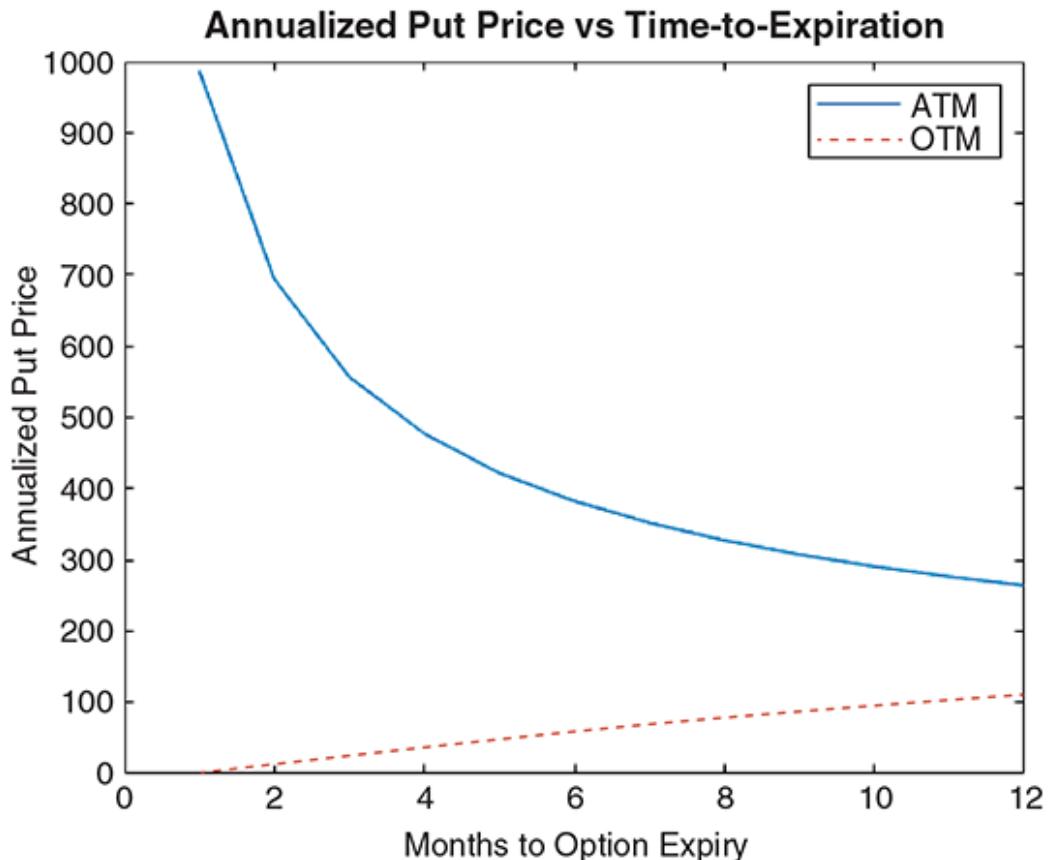


Figure 2.4 Annualized put option prices as function of time to expiration based on the Heston model.

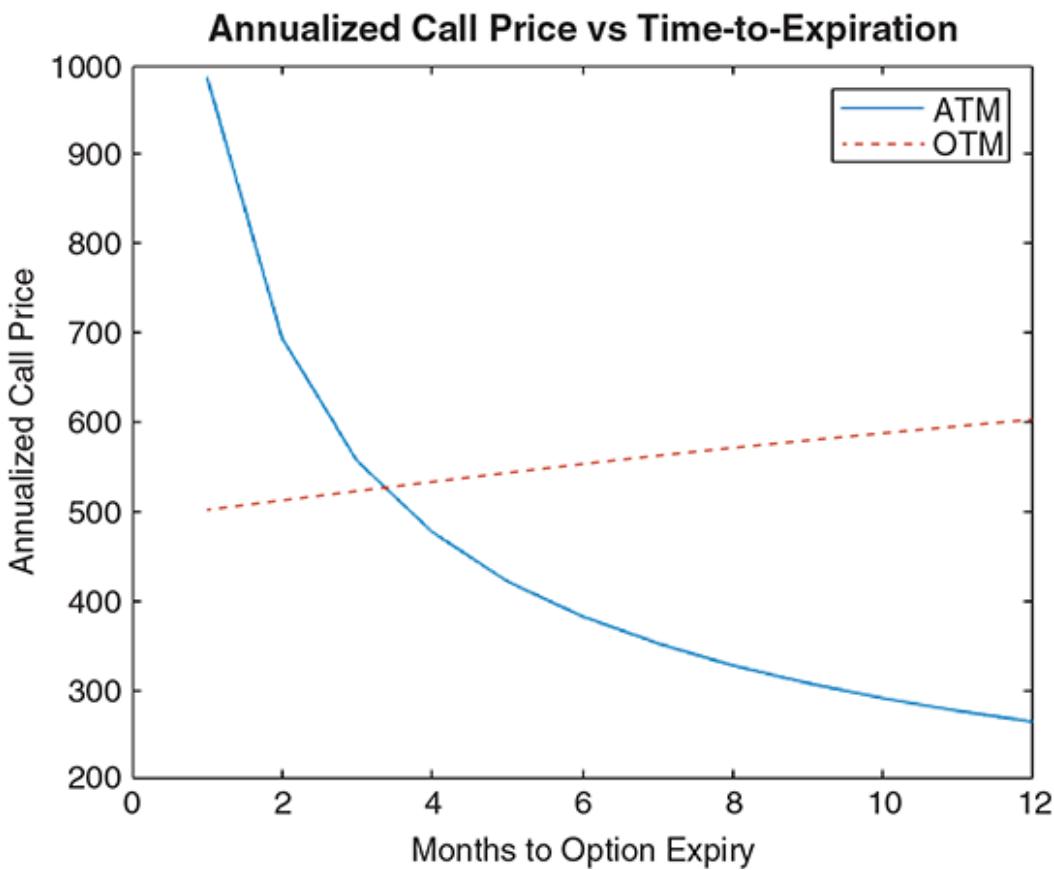


Figure 2.5 Annualized call option prices as function of time to expiration based on the Heston model.

* * *

Exercise: Ask ChatGPT for an equivalent Python version.

* * *

Based on these curves, it seems that we can short a long-dated OTM option and hedge that by buying a short-dated OTM option that is continuously being rolled forward before its expiration, unless it becomes an ATM option at some point due to SPX level changes. At that point, we should liquidate the positions because the cost of hedging the long-dated option will outweigh the annualized time value of that option. The opposite strategy, buying a long-dated ATM option and shorting a short-dated option won't work because when the short-dated option expires, the long-dated option will most likely not be ATM anymore and the noted term structure

won't apply (unless we expect a catalyst in the market that might produce significant move in the underlying after the first option expires, such that the long-dated option becomes at the money). There is a wide range of SPX levels that make the options OTM, but a narrow range that make them ATM.

What about the implied volatility term structure? Are they the same or opposite to the annualized option value as ChatGPT suggested earlier? But wait, recall that in the Heston model, volatility is stochastic and changes constantly. What does the implied volatility term structure (also known as the volatility surface) even mean?

User Prompt

Given that Heston model uses a stochastic volatility model as the basis, does it even make sense to ask for the implied volatility of an option since it is stochastic over time?

ChatGPT Response

Your question touches on a nuanced aspect of option pricing theory and the interpretation of implied volatility within the context of stochastic volatility models like the Heston model.

Implied Volatility in the Black-Scholes Framework In the Black-Scholes model, implied volatility is a constant value that, when input into the model, produces an option price matching the market price. It represents a single, average measure of expected volatility over the life of the option. This model assumes volatility is known and constant, which simplifies the calculation but does not capture the complexity of real market conditions.

Stochastic Volatility Models Stochastic volatility models, including the Heston model, recognize that volatility is not constant but varies over time in a random manner. These models provide a more realistic framework for option pricing by incorporating a stochastic process for volatility.

Implied Volatility within Stochastic Volatility Models Even though the Heston model and other stochastic volatility models use a dynamic view of volatility, practitioners still compute implied volatilities from these models for several reasons:

1. **Market Benchmark:** Implied volatility serves as a standardized benchmark, allowing traders and analysts to compare option prices across different strikes and maturities in a consistent way, even within the context of models that recognize volatility's dynamic nature.
2. **Model Calibration:** Implied volatility is used to calibrate stochastic volatility models to market data.

By adjusting model parameters to match the observed market implied volatilities across a range of strikes and maturities, practitioners ensure that the model reflects current market conditions as closely as possible.

3. **Volatility Surface:** The concept of implied volatility is central to constructing the volatility surface, which plots implied volatilities across different strikes and maturities. Even in the context of stochastic volatility models, the volatility surface provides valuable insights into market expectations and sentiment.
4. **Communication Tool:** Despite its limitations, implied volatility remains a useful tool for communicating market conditions and risk perceptions among traders, analysts, and other market participants.

Conclusion While it might seem counterintuitive to discuss implied volatility in the context of a model that inherently acknowledges volatility's stochastic nature, doing so allows for the practical application of these models in real-world trading and risk management. Implied volatility, derived from market prices of options, bridges the gap between theoretical models and market practice, serving as a common language for market participants and a tool for model calibration.

Unlike what academic theoreticians often do, we will not bother with plotting the volatility surface. It isn't empirically observable, and we can't make money from it directly. As traders, we prefer things that we can actually trade.

2.5 Conclusion

We asked ChatGPT to create the backtest codes for a long-short equity trading strategy using the Fama-French factors,

and a hedged VIX futures carry strategy. We also asked ChatGPT to suggest an options calendar spread strategy and explore the term structure of options prices. Less demandingly, we asked Copilot to find out if someone has published a deep reinforcement learning technique for portfolio optimization. From these experiments, we conclude the following:

1. We can't really use ChatGPT to automate the entire process. What is needed is a combination of human intelligence, knowledge, experience and the LLM's suggestions. The situation is quite analogous to self-driving cars. Fully autonomous vehicles are rarely found except in a few experimental urban areas, but assisted driving technology has been embedded in almost all new cars for years already (Chan, 2022). We are pretty far from AGI (artificial general intelligence).
2. The responses from ChatGPT are often not reproducible: the same prompt can result in different responses. This is difficult for most engineering projects that require strict reproducibility.
3. It may be more efficient to use ChatGPT as a coach to speed up your own coding tasks than to ask it to write good codes every time you need one.

These conclusions are not dissimilar to those of Simon Willison (Django and Datasette creator) as reported in Claburn (2024). We need a “human in the loop” for the most serious applications of AI.

That is not to say that there aren't occasional flashes of brilliance or evidence of conceptual understanding of quantitative finance, such as when ChatGPT understands that a trading strategy that holds a position for N days may have up to N positions on any single day, or whether subtracting the risk-free rate from the gross returns is always necessary when computing the Sharpe ratio. These “sparks of AGI” are

well-documented in the paper by Bubeck et al. (2023). But such brilliance vanished when we asked ChatGPT to analyze the VIX paper and produce pseudocodes for the carefully described trading strategy contained therein. Even after pointing ChatGPT to the specific paragraph with the relevant information, it failed to capture the entire trading strategy. Perhaps it is just a limitation of how long a paper GPT-4o can handle at this time, but we can only say that the state-of-the-art available to a professional trader is simply not adequate as of this writing. As our effort to locate a paper on applying deep reinforcement learning to portfolio optimization shows, Copilot isn't very good at locating information that meet our specific professional needs. We can perhaps improve that ability by fine-tuning it on a quantitative finance database, or with particularly expert prompt engineering. For example, it has been known that "chain-of-thought prompting," i.e., "asking language models to generate step-by-step solutions enabled the models to solve problems that had previously seemed beyond their reach" (Brukbaker, 2024). But if such highly technical human skills are required of the user, it can hardly be called AGI. Of course, with each new generation of GPT (we are using GPT-4o as of this writing), its capability will improve. What ChatGPT and other Gen AI tools are good at is providing a first draft of something that may be partially correct. As such it is useful as a junior collaborator, motivating the senior human collaborator to try new things. Often, the first draft is the most difficult (a.k.a., writer's block). Revisions and improvements are easier for humans. So, if you ask us: "After writing these two chapters, would you still use Gen AI to help with investment strategies discovery and backtest?" The answer is yes.

You may wonder, given the state of Gen AI, why we should dive deeper into Gen AI later on. The reason is this: most of the tasks we have given Gen AI so far are those that have a clear answer, and human experts can perform them equally well if not better. Of course, not all of us are experts in every

subject, and it saves us time to get AI to do what we know how to do if it can do it well. That was the reason we explored no-code AI. However, there are tasks where there are no clear answers, and even human experts may disagree on what the best answers are. In medical diagnostics, we may ask, “Given the symptoms, what is the likely disease affecting the patient?” In asset management, “Will NVDA increase its earnings next quarter?”, “Will oil price go down next month?”, or “Is what the Fed chair just said in the press conference positive or negative for the S&P 500?” (The Gen AI answer to the last question is actually an example later in this book.) These are all questions with no clear answers, except in hindsight. This is the realm of discretionary or fundamental trading. Because machines and humans are on more equal footing for these questions, we may find that Gen AI can add more value. We want to find out if building a “George Soros on a chip” is possible.

The following chapter will set the stage for this ambitious project. This will inevitably involve usage of Generative AI APIs.

* * *

2.6 Exercises

1. Ask ChatGPT to create the backtest code for the paper Zhang et al. (2019) on Deep Reinforcement Learning for Trading. Does ChatGPT use Tensorflow? Can you create unit tests to see if it produces the codes correctly?
2. What is Retrieval Augmented Generation (RAG), and how would it correct some of the hallucinations the LLMs produce?
3. Ask the latest version of ChatGPT to see if it can now summarize the VIX strategy from the cited paper correctly.

4. Ask ChatGPT to create the code to compute the expected profit of an iron condor for SPX options with a holding period of a month, assuming normal distribution of returns and reasonable volatility estimates. Watch how it does numerical integration with aplomb! Bonus question: ask it to generate the code to compute its Sharpe ratio too.

Appendix 2A.1 Computing Next-day's Return

Here is the Python program that computes the next-day (or in general, next-N-day) returns. These returns will be used as "labels" (or the Y variable) for the Fama-French regression model.

```

import pandas as pd

def calculate_N_day_ahead_returns(file_path,
output_file_path, N=1):
    try:
        # Load the data
        stock_prices = pd.read_csv(file_path)

        # Rename column names
        stock_prices=stock_prices.rename(columns=
{'tday':'Date', 'syms':'Stock_ID', 'mid': 'Price'})

        # Convert the 'Date' column from integer
        # YYYYMMDD format to datetime format
        stock_prices['Date'] =
pd.to_datetime(stock_prices['Date'], format='%Y%m%d')

        # Sort the data by 'Stock_ID' and 'Date'
        stock_prices.sort_values(by=['Stock_ID',
'Date'], inplace=True)

        # Shift the prices by -N days to align with the
        # price N days ahead
        stock_prices['Price_in_N_days'] =
stock_prices.groupby('Stock_ID')['Price'].shift(-N)

        # Calculate the N-day returns
        stock_prices[str(N)+'_Day_Returns'] =
(stock_prices['Price_in_N_days'] -
stock_prices['Price']) / stock_prices['Price']

        # Dropping the column used for calculation
        stock_prices.drop(columns=['Price_in_N_days'],
inplace=True)

        # Save the results to a new CSV file
        stock_prices.to_csv(output_file_path,
index=False)

    return "Successfully computed 30-day returns
and saved to " + output_file_path

```

```
except Exception as e:  
    return f"An error occurred: {e}"  
  
# Example usage  
file_path = 'CRSP_mid.csv' # Replace with the actual  
file path  
output_file_path = 'CRSP_nextDayReturns.csv' # Replace  
with the desired output file path  
result = calculate_N_day_returns(file_path,  
output_file_path)  
print(result)
```

Appendix 2A.2 Uploading the Fama-French Factors

The Fama-French factors file was downloaded from https://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html. We just need to create a Python Pandas dataframe to hold the data with proper dates.

```

import pandas as pd

# Path to the file (replace with your file path)
file_path = 'F-F_Research_Data_Factors_daily.CSV'

# First attempt: Skipping initial rows that might
# contain descriptions or headers
ff_data_1 = pd.read_csv(file_path, skiprows=3)

# Assuming successful read, filtering out non-date
# entries
ff_data_filtered = ff_data_1[ff_data_1['Unnamed: 0'].apply(lambda x: str(x).isdigit())]

# Rename first column to 'Date'
ff_data_filtered.rename(columns=
{ff_data_filtered.columns[0]: 'Date'}, inplace=True)

# Change first column to datetime type
ff_data_filtered ['Date'] =
pd.to_datetime(ff_data_filtered['Date'])

# Example query: Finding the SMB factor on a specific
# date
smb_factor = ff_data_filtered[ff_data_filtered['Date']
== '20151106']['SMB'].iloc[0]
# This returns the correct numerical value of 0.87

```

Including a “unit-test” like the last line of this code is important in any program, but especially for those generated by ChatGPT.

Now, we need to combine the daily returns data and the factor data correctly, using the former as the “label” or “Y” variable in our regression fit and the latter as “features” or “X” variable. Prompting ChatGPT for that, the response more or less looks like Appendix 2.3, which follows, with some human editing.

Appendix 2A.3 Combining Fama-French Factors with Next-day's Returns

We have created the dataframe that contained the Fama-French factors as features for our training data. Now we need to combine them with the labels, which are the next-day's returns, in order to form the complete dataframe for training and testing a linear regression model.

```
import pandas as pd
from sklearn.linear_model import LinearRegression

# Assuming ff_data_filtered and next_day_returns_data
# are your loaded DataFrames

# %% Load and parse daily returns of stocks
next_day_returns_data =
pd.read_csv('CRSP_nextDayReturns.csv')
next_day_returns_data['Date'] =
pd.to_datetime(next_day_returns_data['Date'],
format='%Y-%m-%d')
next_day_returns_data['Date'] =
pd.to_datetime(next_day_returns_data['Date'],
format='%Y%m%d')

# %% Load and parse Fama-French data
file_path = 'F-F_Research_Data_Factors_daily.CSV'
# First attempt: Skipping initial rows that might
# contain descriptions or headers
ff_data_1 = pd.read_csv(file_path, skiprows=3)

# Assuming successful read, filtering out non-date
# entries
ff_data_filtered = ff_data_1[ff_data_1['Unnamed: 0'].apply(lambda x: str(x).isdigit())]

# Rename first column to 'Date'
ff_data_filtered.rename(columns=
{ff_data_filtered.columns[0]: 'Date'}, inplace=True)

# Change first column to datetime type
ff_data_filtered ['Date'] =
pd.to_datetime(ff_data_filtered['Date'])

# Example query: Finding the SMB factor on a specific
# date
smb_factor = ff_data_filtered[ff_data_filtered['Date']
== '20151106'][['SMB']].iloc[0]
# This returns the correct numerical value of 0.87
```

```

# Merge the datasets on the 'Date' column
combined_data = pd.merge(ff_data_filtered,
next_day_returns_data, on='Date')

# Unit test
# Function to perform the unit test
def test_factors_on_date(data, date, expected_smb,
expected_hml):
    # Extracting the row for the given date
    data_on_date = data[data['Date'] == date]

    # Checking if the date exists in the data
    if data_on_date.empty:
        return "Date not found in the data."

    # Extracting the SMB and HML values
    smb_value = data_on_date['SMB'].iloc[0]
    hml_value = data_on_date['HML'].iloc[0]

    # Testing the conditions
    smb_test = abs(smb_value - expected_smb) < 0.01 # Approximate check
    hml_test = abs(hml_value - expected_hml) < 0.01 # Approximate check

    return {
        "SMB Test Passed": smb_test,
        "HML Test Passed": hml_test,
        "Actual SMB Value": smb_value,
        "Actual HML Value": hml_value
    }

# Performing the test for 20151106
test_results = test_factors_on_date(combined_data,
'20100423', 0.37, 0.62)
test_results

# Function to perform the unit test for MSFT 1 day
return
def test_msft_return_on_date(data, date,
expected_return):

```

```

# Extracting the row for the given date
data_on_date = data[(data['Date'] == date) &
(data['Stock_ID']=='MSFT') ]
print(data_on_date)

# Checking if the date exists in the data
if data_on_date.empty:
    return "Date not found in the data."

# Extracting the MSFT 1 day return value
msft_return_value =
data_on_date['1_Day_Returns'].iloc[0]

# Testing the condition
return_test = abs(msft_return_value -
expected_return) < 0.01 # Approximate check

return {
    "MSFT Return Test Passed": return_test,
    "Actual MSFT 1 Day Return Value":
msft_return_value
}

test_msft_return_on_date(combined_data, '20100423',
0.00508690122)
{'MSFT Return Test Passed': True,
'Actual MSFT 1 Day Return Value': 0.0035501048894635}

test_msft_return_on_date(combined_data,
'20081021', -0.078)
{'MSFT Return Test Passed': True,
'Actual MSFT 1 Day Return Value': -0.0777468408652821}

```

It passed all the unit tests.

Chapter 3

Whirlwind Tour of ML in Asset Management

You may not have realized it, but you have probably been applying machine learning to asset management for many years now. Maybe you have used linear regression to find the beta of a portfolio against a market index? Maybe you have tried Principal Component Analysis as an alternative to the Fama-French factor model described in [Chapter 2](#)? Though they may seem retro, these are all examples of machine learning. On the other hand, you may think that machine learning in asset management is all about using a robot to predict stock returns. As we will explain in the section on supervised learning, that may be the least successful application of AI in finance.

In this chapter, we will take a whirlwind tour of machine learning in asset management, ending with how Generative AI can help beyond the no-code usages we have already demonstrated in the previous two chapters. (We will leave the nuances of Deep Learning and Generative AI to the rest of the book.) This tour will include the following:

- Unsupervised learning: clustering with application to capital allocation via Hierarchical Risk Parity (HRP), Principal Component Analysis (PCA), Cluster-based Feature Selection (cMDA), and Hidden Markov Model (HMM)
- Supervised learning: linear and logistic regressions with regularizations, tree-based models, and neural networks. Various performance metrics, hyperparameter optimization and (cross-)validation. Applications to Corrective AI (CAI) and Conditional Parameter/Portfolio

Optimization (CPO), but emphatically not asset return predictions

- Comments on Recurrent Neural Networks and Deep Reinforcement Learning
- Data and features engineering; features importance ranking and selection; application of Generative AI to features engineering

The intention of this chapter is to highlight some of the most important concepts in machine learning that are relevant to asset managers but not delve into details in any one of them. You can find much better books if you want details. For foundational machine learning concepts, see Murphy (2012) and Bishop (2006). For machine learning for asset managers, see López de Prado (2018, 2020). For neural networks, see Goodfellow et al. (2016).

We will follow this notational convention: \mathbf{X} is a matrix, \mathbf{x} is a vector, x is a scalar. In particular, \mathbf{x} is a feature vector, y and \mathbf{y} are inferred (predicted) scalar and vector label respectively, while \hat{y} and $\hat{\mathbf{y}}$ are the actual scalar and vector label respectively. \mathbf{X} often denotes a data or design matrix with n rows of samples and m columns of features. (We shall explain what features, labels, and design matrix mean in the supervised learning section.)

3.1 Unsupervised Learning

We sometimes expect a machine to do things that we know how to do, if only with hindsight. Deciding whether to buy or sell an asset and hold it for one day is one such example, because with historical data we would immediately know whether that was a good decision or not. Other times, we want a machine to do things that we couldn't even tell if the solution provided is right or is the best available. Deciding what stocks should be grouped into clusters for the purpose

of portfolio optimization via Hierarchical Risk Parity (HRP) is an example. We don't have historical data showing what the optimal clusters are, nor an analytical method to prove that a given cluster is optimal. Finding these clusters, historical or current, is exactly what we want to apply unsupervised machine learning to. Similarly, if we want to find out what the common factors that drive asset returns are based on the historical returns data alone via Principal Component Analysis (PCA), we can't use any observable variables except the returns themselves. Finally, we may want to use a Hidden Markov Model (HMM) to model the regime changes in the financial market when we build probabilistic models of asset returns. As the adjective *Hidden* suggests, the discrete Markov states that we hypothesize aren't observable, and we must infer from the "emissions"—such as the returns of a stock—in order to estimate the parameters of the HMM. All these tasks fall under the category of unsupervised learning, which is all about capturing patterns in unlabeled data. We will discuss each of these unsupervised learning techniques to illustrate the general concept.

3.1.1 Hierarchical Risk Parity (HRP)

Portfolio optimization, or how much money we should allocate to each component asset of a portfolio, is an age-old finance problem. Harry Markowitz earned a Nobel Prize in Economics for inventing the mean-variance optimization approach in 1952 (see [Section 1.4](#) on finding the Efficient Frontier of a portfolio.) However, it is well-known that mean-variance optimization works only in hindsight—it doesn't usually produce an optimal portfolio in the future. That's because the mean-variance optimal portfolio is highly sensitive to the input returns. Ang (2014) illustrated this well in a section baldly titled "Garbage In, Garbage Out" (p. 95). A portfolio that is allocated to five developed countries' equity indices will change its allocation to the US index from -9% to

41% if we change the estimate of its annual mean return from 10.3% to 13%. If you tell me you can predict the US equity index to within 2.7% next year, we have a Brooklyn Bridge to sell you. Hence many asset managers despair (including the mighty Bridgewater Associates) and adopt a seemingly simplistic approach instead: just allocate equal “risk” to each asset, with risk measured simply by the volatility of its returns. This is called the *risk parity* allocation method. If asset A has risk of 20% and asset B has risk of 40%, then we allocate capital to A:B in the ratio 2:1 (i.e., inversely proportional to the volatility). This simplistic approach actually works better in practice than the sophisticated mean-variance optimization, but there are some issues with this approach too. Risk parity assumes that all the asset returns are statistically independent (and thus uncorrelated). If the two assets are in fact highly correlated (think F vs GM stocks) and have similar volatility, risk parity will allocate to them on a 1:1 basis approximately, but together the two assets contribute to the portfolio $\sqrt{2}$ times the risk of two uncorrelated assets. Hierarchical risk parity (López de Prado, 2020) is a method to address that. The basic idea is to cluster the assets within a portfolio based on their correlations. Assets that are highly correlated are going into the same cluster. The outline of the algorithm is this:

1. Apply *hierarchical clustering* (Hastie et al., 2009) to the covariance matrix to find clusters of highly correlated assets. (There are various techniques to decide on the optimal number of clusters, but that is of secondary importance. You might regard this number as a hyperparameter to be optimized, a concept to be discussed in the supervised learning section.)
2. Apply your favorite portfolio optimization method to the assets within each cluster to find the intra-cluster allocations. Since all assets in the cluster are similar, we can just apply the usual risk parity allocation which will

not overweigh any asset's risk in this case. (Note that correlation ~ 1 doesn't mean asset volatilities are the same within the cluster.) Call this allocation $w_c(s)$ for asset s in cluster c , where $\sum_s w_c(s) = 1$.

3. Apply your favorite portfolio optimization method to each cluster's return to find the capital allocation across different clusters. As the correlations between these clusters are supposed to be very low, the covariance matrix of the cluster returns is almost diagonal. Hence mean-variance optimization applied here won't suffer the instability and sensitivity to inputs. In fact, if $W(c)$ is the weight given to cluster c , with $\sum_c W(c) = 1$, and m_c and σ_c are the mean and standard deviation of returns of cluster c , then $W(c) \sim m_c / \sigma_c^2$ which many readers will recognize as the allocation based on the Kelly formula (Chan, 2017).

4. The final allocation to assets in cluster c is $\omega(s) = W(c) * w_c(s)$. Note that if all the clusters have similar returns and volatilities, and all the assets within a cluster c have similar volatilities, then $\omega(s) = \frac{1}{n(c) \bullet N}$, where $n(c)$ is the size of cluster c , and N is the number of clusters. This is, of course, very different from a naive equal allocation which would give $\omega(s) = \frac{1}{\sum_c n(c)}$.

* * *

Exercise: Ask ChatGPT to plot the dendrogram of these five stocks based on the correlations of their daily returns in the last 5 years: F, GM, TSLA, BAC, JPM. Stop at a maximum of three clusters. Does it look like [Figure 3.1](#)?

* * *

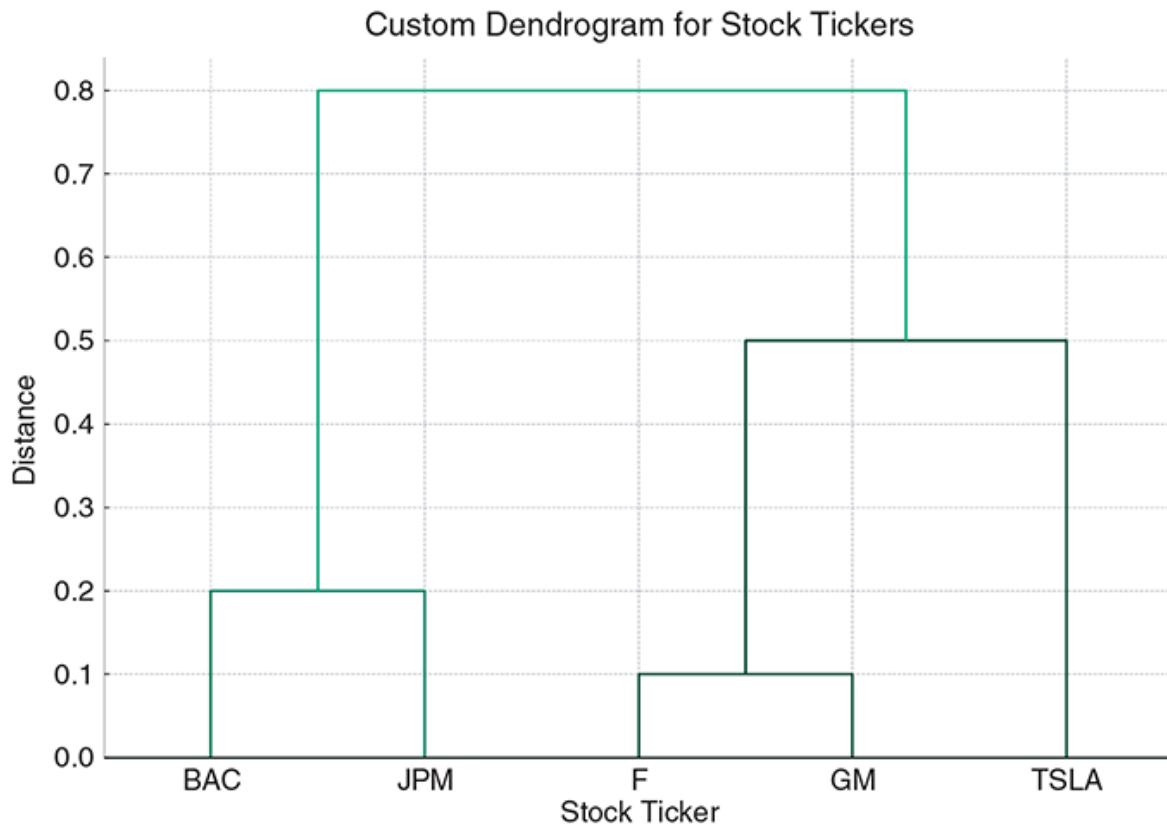


Figure 3.1 Dendrogram of five stocks based on the correlations of their daily returns in the last 5 years.

3.1.2 Principal Component Analysis (PCA)

In [Chapter 2](#) we saw how we can use the Fama-French factors to predict returns of stocks. These factors are observable: e.g., anybody can compute the SMB factor by gathering stocks' returns and market capitalizations. PCA is a method of finding factors that are “statistical”—wholly constructed from the stocks' returns data alone—and use these factors to analyze or predict these same returns. Hence it is another example of unsupervised learning. PCA factors (more precisely, factor exposures or loadings) are created by either finding the top eigenvectors (ranked by their eigenvalues) of the covariance matrix of returns, or by finding the top “right singular vectors” by applying singular value decomposition (SVD) directly to the data matrix of

returns (e.g., rows of daily returns vectors of various stocks). For more details, see Chan (2021), Example 7.1. By “top,” we meant the eigenvectors associated with the largest eigenvalues, or the right singular vectors associated with the largest singular values. The cutoff value of what makes the cut to be top is a detail that you can find discussed in López de Prado (2018) and can also be considered a hyperparameter to be optimized when results are used downstream in an investment process.

In addition to their use as statistical factors that drive asset returns, PCA or more generally SVD can often be used for denoising or feature selection in machine learning applications. Specifically, if we have a set of input features that can be used in supervised learning (see the next section), and these features are highly correlated, we can apply PCA to find the top “orthogonal features” that are uncorrelated and *significant*. Significant means that these features are top-ranked in statistical importance relative to all the other features (which doesn’t necessarily mean they are statistically significant in the absolute sense.) For more details, read López de Prado (2018). In [Chapter 6](#), we will discuss additional unsupervised learning approaches, such as probabilistic principal component analysis (PPCA), which is the probabilistic version of PCA, as well as Factor Analysis (FA), which can be seen as an extension of PPCA.

* * *

Exercise: Ask ChatGPT to generate two random series with 100 samples each, and with covariance matrix $\begin{bmatrix} 1 & 0.5 \\ 0.5 & 2 \end{bmatrix}$.

Create a scatter plot, and superimpose the eigenvectors based on PCA on it. Does it look like [Figure 3.2](#)?

* * *

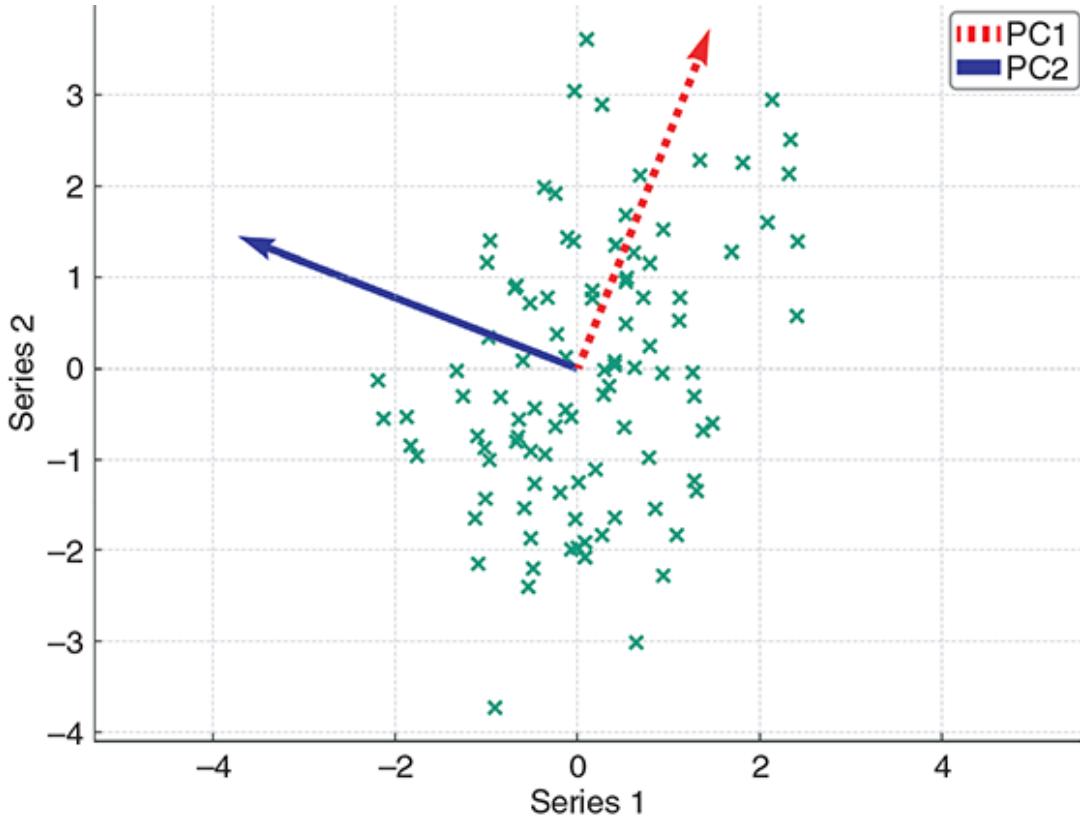


Figure 3.2 Principal components of two correlated series.

Speaking of features selection and importance ranking, there is another way unsupervised learning can help to be described next.

3.1.3 Cluster-based Feature Selection (cMDA)

As we will discuss in the context of supervised learning, feature importance ranking and selection is a method to *regularize* a machine learning process so that it doesn't learn from statistically insignificant and noisy input features.

Feature importance ranking also improves interpretability of a machine learning model, as a human user can more readily understand how the model derives its predictive power with a small number of important features instead of a large number of possibly irrelevant inputs. Whether a feature is significant

(important) of course depends on the label (target variable) during the training phase in supervised learning. We will detail a few feature selection methods when we come to supervised learning, but there is a common problem with many of those methods. If many of the features are correlated or in general highly statistically dependent with each other, these feature selection methods often randomly select one but not the others as the important feature. Two highly similar features may receive two very different importance ranks. That is arbitrary and not conducive to interpretability, as a cluster of similar features often tells a better story than an individual feature. López de Prado (2020) described a cluster-based feature selection method to address this issue. In Man and Chan (2021b), we applied hierarchical clustering to find these clusters and presented evidence that applying features importance ranking and selection based on clusters instead of individual features yield better predictive outcomes and better interpretability. From that paper, we clustered some features that are used for predicting the returns of SPX and found two big clusters that can be readily interpreted as fundamental vs technical features, as shown in [Table 3.1](#).

TABLE 3.1

Results of cluster-based features importance ranking based on their importance in predicting SPX returns.

Topic	Cluster Scores	Features
Fundamental	0.667	d/p, d/y, e/p, b/m, ntis, tbl, lty, dfy, dfr, infl
Technical	0.333	d/e, svar, ltr, tms

3.1.4 Hidden Markov Model (HMM)

Hidden Markov Model is a popular technique (Chan, 2017) of quantitative investors, because the connection between the “hidden states” and market regimes is so seductive. Who wouldn’t want to model the transitions between market regimes as transitions on a Markov chain? The problem is that if the market regimes are observable and well-defined, we wouldn’t need a *hidden* Markov model. For example, you can define the start of a “bear” market regime as when the price drops 20% or more below the high watermark, and the end of this bear market as when the price rises 20% or more above the low watermark since the beginning of the bear market. (The end of a bear market is the beginning a bull market, and vice versa.) However, predicting these regimes would be a supervised learning problem, not a problem for HMM. If we can’t observe the market regimes, all HMM is doing is to introduce more parameters into a probabilistic model of returns where, for example, a simple static mixture of Gaussians, known as a Gaussian Mixture Model (GMM), doesn’t fit the data well enough. The correspondence with hidden market regimes is aspirational, or more bluntly, fictional. Of course, the more parameters you have, the better the fit for the training set. If two hidden Markov states don’t fit the observed returns well enough, how about two hundred? The question is: does the HMM predict the out-of-sample returns well? Here the evidence is weak. Chan (2017) found that it fails miserably for SPY, but Catello et al. (2023) found that it worked on a few stocks for a few time periods. The jury is still out. In [Chapter 6](#), we will delve deeper into GMM and provide an illustrative example of market regime detection.

3.2 Supervised Learning

The bulk of investment applications of machine learning falls under the category of supervised learning. Predicting stock returns using technical indicators? Finding the beta of a portfolio relative to a market index? Predicting whether an investment decision will be profitable? All these are familiar examples of supervised learning.

Supervised learning means that we know the truth, at least historically, and we want to train a machine learning model with that knowledge so that it can make good inferences (predictions) on data it hasn't seen yet. The truth is variously called label, target, variate, dependent variable, or simply y . To train the machine learning model, we also need features, a.k.a., predictors, regressors, covariates, factors, independent variables, or simply x . When the label is discrete (e.g., a stock going up or down), the problem is called classification. When the label is continuous (e.g., a stock will have 11.2% return next year), the problem is called regression.

We wrote as if predicting stock returns is the best use case of supervised learning. Actually, it is the worst. When we are learning from historical data to predict something, it is imperative that what we try to predict won't change the target variable's behavior because the prediction is successful. For example, predicting the next earthquake won't change when it will happen. But when you build a stock prediction model that is based on learning from past data, yours won't be the only such predictive model since the data is public. Arbitrage activities will cause the returns prediction to gradually diminish in accuracy—that's known as alpha decay in the business. Taking action based on the prediction has effectively changed the behavior of the variable being predicted. This effect can be seen as a feedback loop, where our actions change the system being observed and predicted, is not considered in the standard formulation of supervised

learning problems but is instead the subject of control systems or reinforcement learning (RL), which will be described later in this chapter. There are much more effective use cases of machine learning in asset management: Corrective AI (Belov et al., 2023) and Conditional Portfolio Optimization (Chan et al., 2023). The former applies classification and the latter applies regression. Both of them are more effective than directly predicting asset returns because what they try to predict are not subject to arbitrage. We will explain why.

Corrective AI (CAI) was called metalabeling by López de Prado (2018). It means that we treat a conventional quantitative investment system as an “expert system,” with investment rules defined by humans based on their market/mathematical insights and with a small number of inputs. Then we apply machine learning as a “corrective layer” using supervised learning to determine the “probability of profit” (PoP) of each investment decision or each position over some time period. If the PoP is above a threshold, we accept the decision/position. If not, we reject the decision/position and remain “flat” or “risk-free.” The labels for CAI are profitable (the positive class, with label 1) vs non-profitable (the negative class, with label 0). For an example trading application of CAI, see Belov et al. (2023). This use of ML prediction isn’t subject to arbitrage because no one else is supposed to know your expert system’s next prediction of asset returns. So even if everyone can predict with high accuracy whether your expert system will be profitable, they can’t take any action that may change the outcome.

Conditional Portfolio Optimization (CPO) is the invention of [Predictnow.ai](https://predictnow.ai). It uses ML to predict the performances (e.g., returns, Sharpe ratio, max drawdown) of a portfolio under various asset allocations and rank such performances so that we can pick the best asset allocation with the highest predicted performance. It is a novel portfolio optimization

method that can use big data as input to adapt to market regime changes. For a more complete exposition of CPO and its many successes, see Chan et al. (2023). CPO also isn't subject to arbitrage because few investors have the exact same optimization objective on the same exact portfolio. The effectiveness of the optimization also doesn't depend on accurately predicting individual asset returns, nor does it depend on even on accurately predicting portfolio returns, making this method highly fault-tolerant.

There are numerous supervised learning techniques, ranging from the simple and well known, to the complex and arcane. Most of these techniques can be applied to both classification and regression, with small adaptations. Many of these techniques can be fine-tuned by optimizing their "hyperparameters." Some techniques to improve the performance of a supervised learning system on unseen (out-of-sample, test) data fall into a category called regularization, which are a rich and varied set of methods. We will highlight the various supervised learning techniques as well as methods of hyperparameter optimization and regularizations next.

3.2.1 Linear and Logistic Regressions

We trust that anyone who studied finance or statistics have seen the regression fit $y = \beta \cdot \mathbf{x}$, where \cdot indicates a dot (scalar) product of β (vector of regression coefficients) and \mathbf{x} (features vector) giving the scalar label y . This works for inferring (predicting) one label at a time. To train this model —i.e., finding the best values of β that can make accurate inferences (predictions)—we need a lot of historical data with (features, label) pairs of (\mathbf{x}, y) forming the data matrix \mathbf{X} (each row represents one sample data with many columns of different features, augmented with a column of ones to account for possible constant offset) and the label vector \mathbf{y} (a

column vector of the corresponding univariate labels.) \mathbf{X} is also called a “design” matrix. We can use ordinary least square (OLS) fit to give us $\boldsymbol{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$. See Murphy (2012).

* * *

Exercise: Ask ChatGPT to generate a hypothetical design matrix of four samples of three features. Name these features “Oil 30d return,” “GDP Growth,” and “HML.” All the features should be normally distributed with mean and STD of 0.5%. Does the matrix look like [Table 3.2](#)?

* * *

TABLE 3.2

Hypothetical design matrix for three features with four samples.

Sample	Oil 30d return	GDP Growth	HML
0	0.013820	0.007001	0.009894
1	0.016204	0.014338	0.000114
2	0.009750	0.004243	0.004484
3	0.007053	0.005720	0.012271

Another way to optimize the parameters of the linear regression model is to maximize the log likelihood of the training data via *maximum likelihood estimation* (MLE). In other words, we maximize the joint log likelihood $\sum_i \log(P(y_i))$ of obtaining a set of observations y_i by tweaking the values of $\boldsymbol{\beta}$ given the values of \mathbf{x}_i . (Note we use index i as it is the conventional row index for a design matrix.) It turns out that if we assume that \mathbf{y} has a Gaussian distribution, maximizing the log likelihood is the same as minimizing the mean squared errors (MSE) of the inferred labels vs the actual labels on the train set (Goodfellow et al.,

2016, section 5.5.1). The solution to that is the same as the OLS fit.

But in trading, we often just want to predict whether the asset will go up or down the next day, or whether a trading strategy will be profitable the next day. In other words, we have a classification problem, not a regression problem. So, y is often a discrete or even binary variable 0 or 1. In this case, we don't just want to predict whether y will be 0 or 1, we want to compute the *probability* $P(y)$ of each. Like linear regression, we assume $P(y)$ is a linear function $\beta \bullet \mathbf{x}$ of the features \mathbf{x} . But it isn't as straightforward here because $P(y)$ can only take on values between 0 and 1, and a linear function gives values between $-\infty$ and ∞ . To fix that, we can feed this linear function through a sigmoid function $P(y) = 1/[1 + \exp(-\beta \bullet \mathbf{x})]$ first.

* * *

Exercise: Ask ChatGPT to plot a sigmoid function. Does it look like [Figure 3.3](#)?

* * *

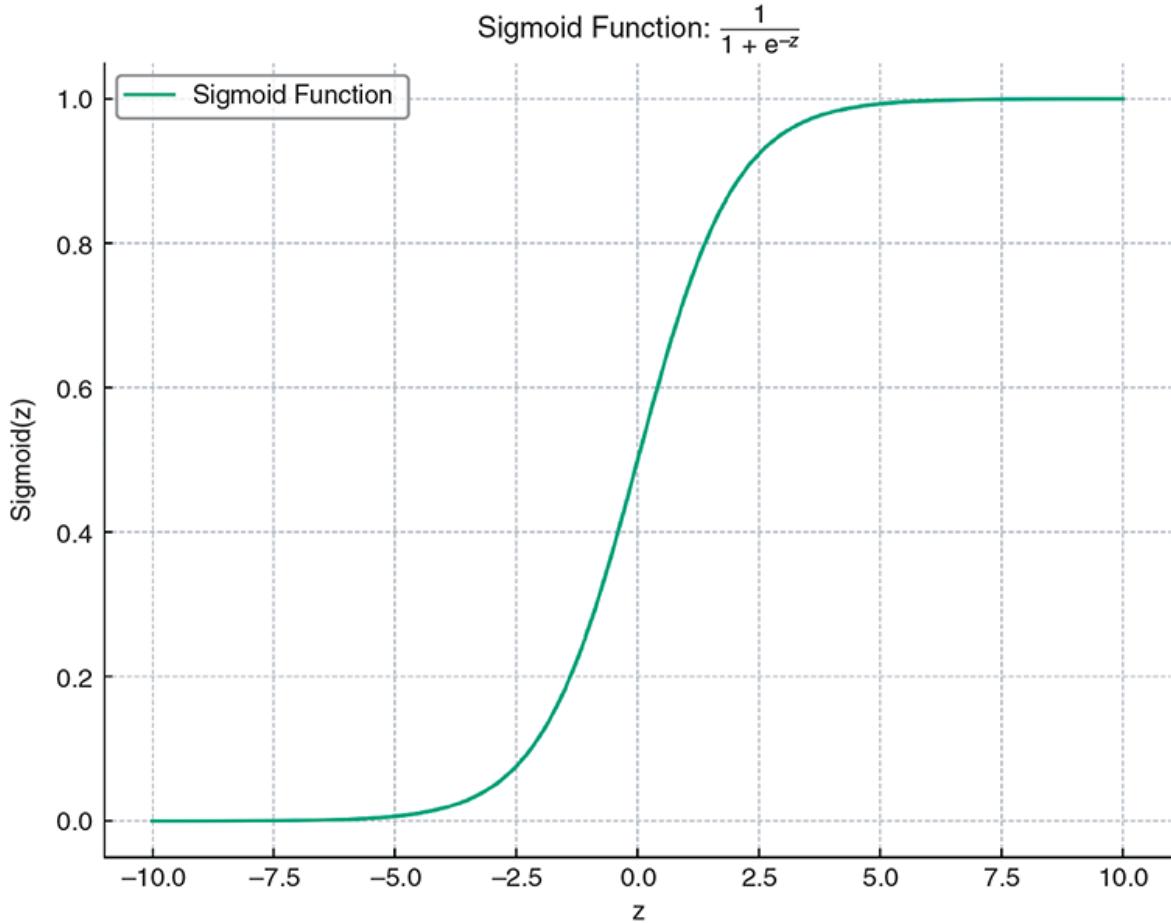


Figure 3.3 Illustration of a sigmoid function.

We now have a quasi-linear parametric probabilistic model of the binary variable. To find the best β based on the observations y , we can again apply MLE to this probabilistic model, but there is no closed-form solution like the OLS fit for linear regression. A numerical optimization method such as gradient descent is needed. This gives us a taste of what's to come—much of neural network and deep learning training is based on gradient descent too, though in neural networks, the regression coefficients β are called “weights” \mathbf{W} instead. We will discuss that soon. This scheme is called logistic “regression” despite being a classification problem. The sigmoid function $1/[1 + \exp(\dots)]$ above is also called a logistic function.

Once we have $P(y|x)$ trained, we can set a threshold to decide when we should infer/predict $y = 0$ or 1 for a given input \mathbf{x} . The conventional choice is 0.5 . But let's say you want your returns prediction to generate a long bias for your trading model, you can always set the threshold to lower than 0.5 . The best threshold is not a problem for machine learning—it is a trading strategy construction problem. You can backtest and optimize the best threshold to achieve the highest CAGR, Sharpe ratio, or the lowest maximum drawdown etc. In other words, it is a hyperparameter to be optimized based on financial metrics.

3.2.2 L1 and L2 Regularizations

In theory, linear and logistic regressions look simple to train (with even a closed form solution for linear regression). But in practice, if we have a large number of features and we don't know whether they are significant or if they are redundant (i.e., highly correlated, also called multicollinear), we will find that β can be very unstable (sensitive to small changes in \mathbf{X}) and it will have large errors. Furthermore, the matrix $\mathbf{X}^T \mathbf{X}$ may even be singular and non-invertible and can't generate any β at all. Also, having too many input features leads to the problem of overfitting in machine learning: the fitted model may work very well on the training data, but generalize very poorly on unseen test data. All these issues call for procedures called “regularization” or “features selection.” The former means reducing the complexity of the model by reducing the “size” of model’s parameters or the number of parameters in a model, and the latter means reducing the number of features in particular. Some regularization techniques might also do automatic feature selection, as we will see later. In the context of linear or logistic regressions, two popular regularization and feature selection techniques are called L1 and L2 regularizations,

which are two different ways to “size” or measure the model’s complexity.

L1 regularization is just adding a penalty term like $-\lambda \sum_j |\beta_j|$ to the log likelihood function to be maximized. (Note we use index j as it is the conventional column index for a design matrix that enumerates individual features.) You may recognize $\sum_j |\beta_j|$ as being the L1-norm for a vector β . $\lambda > 0$ is an adjustable penalty hyperparameter that you can optimize in a “validation set.” (More on hyperparameter optimization and validation sets later.) The point is that this penalty term will decrease the log likelihood if you have too many nonzero β_j ’s, and thus will force the training program to reduce the β of some of the insignificant or correlated features close to zeros, performing an automatic feature selection.

Similarly, L2 regularization is adding a penalty term like $-\lambda \sum_j \beta_j^2$ to the log likelihood function to be maximized. You may recognize $\sum_j \beta_j^2$ as being the L2-norm for a vector β . This penalty term has similar effect $\sum_j |\beta_j|$. But is L1 or L2 regularization better? Researchers found that L1 regularization often results in more zero β_j ’s and therefore the resulting model is more sparse and better regularized. L1 is therefore more of a feature selection than a regularization technique. Murphy (2012, Figure 13.3) explained this well: the constraint $\sum_j |\beta_j| \leq B$ forms a diamond-shaped ball B that has many vertices where a β_j vanishes, and the contour surface where the log-likelihood function is a constant and which we seek to maximize can more easily satisfy the constraint by touching one of these vertices. We reproduced the effect in [Figure 3.4](#).

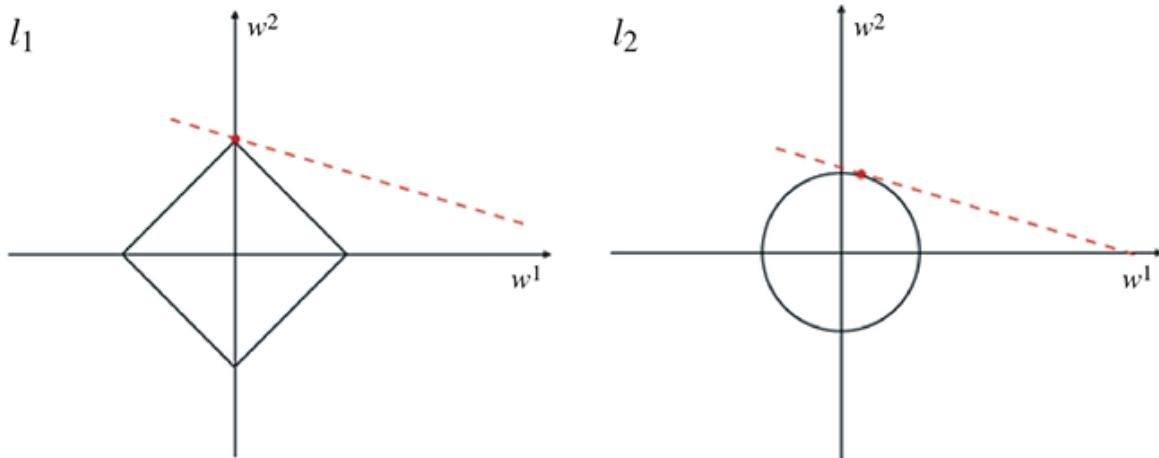


Figure 3.4 Illustration of why L1 regularization can more easily get us zero weights. The black lines indicate where the constraints are satisfied (a diamond for L1, a circle for L2). The red dashed lines represent the constant likelihood surfaces (contours), which we try to maximize by moving it downward. The points where the likelihood surfaces touch the constraints represent the optimized weights. You can see that L1 gives $w^1 = 0$ but L2 does not. (By Mxwsn—

Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=45672058>)

The red dashed line in the figure indicates the constant log likelihood contour, while the diamond and circle and their interiors indicate the L1 and L2 constraints, respectively. The contour will likely intersect the diamond at a vertex with vanishing β_j (or w^j in Figure 3.4.)

L1 and L2 regularizations are also called Lasso and ridge regularizations respectively. One can also apply both L1 and L2 regularizations simultaneously with a penalty term $-\lambda_1 \sum_j |\beta_j| - \lambda_2 \sum_j \beta_j^2$. This is called Elastic Net regularization. Ernie finds these names to be terribly hard to remember and prefers L1, L2, and L1 + L2.

Just as optimizing the parameters of a logistic regression using gradient descent is a general technique that extends to optimizing a neural network, L1 and L2 regularizations during training is also a general technique of regularizing neural networks. However, for very deep neural networks, more specific regularization techniques are applied in practice, as we will see next.

3.2.3 Hyperparameter Optimization, Validation and Cross-validation

Every machine learning model has parameters and hyperparameters. In the linear and logistic regression models above, β contains the parameters, and if you applied L1 or L2 regularizations or both, then λ_1 and λ_2 are the hyperparameters. When we discuss later more complex learning algorithms such as random forest, gradient-boosted trees, or neural network, you will find a larger set of hyperparameters. You can even consider the length of the train set, or even which ML algorithm to apply, as hyperparameters. Some hyperparameters, such as the probability threshold (“decision boundary”) for classification, relate to the financial outcome you want to achieve, and is not a matter for machine learning to decide. We described how we could train the parameters in the case of linear and logistic regression, but that’s assuming we have a fixed set of hyperparameters. How do we optimize these hyperparameters? Validation or cross-validation is the answer. This process is called hyperparameter optimization (HPO). In validation, we divide the data into three non-overlapping subsets: the train set, the validation set, and the test set. Assigning some reasonable starting values for the hyperparameters (e.g., λ_1), we use the train set to find the parameters (e.g., β). Then we test how the model performs in the validation set. The performance metric can be mean squared error (MSE) in the regression case, or log likelihood

in the classification case, or it can be something more complicated like F1 score or AUC score that we will discuss later, or you might even use your financial performance metrics to pick the best parameters. After performance evaluation, we can vary the hyperparameters and retrain the model and repeat the process until the “best” set of hyperparameters that generates the highest performance on the validation set is found. “Best” is limited by your patience and computational resource. There are fancy methods to more intelligently sample the hyperparameter space (such as using a Bayesian optimizer), but often a brute-force grid search is used if we don’t have too many hyperparameters. For example, if the length of the training set is considered a hyperparameter, you will just keep increasing the length until performance on the validation set stops improving. Once the best set of hyperparameters and the corresponding trained model is found, it can be tested on the test set to see if all this effort yields good out-of-sample performance. If not, you are not supposed to go back to modify the hyperparameters and try again (e.g., adding features, try different ML algorithms, try different training sets, ...). You should admit defeat and abandon your project (López de Prado, 2015). However, you are allowed to change your labels and start a new project.

There is one small issue with taking a chunk of data out to serve as a validation set: we have reduced the amount of data that can be used for training, which reduces the ability of your model to generalize. However, as we will see more details in [Chapter 10](#), modern techniques allow you to train a model using small datasets with good generalization by leveraging pre-trained models. Also, especially for financial time series, which tends to violate the standard assumption of independent and identically distributed (IID) observations in most ML models, who is to say that a specific period should be used to pick the optimal hyperparameters and not another? An easy way to remedy that is to apply “cross-validation”: cut up any data *that is not in the test set* into K

subsets (called “validation folds”), train model k on the data *not* in the k-fold and make inferences on the k-fold. Average the validation set performance over all k-folds and optimize hyperparameters based on this performance. This way, we have used all the data outside of the test set for optimizing the hyperparameters. (By “averaging performance,” we mean making inferences/predictions on every sample in the k-folds and then compute any performance metric you like on those inferred labels. We do *not* mean just averaging the Sharpe ratio or some other nonlinear metrics over different folds.)

There are further issues when cross-validation is applied to time-series data, such as when the features used for training model k overlaps in time with the time period of the k-fold, thus possibly inflating the performance of model k on k-fold due to look-ahead bias. López de Prado (2018) discussed the remedies to this problem extensively.

3.2.4 Performance Metrics

For asset managers, the ultimate performance metrics of any ML system are financial metrics such as Sharpe or Calmar ratios. But if we want an intermediate performance metrics just for the machine learning component of an investment system, there are other standard ones. These can be used for ML model validation, comparison, or hyperparameter optimization.

For regression problems, the performance metrics are the familiar Mean Squared Error (MSE, the closer to zero the better) and R -squared (R^2 , the closer to 1 the better). As a reminder, R^2 measure the proportion of the variance in the label that can be explained by the features. However, before we get too excited by a good MSE or R^2 , we should perform some sanity checks. As an illustration, see the Anscombe quartet

(https://en.wikipedia.org/wiki/Anscombe%27s_quartet) in Figure 3.5.

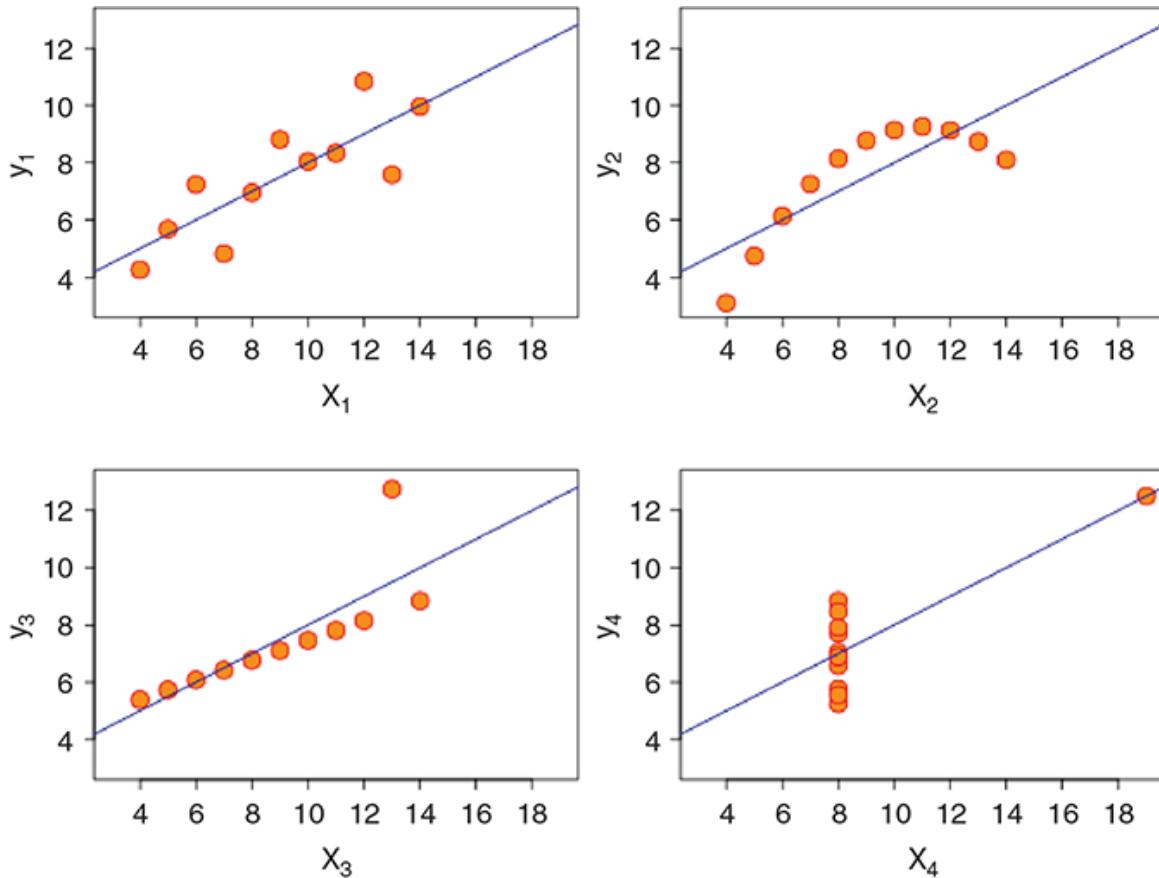


Figure 3.5 The Anscombe quartet. All four data sets have same means, variances, correlations, regression coefficients, and R^2 .

Source: Anscombe / <https://commons.wikimedia.org/wiki/File:Anscombe.svg#/media/File:Anscombe.svg>, last accessed on 13 January 2025 / CC BY 3.0.

These four very different scatter plots of (x, y) turn out to have the same high R^2 of 0.67 and many other descriptive statistics, but no one would think that linear regression is a good fit for the second and fourth set of data. Meanwhile the regression coefficient is biased for the third set due to an outlier. Visual inspection tells a very different, and more accurate, story than a numerical statistic such as R^2 . However, when we have a high dimensional data set, it is

very difficult to find these anomalies via visual inspection. What we can do in those cases is to perform exploratory data analysis (EDA) to determine if there are any outliers in the data and if those are due to errors and should be excluded. Then we can apply features importance ranking to find the top-ranked features. Even better, we can use cluster-based MDA (cMDA) (see López de Prado [2020]) to find the top-ranked clusters of features, and see whether they make sense. Sometimes, especially important features that have high predictive powers turn out to embed look-ahead-bias and should be discarded. We will discuss more about feature engineering later in this chapter.

For binary classification problems, and the majority of the probabilistic approaches to machine learning, we have already seen that the usual metric to optimize is the log likelihood (LL). This is also the metric to compare the inference powers of different classifiers (or maybe the same classifier with different hyperparameters) given labels in the test set. However, the LL score doesn't give an intuitive understanding of how good a classifier is, just as MSE doesn't give an intuitive appeal of how good a regression is because its value depends on the labels' inherent variance. There are other classification performance metrics that are often used instead.

The most common classification metric is accuracy of prediction: $\frac{\# \text{ correct predictions}}{\# \text{ predictions}}$. You might think that any accuracy over 50% means the model is better than random and is working. But imagine if the labels are very imbalanced: 90% of them are 0 and only 10% of them are 1. We can just blindly bet on 0 and get a 90% accuracy rate. We need additional measures to demonstrate the power of our ML model. Precision and recall are such measures. Typically, these are applied to the minority class, or the class with label 1 that we consider the results are "positive." For example, in Corrective AI, this would be the case where the investment

decision is profitable. The negative class is the one presumed to be in the majority with label 0. Precision is

$\frac{\# \text{ correct predictions of } 1}{\# \text{ predictions of } 1}$ and recall is $\frac{\# \text{ correct predictions of } 1}{\# \text{ of actual } 1}$. The

correct predictions of 1 are called the true positives (TP), while the incorrect predictions of 1 (also known as Type I error) are called the false positives (FP). The correct predictions of 0 are called the true negatives (TN), while the incorrect predictions of 0 (also known as Type II error) are called the false negatives (FN). So, you can also write

$\text{precision} = \frac{TP}{TP + FP}$ and $\text{recall} = \frac{TP}{TP + FN}$. Typically, both recall and precision should be above 50% to indicate any predictive power. In our example, if the model were to always bet on 0, recall will be 0, indicating the model doesn't work. If the model were to always bet on 1, precision will be just 10%, again indicating the model doesn't work. To take into account both precision and recall, people often calculate the F1 score which is the harmonic mean of recall and precision:

$2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$. You can see that F1 score is obviously 0 if either precision or recall is zero. It will also be far smaller than 0.5 if either precision or recall is far smaller than 0.5. If we want to drill down to details, we can also compute the confusion matrix for each of the quadrant ([Table 3.3](#)):

TABLE 3.3

The confusion matrix for binary classification.

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

We find this mnemonic helpful: Type I errors (FP) are more serious than Type II errors (FN) in trading because FP may mean accepting an investment decision that is wrong and we

lose money, while FN means not accepting an investment decision that turns out to be correct and profitable but we are keeping our money safe in the bank. Hence FP is given priority as Type I error. The confusion matrix is also giving the priority of the first row and column to the positive class.

When we have a multiclass classification problem (e.g., return is labeled very negative, mildly negative or 0, mildly positive, very positive) the precision and recall scores need to be computed by regarding each class as the positive class in turn and the other classes grouped together as the negative class. The F1 score is then computed as some sort of average over these recall and precision scores. We can't explain it better than Copilot:

Copilot Response

Micro-average: This method aggregates the contributions of all classes to compute the average metric. It calculates the F1 score globally by counting the total true positives, false negatives, and false positives.

Macro-average: This method calculates the F1 score for each class independently and then takes the average. This treats all classes equally, regardless of their size.

Weighted-average: This method calculates the F1 score for each class independently but when it averages them, it uses a weight that depends on the number of true instances for each class.

A confusion matrix for a three-class classification problem would look something like [Table 3.4](#).

TABLE 3.4**Confusion matrix for three-class classification.**

	Predicted Class 1	Predicted Class 2	Predicted Class 3
Actual Class 1	TP	FN	FN
Actual Class 2	FN	TP	FN
Actual Class 3	FN	FN	TP

(Of course, each of the TP and FN will have a different number in each cell.)

* * *

Exercise: Ask Copilot to generate a random 3×3 confusion matrix and ask it to compute recall and precision for each class, and compute the three types of F1 scores. Manually check if these numbers are correct.

* * *

There is one particular hyperparameter that we usually want to optimize in a binary classification problem: the probability threshold p_0 that decides whether we should infer 0 or 1. This is also called the “decision boundary” in ML in general. Which performance metrics should be used for this?

The usual objective function to be optimized, the log likelihood (LL), does not include p_0 as the hyperparameter as it only requires the probability distribution and the actual labels as input. If we use precision or recall as the objective, increasing one by tuning p_0 will typically lead to the decrease of the other. This can be seen from the hypothetical PR curve in [Figure 3.6](#) (generated by ChatGPT), which plots

the precision against the recall with decreasing $p(0)$ toward the right side.

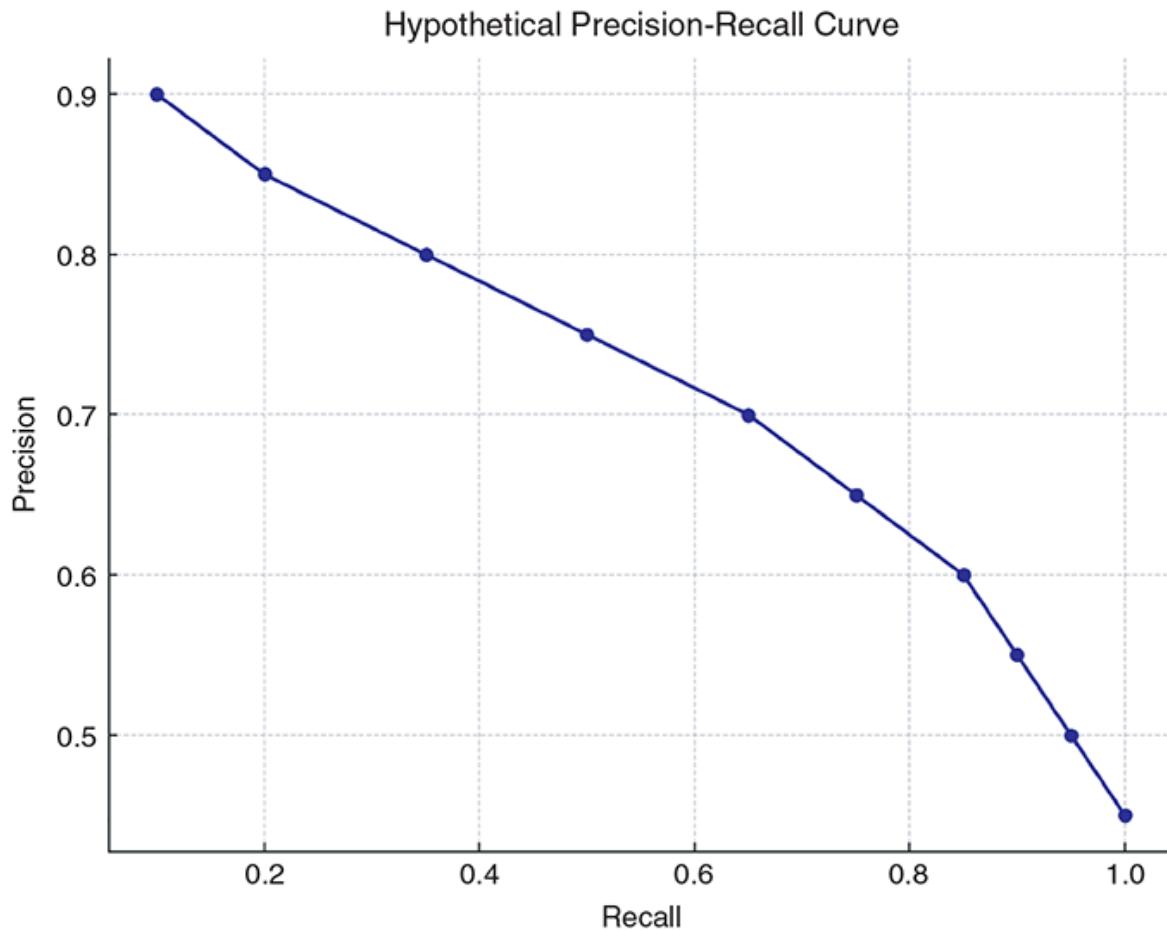


Figure 3.6 A typical precision-recall curve.

We can also see this from the ROC curve in [Figure 3.7](#), which plots the True Positive Rate (TPR, a.k.a., recall) against the False Positive Rate ($FPR = FP/\#actual\ negatives = FP/(FP + TN)$) again with decreasing $p(0)$ toward to right side. As $p(0)$ decreases, naturally we get more predictions of positives, some right and some wrong, and thus both TPR and FPR increase. It is obvious that recall can at most be 1, and FPR can also be at most 1, both of which happen when we mislabel all the negatives as positives. A binary classifier that assigns probabilities randomly with $p(0) = p(1) = 0.5$ will give a straight diagonal line as the ROC curve, and the more discriminating power the classifier has, the more negative its

curvature. This means at any point of the curve except the endpoints, $\text{TPR} > \text{FPR}$. (If a ROC curve has positive curvature, we just need to infer 0 when $p(1) > p_0$ and we will recover the same discriminating power. That is, a false prophet that is consistently wrong is a true prophet.) This also means that we can use the area under the ROC curve as the statistic, which describes the discriminating power of the classifier. This area is called the AUC (area under curve), and it should be greater than 0.5 for a good classifier. This statistic is useful for comparing different classifiers as well, particularly during the modeling phase, where we want to evaluate model performance independently of the chosen probability threshold. There is also a related measure called AUCH (area under convex hull), which is very similar to AUC but applies to the convex hull that is fitted to the actual precision and recall data.

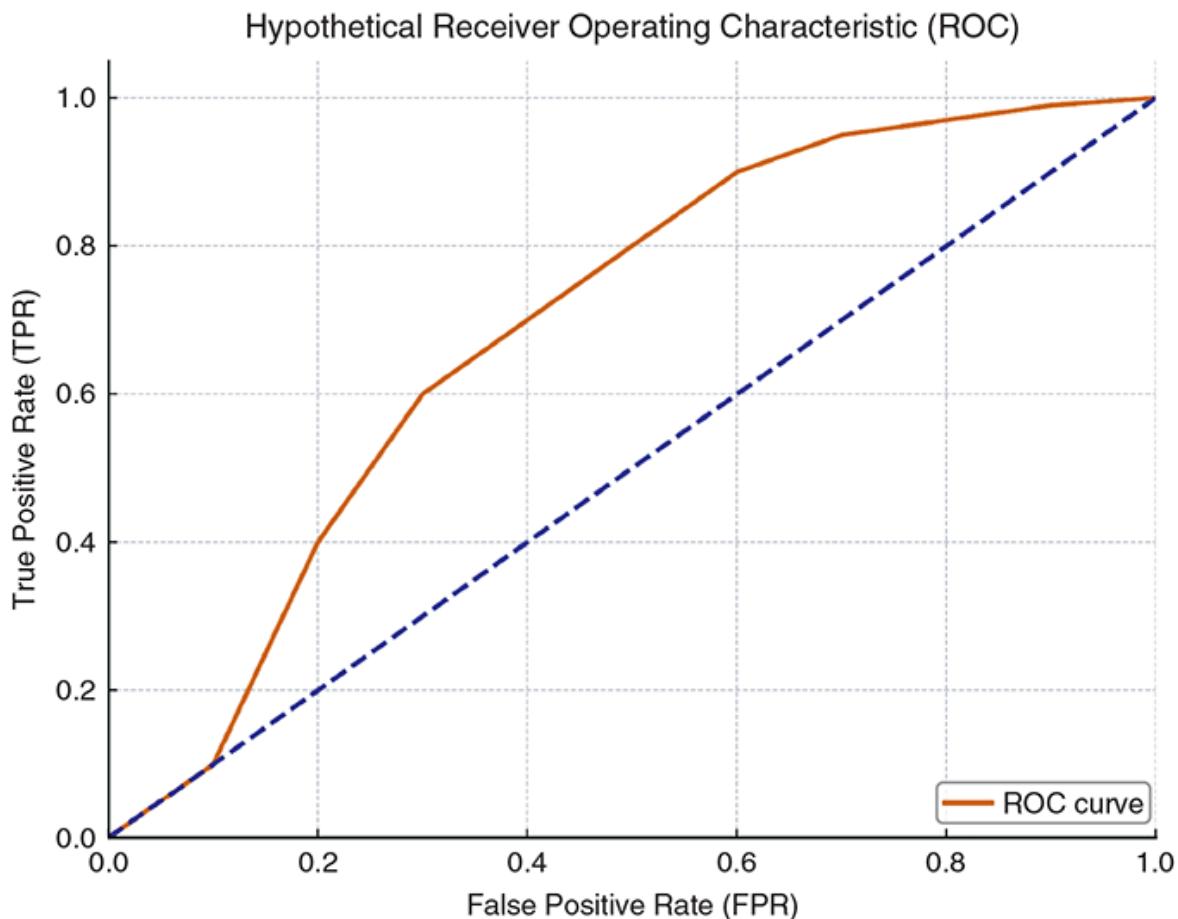


Figure 3.7 A typical ROC (Receiver Operating Characteristics) curve.

* * *

Exercise: Ask ChatGPT to generate 1,001 random samples of $\gamma = 0$ or 1 based on the binomial distribution, with the probability of class 1 determined by a sigmoid function of a scalar feature x that takes on values -500 to 500 increments of 1 . Ask it to fit a logistic regression on this simulated data, and compute the various performance metrics just described. Are those metrics computed correctly? Does the model fit the data well? (Hint: it has to! This is just in-sample fitting.)

* * *

3.2.5 Classification and Regression Trees, Random Forest, and Boosted Trees

Tree-based models are widely regarded as the most effective ML technique for tabular, inhomogeneous features (Shwartz-Ziv and Armon, 2021). Inhomogeneous features mean that features may have very different statistical characteristics. In finance, one feature might be annualized GDP growth, another might be a categorical feature of whether Fed interest rate announcement was made on the previous day, and yet another might be the 1-day return of the SPY. In contrast, neural networks are widely regarded as the most effective ML technique for homogeneous features such as image, videos, acoustic (including speech), and textual features. Homogeneous features are exemplified by words in a text file: every word is usually drawn from the same vocabulary. As we shall see, tree-based models typically select one feature at a time, so the fact that different features have different statistical properties is of no hindrance. Plain-vanilla classification and regression trees are also more explainable/interpretable than neural networks, though the embellished versions such as gradient-boosted trees or random forest are less so.

The most popular and basic method of training a tree is called CART (Breiman et al., 1984). Depending on the context, sometimes we also call the tree itself a CART. For a classification problem, the algorithm starts with the entire train set and goes through the list of features and picks one that separates the samples into two subsets (“child nodes”) such that each one contains “mostly” samples of the same label. To be technical, that means we choose the feature such that the weighted sum of the Gini indices of the two subsets is the lowest. If the feature chosen is continuous as opposed to categorical, then the choice of the feature and the threshold c (denoting cutoff) to divide the feature’s range of values into two are made at the same time. Repeat this

process recursively with each child node, until a child node's number of samples drops to or below the minimum, or when the depth of the branch reaches a certain maximum. As you can tell, it is a greedy algorithm, because it always picks the best feature one at a time, without regard to the possibility that picking a few suboptimal features consecutively may ultimately result in a lower Gini index. It is easy to see how we can infer the label given a new set of features—just follow the decision rules down to the leaf node and see which class it belongs to. The probability of ending up in that class y based on feature x_1, \dots, x_j picked before the leaf node $n+1$ is simply

$$p(y|x_1 \geq c_1, x_2 \geq c_2, \dots, x_j \geq c_j) = \frac{\# \text{ class } y \text{ training samples in leaf node } (j+1)}{\# \text{ training samples in leaf node } (j+1)}.$$

(There is no loss of generality to assume the decision rule is \geq because we can always apply a minus sign to the continuous features, and the $=$ takes care of the categorical features.)

For a regression problem, the CART algorithm is very similar, except that the criterion for choosing a feature is determined by minimizing the weighted MSE of the child nodes. For inference, we can just compute the average value of the labels in the training samples in a leaf node

$$p(y|x_1 \geq c_1, x_2 \geq c_2, \dots, x_j \geq c_j) = \frac{\sum_{s \in \text{node } j+1} y_s}{\# \text{ training samples in leaf node } (j+1)}.$$

The simplest regularization techniques for trees are to set a minimum for the number of samples in any of the nodes and/or set a maximum depth (i.e., the maximum number of features used) of any branch of a tree. Data scientists have also found that it is helpful to average the inferences of an ensemble of K trees where each tree is trained using samples randomly drawn with replacement from the original train set. This ensemble is called the Random Forest. For a classification tree, that means computing the average of the probability of inferring class y and picking the y that has the highest probability given the feature values. For a regression

tree, one can just average the inferred values of the label across all the trees given the feature values.

Performance of any machine learning model can be boosted by a technique called, you guessed it, boosting. The most popular implementation of boosting is called gradient boosting. It is yet another recursive algorithm. Assume we already have a boosted model M_k (which is initialized to the original unboosted model $k = 0$). It makes an inference \mathbf{y}_k (we use a vector here to indicate many samples of labels). We compute the errors of these inferences $\mathbf{y}_k - \hat{\mathbf{y}}$ and use these errors as the new labels to train an error prediction/correction model m_k . A new ensemble model $M_{k+1} = M_k - m_k$ is now ready for the next iteration. (We use this notation to indicate the output of M_{k+1} is the output of M_k minus the output of m_k .) The inference errors (at least on the train set) of M are supposed to decrease with every iteration. (See <https://developers.google.com/machine-learning/decision-forests/intro-to-gbdt>.) We will stop when the errors become small enough (in the classification case, the number of positive cases become small enough), or when the improvement in error rate is not big enough, or when we exceed the maximum number of iterations. Notice that although we hook up several models together in a boosted model/tree, we add these models sequentially and the parameters of the existing sequential of models are not modified in subsequent training. As we shall see, this is similar to neural networks (NN) in that NN also connect multiple elementary models together, but in a NN the parameters of the elementary models do change during training.

One way to regularize a boosted model to reduce overfitting is to introduce a learning rate $0 \leq \epsilon \leq 1$ such that

$M_{k+1} = M_k - \epsilon m_k$. In other words, we don't trust the full predicted error to correct the model. The learning rate ϵ and the maximum number of trees $n_estimators$ in the ensemble

are the typical hyperparameters to be optimized. When ϵ is close to 0, *n_estimators* need to be high to minimize the errors. We will find the concept of learning rate to be relevant to neural network training, discussed next.

There are many attempts to improve on the classical CART algorithm for tree-growing. There are also deep learning techniques (specifically using attention) applied to trees (Zhuang et al., 2024; Arik and Pfister, 2020). In financial applications, Cong et al. (2024) described the Panel Tree approach for classifying entire stock returns series that result in superior investment performance. These efforts are made because of the unique interpretability of CART and its power on tabular data. There is still ongoing debate about the predictive power of tree-based models compared to deep learning for tabular data. An excellent paper showcasing the performance of tree-based models in this context is by Grinsztajn et al. (2022), which we recommend to readers for further details.

3.2.6 Neural Networks

You may notice that the machine learning algorithms we have described so far, linear or logistic regressions or CART, always output a scalar label y , whether it is continuous-valued or categorical. (We may write y as a vector sometimes, but only because we stack all the samples together during training.) They can't be used to produce a vector. You can of course build multiple models using the same input to produce a vector as in multivariate regression, but those models are trained separately and have independent parameters. However, sometimes we find that training a model on labels that are vectors generates better inferences. For example, in Kolm et al. (2021), the authors used a neural network to predict the term structure of returns based on the limit order book. The fact that we know (as labels, not features) the future returns over several time

frames provides a richer information set to train a better model. Neural networks is a much more flexible ML algorithm that can produce such vector outputs. This benefit may outweigh the drawback we mentioned earlier that tree-based models work better for tabular and heterogeneous features—when the labels are vectors, you have no choice but to use NN.

* * *

Exercise: What are the exact inputs used in the Kolm paper just cited? How many different time horizons are used for the labels? What kind of neural network architecture is used? Ask Copilot to help, or even generate codes based on this paper!

* * *

To motivate NN, we saw earlier how boosted machines combine multiple elementary models (e.g., each of them a single tree) sequentially to boost performance. To extend this concept, why not connect these models both in parallel and sequentially? Why not take multiple outputs from different models so that the global output is a vector or even a tensor? Why not allow all these machines' parameters to be tunable during training instead of fixing them sequentially? If we allow all these possibilities, we end up with a feedforward neural network, also known as a multilayer perceptron (MLP), which is the simplest type of NN (Goodfellow et al., 2016). We show a simple one in [Figure 3.8](#).

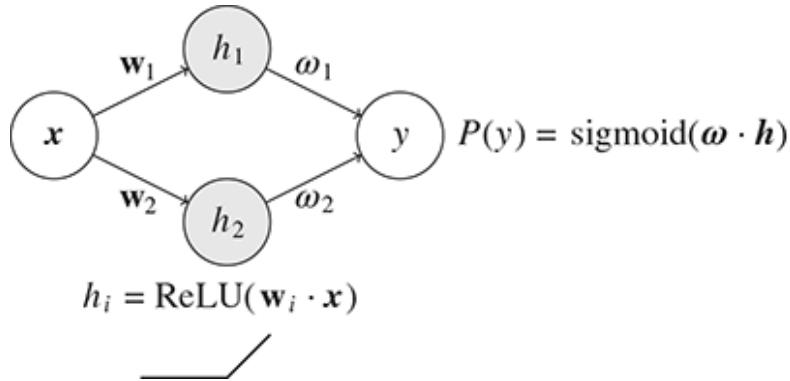


Figure 3.8 A one-hidden-layer MLP that takes a vector of features \mathbf{x} to classify a scalar, binary label y . Every arrow denotes a linear function (dot product) with weights (regression coefficients) and biases (offsets). If this were for regression, we can omit the sigmoid function at the output node. In general, we can also predict a vector of binary labels \mathbf{y} , with different weights ω . We can also need more nodes to the hidden layer, and/or add more hidden layers. To simplify notation, we have used w and ω to represent both the weights and biases.

The simplest MLP has three layers:

1. An input layer that is just a several linear regressors i that each accept the same features \mathbf{x} and each output a scalar $\mathbf{w}_i \bullet \mathbf{x}$. We use \mathbf{w}_i instead of β_i because this is typical in NN literature. To simplify notations, we have also used \mathbf{w} instead of \mathbf{w} and \mathbf{b} to include both weights and “biases,” which are the constant offsets in linear regressions. (This means we assumed a “1” to be added to the input vector \mathbf{x} . This is for notational convenience because, in practice, for reasons of computational time and memory complexity, it is more efficient to add the biases directly rather than appending one to the input,

particularly when the number of regressions becomes large.) They form the fundamental parameters of a NN.

2. These scalars $w_i \bullet x$ are each fed into a hidden node. Here we introduce a nonlinear function that is as close to a linear model as possible: like the ReLU unit $h(z) = \max(0, z)$. The output of each ReLU unit is again a scalar, $h_i = \max(0, w_i \bullet x)$, and known as the activation.
3. These scalars h_i together form a vector that is fed into the output node(s), which is another linear regressor $y = \omega \bullet h$ for a regression problem, or our familiar sigmoid function $y = 1/[1 + \exp(\omega \bullet h)]$ for a binary classification problem. The latter represents the probability of a class, as in logistic regression. Again, we have simplified notations to use ω to represent both weights and biases, and assumed we added “1” to the vector h .

Since we have used the same symbol to represent both weights and biases, we will also refer both weights and biases as just “weights” in the text.

We can extend MLP by adding more hidden layers, where the outputs of one hidden layer are used as inputs of another hidden layer. We can also use higher dimensional tensors as inputs, weights, and outputs. For multiclass classification, we can replace the sigmoidal function with the softmax function

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

Do not be alarmed by all these matrix multiplications. NN libraries such as PyTorch or TensorFlow will deal with them for you, as long as you specify the dimensions of the inputs and outputs, the number of hidden layers, and the number of nodes each layer.

There is research to show that a MLP with just one hidden layer can approximate any reasonable function (Hornik et al., 1989). But what this research didn't say is that we will need $\exp(\# \text{ number of features})$ nodes in this layer to have a good approximation. People found that it is generally more efficient to train a network with more hidden layers, each with few nodes, than a single hidden layer with many nodes.

To train a NN, we need to decide on the objective function. As in linear and logistic regression and almost every other ML problem: optimizing the log likelihood is most of the time the right choice. Assuming a Gaussian probability distribution for the output layer, this is again equivalent to minimizing the MSE for a regression problem as mentioned for linear and logistic regressions. To maximize the log likelihood for logistic regression, we can again resort to gradient descent of the parameters, now called weights. The problem with applying gradient descent to the much larger set of parameters that are typical of a NN and with a large training set is the prohibitive computational time. Hence there is a more effective algorithm called stochastic gradient descent (SGD) that samples a small subset of the train set when computing the gradient. This subset is called the minibatch—at most a few hundred samples. The smaller the size of a minibatch, the better the network is regularized. Just as in the boosting algorithm we described above, we will update the parameters $\theta_{i+1} = \theta_i - \epsilon g$ where θ represent all the weights of the network, g is the gradient, and ϵ is the learning rate. (There are several techniques to accelerate model training by modifying these equations, as we will see later.) The computation of g can be efficiently performed by a forward-backward algorithm called backpropagation.

Backpropagation is based on the chain rule of calculus to recursively compute g , which is the first partial derivative of the negative log likelihood L (or in general any cost function J) with respect to a parameter θ_i . The inferred output node y (together with the actual label \hat{y}) of a NN can be used to

compute the cost function. Using the chain rule of calculus,

$g_i = \frac{\partial J}{\partial \theta_i} \Big|_{\hat{y}} = \sum_j \frac{\partial J}{\partial h_j} \Big|_{\hat{y}} \frac{\partial h_j}{\partial \theta_i}$. Note that the sum is over all the hidden nodes j that lead forward to the output node \mathbf{y} , and h_j is the output from node j . In other words, over all the parent nodes of \mathbf{y} . To compute $\frac{\partial h_j}{\partial \theta_i}$, we can invoke the chain rule

again and express it in terms of all the partial derivatives

$\frac{\partial h_j}{\partial u_k}$ with respect to all the outputs from all the parent nodes

k that lead forward to the node j . And so on and so on until we reach the node with the parameter i . Except for the

output node, all the other partial derivatives $\frac{\partial h_j}{\partial h_k}$ are just

simple functions of all the outputs of the parent nodes of j as well as the weights that take the output of parent node k and turn it into an input to the child node j . For example, for hidden node $j = 3$, let's say

$h_3 = \text{ReLU}(0.1 * h_1 + 0.2 * h_2 - 0.2)$, and $h_1 = 1$ and $h_2 = 1$,

then $\frac{\partial h_3}{\partial h_1} = 0.1$. If $h_1 = 1$ and $h_2 = 0$, then $\frac{\partial h_3}{\partial h_1} = 0$. For the

output node, let's say J is the MSE function, and

$J = \sum_s (0.1 * h_3 + 0.2 * h_4 + 0.3 - \hat{y}_s)^2$ where s is over all samples in the minibatch, and $h_3 = 1$ and $h_4 = 0.5$. Let's further assume we only have 1 sample in the minibatch and

$\hat{y} = 1$, then $\frac{\partial J}{\partial h_3} = 2 * (0.1 * 1 + 0.2 * 0.5 + 0.3 - 1) * 0.5$.

* * *

Exercise: In the previous example, what would $\frac{\partial h_3}{\partial h_1}$ be given

two alternate sets of h_1 and h_2 ? If the minibatch has two

samples, and $h_3 = -1$, $h_4 = 1$, and $\hat{y} = 0.5$ for the second sample, what would J be?

* * *

To start the training (backprop) process, we take samples from a minibatch of train data, initialize the parameters θ_1 to some values (recommendation: initialize weights to small

random values in the range $\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$ where m is the

number of features, and initialize biases to zero) and run them *forward* through the network to compute all the outputs u_i of all the nodes i . That is, going layer by layer and link by link through all the linear and ReLU or sigmoid functions, through all the matrix multiplications of weights, and then finally arrive at the output nodes y . All the y from different samples are combined with their corresponding actual labels

\hat{y} to produce $\frac{\partial J}{\partial u_n} \Big|_{\hat{y}}$ aggregated over all samples in the

minibatch. This is the forward propagation that produces the output values of all the nodes. Now we are ready to compute the gradient g using the chain rule described earlier, starting from output node and going *backward*, like the numerical example above. After g is computed, we can use it to update $\theta_2 = \theta_1 - \epsilon g$. That finishes one iteration of the SGD, and we can start again with a different minibatch. We will discuss later when we should stop iterating. It is often not necessary to wait till the gradient is 0 to stop, nor should one be concerned whether we have reached a local or global minima in the cost function.

In this forward-backward algorithm, no partial derivative is ever computed more than once, because whenever one is computed, it is stored for use in another backward propagating path. For any network with η nodes, there can't be more than η^2 edges and therefore more than $O(\eta^2)$ operations to compute the gradients. Most often, the NN

won't be fully connected and has only $\mathcal{O}(\eta)$ edges, resulting in $\mathcal{O}(\eta)$ operations.

If you find training a NN daunting, keep in mind that you won't actually need to implement any of these complicated steps—PyTorch, TensorFlow, etc. make use of automatic differentiation (also known as autograd) and do it for you. But it is good to know the high-level principles just so you know what hyperparameters to try.

To regularize a NN, we have already mentioned the number of samples in a minibatch is one hyperparameter. The smaller this is, the more regularized is the network. Another hyperparameter for regularization is the learning rate ϵ during gradient descent. The smaller ϵ is, the more regularized the network. However, it is found that training works better if ϵ decreases linearly over time. Goodfellow et al. (2016) suggests that after a few hundred iterations, ϵ should drop to 1% of its initial value and stay there. As an initial value, one can optimize it over the validation set with the first 100 iterations and set it just a little higher than the optimal. Other technique is to use a cyclical learning rate, where the learning rate varies within a range of values rather than only decreasing it in a step-wise manner (which, by the way, is already implemented in PyTorch, see https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.CyclicLR.html). More details about this approach can be found in Smith (2017).

We can also use L1 or L2 regularizations. The cost function J above can be set to the negative log likelihood plus $\lambda_1 \sum_i |W_i|$ or $\lambda_2 \sum_i W_i^2$ or both. Note that typically we only regularize the weights but not the biases. As usual, the bigger the λ 's, the more regularized the network. However, as mentioned previously, these regularization techniques are less commonly used in practice for deep neural networks.

Yet another popular regularization technique is early stopping: essentially treating the number of training steps

(one forward-backward iteration) as a hyperparameter to be optimized on the validation set. Once the cost drops to a plateau in the validation set as a function of training steps, we will stop training even if further training will decrease the cost on the train set further.

Last but not least, there is the powerful technique of dropout (Srivastava et al., 2014): randomly removing non-output nodes of a network for each minibatch's iteration (typically input nodes are included with probability of 0.8, hidden nodes are included with probability of 0.5). However, researchers found the dropout doesn't work well when the train data has fewer than 5,000 samples.

While SGD is a canonical training technique, researchers have tried to improve its convergence rate. There is a method called momentum that uses an exponential moving average of the gradients computed over several iterations. This is called the “velocity” \mathbf{v} , and the update rule becomes $\theta_2 = \theta_1 - \epsilon\mathbf{v}$. Other improvements have been proposed, with names like RMSProp, Adam, and BFGS. There is also “batch normalization,” which seeks to keep the mean and variance of the output of input or hidden units fixed during training by normalizing them to almost zero and one, respectively. Only correlations between hidden units and higher order statistics are allowed to change. Details can be found in Goodfellow et al. (2016).

* * *

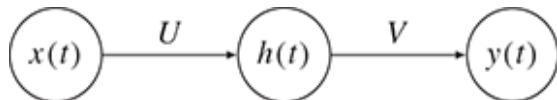
Exercise: Construct a neural net with one hidden layer where the layer has two hidden nodes for classification. Connect the output of the first hidden node to both the output node and the second hidden node. Note that this means it is no longer a feedforward NN nor a MLP. Label = Is next day return positive? Features = last 1 day SPY return, last 2 days SPY returns. With the help of ChatGPT, write a Python program without using any NN libraries like Tensorflow to train this

network. What is its performance? Can you tune the hyperparameters to improve validation set?

* * *

3.2.7 Recurrent Neural Network

Let's say we have a standard time-series prediction problem in trading: we want to predict the next period's return using a technical indicator known in the current period. For concreteness, let's say we want to use a MLP (or indeed any ML system such as CART) to predict the next period's return $y(t)$ using the market index return $x(t)$, as feature. Each sample, i.e., each period, has a scalar label and a scalar feature. We can use the simple MLP in [Figure 3.9](#) to represent this:



[Figure 3.9](#) A simple MLP for time-series prediction.

* * *

Exercise: Ask ChatGPT to help train this MLP using SPY daily returns as the feature, and MSFT daily returns on the next day as labels. Regularize the MLP and optimize the hyperparameters on a validation set. How do the Sharpe ratios look on the train, validation, and test sets?

* * *

U indicates the weights connecting the input node to the hidden node h , and V indicates those connecting the hidden node to the output node y . However, it may be advantageous for a ML model to remember what the past features and labels were when predicting the next label, instead of treating them as independent samples, especially during inferences. One simple way to pass information from the previous features *and* labels from timestep $t - 1$ to t is to feed

the output of $h(t - 1)$ as input to $h(t)$. We may envision a network at t that looks like [Figure 3.10](#).

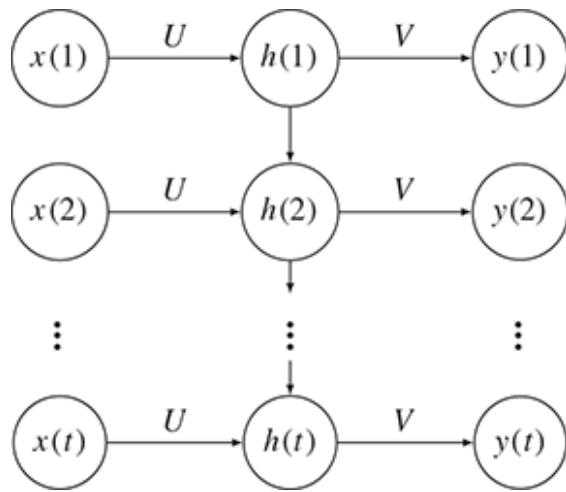


Figure 3.10 Tying the weights to the same values and connecting the hidden nodes to transmit information from $t - 1$ to t .

Since the time series is assumed stationary (if not, ML won't work), we can force all U , V , and W (the weights connecting the hidden nodes) to be the same at every t . During training, all the features and labels up to $t - 1$ would be available and are used to indirectly determine $h(t - 1)$ as they will affect the values of U , V , and W . Every time we advance a time step to t , the network adds another input node, hidden node, and output node. Since weights are presumed the same throughout the network, there is no problem in making a prediction at t . Once we have the actual label $\hat{y}(t)$, we can retrain the network and adjust all the weights (if we want to). We have just motivated one prototypical recurrent neural network (RNN). Most books display the RNN by rotating our figure 90° counterclockwise so that time proceeds on the horizontal axis (as is common in stock price charts) and the output is at the top, as in [Figure 3.11](#).

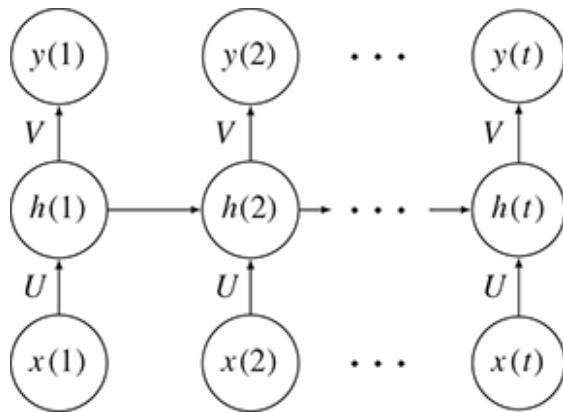


Figure 3.11 Same as [Figure 3.10](#), but rotated 90° counterclockwise so time proceeds from left to right.

Notice the resemblance with a Hidden Markov Model. During prediction, you can think of the previous inputs x from 1 to $t - 1$ as modifying a hidden state $h(t - 1)$ that serves as another input to $h(t)$ and affecting the prediction $y(t)$, just like the hidden state in a HMM that determines the probability distribution of the emissions $x(t)$. A few key differences between standard HMMs and RNNs are that RNNs can model more complex mappings for state evolution and predictions (from states to observations) using neural networks. Also, the evolution of the state in RNNs is deterministic, and not stochastic as it is for HMMs.

There are other prototypical RNN architectures—for example, we can connect the output $y(t - 1)$ to the hidden node $h(t)$. Designing network architecture is like designing a building, the architect can let their imagination run free, subject to training practicalities imposed by the (civil) ML engineer. To actually train a RNN successfully, though, isn't a trivial problem. It is plagued by the “vanishing” and “exploding” gradient problems (Goodfellow et al., 2016). The gist of these problems is that in backpropagation using the chain rule, we have to multiply many gradients together. If the network is very deep, you are essentially computing g^n where g is the gradient and n is the number of layers (depth) of the network. If $g < 1$, $g^n \rightarrow 0$ as $n \rightarrow \infty$, i.e., it vanishes. If

$g > 1$, $g^n \rightarrow \infty$ as $n \rightarrow \infty$, i.e., it explodes. In either case, SGD fails. The solutions to these problems give rise to LSTM, GRU, the attention mechanism, and transformers. The latter became the key to building large language models such as GPT-4. So, we have come full circle. For a practical example of using LSTM for financial predictions, see Sirignano and Cont (2018) and Kolm et al. (2021). More details about the attention mechanism, and transformers models will be given in [Chapter 6](#).

* * *

Exercise: Ask ChatGPT to use Tensorflow or PyTorch to create such as RNN and train it on SPY and MSFT as in the previous exercise. Does your training succeed? If so, how does the performance look on test set? Is it better than the previous simple MLP?

Exercise: Ask ChatGPT, “Can you find a paper that describes financial time-series prediction using LSTM and that comes with a GitHub?” and try to implement the model.

* * *

3.3 Deep Reinforcement Learning

In unsupervised learning, we don't have the correct answer (label) to a problem (such as “What is the best way to cluster stocks?”). In supervised learning, we have the answer in hindsight to a problem (such as “What will the return of MSFT be?”). There is an intermediate situation though: we don't know the correct answer, but we know that one answer is better than another to a problem (such as “What asset allocation is best among these 100 stocks?”). Reinforcement learning (RL) is designed to handle this situation. The prototypical problem is a chess move: you won't know if a move is the “best” move possible, but you know a move is bad if it immediately leads to checkmate! Similarly, the

prototypical problem in finance is portfolio optimization. You won't know what the optimal asset allocation to a portfolio of assets is (though the tangency portfolio from mean-variance optimization, in hindsight, may come pretty close), but you know one allocation is better than another if it generates a better Sharpe ratio in hindsight. Hence reinforcement learning has been applied to many optimization problems in finance such as portfolio optimization and optimal executions (Dixon et al., 2020; Sood et al., 2023). In recent years, successful reinforcement learning schemes often use deep neural network to learn the optimal actions automatically, instead of manually crafting them or updating a table of actions. This is called Deep Reinforcement Learning (DRL).

DRL may seem complicated, but we can implement it in a way very similar to supervised learning. Recall we train a neural network via supervised learning by presenting samples (minibatches) with labels to it, and the labels are used to compute the (negative) log likelihood of the sample given the network parameters (weights). The negative log likelihood can be viewed as a cost function J to be minimized via gradient descent and backpropagation by iteratively adjusting the network parameters as we present training samples to the network. In the case of DRL, we don't have labels and their log likelihood, but we can still compute the cost J given the output of a network. For example, in portfolio optimization, the output of a network can be the asset weights in a portfolio, and the cost may be the negative Sharpe ratio for the next 12 months of that portfolio given such weights. Following Cong et al. (2021), if we are planning monthly rebalance of a portfolio, we can sample every month-end t in the train set and feed the features obtainable at t as input into a network, with the cost J set equal to the negative Sharpe ratio from t to $t + 12$ months. Once the network parameters are trained, it can output the optimal weights every month going forward, given fresh input features. Of course, the network can be retrained periodically

with new data as we go forward as well. Cong et al. (2021) applied this scheme to a stock portfolio and found that it generates a Sharpe ratio of close to 2 in the test set.

In supervised learning, the outcome of the predictive model is called the label or target, but in reinforcement learning, the outcome of the optimizer model is called the “reward.” In general, the cost J is the sum of all the rewards earned over all the samples. You may have heard there is a difference between online vs. offline reinforcement learning. Offline RL is needed when we cannot simulate what reward we would have earned by a different action, and all we have to train on are historical actions actually taken and their rewards. (For example, we wouldn't know if a consumer would actually buy a product if we were to lower its price when conducting price optimization RL.) However, in financial applications such as portfolio optimization, we can definitely simulate what would have happened if we took a different trading decision, since the rewards were based on historical prices. Hence almost all financial RL is online learning.

Note there are other more complicated ways to implement DRL, with names such as Q-learning or Proximal Policy Optimization (PPO). These methods deal with the possibility that sometimes we want to explore less immediately rewarding action to optimize for long-term rewards (the “Exploit vs Explore” tradeoff), and sometimes we want to avoid unstable behaviors during training. In particular, the function that outputs an action based on the current state (called the “policy function”) may instead output a probability distribution of possible actions instead (or more often, the parameters of the probability distribution). See the excellent free online course on RL by Hugging Face (<https://huggingface.co/learn/deep-rl-course/>) for further details. Also, in case we gave the impression that RL based on machine learning can only be implemented via deep neural network, the paper by Fuhrer et al. (2024) that uses gradient-boosted trees should correct that.

3.4 Data Engineering

Without data, there is nothing for ML to learn from. Even with data, there is a considerable amount of expertise needed to vet them properly. We will highlight some common issues with financial data here. Most of these issues are pertinent to training, not inference/prediction.

3.4.1 Unique Company Identifiers

This is mainly a problem for data associated with individual companies and the data related to them (e.g., prices of their stocks, bonds, and options, fundamental information on the company itself, etc.). Different data vendors often use different identifiers for these companies. Even within one data file, the identifier for a company may change over time, and two companies may use the same identifier at different times. This is especially true if the identifier is just the ticker symbol. We should use data that has a “securities master” file: it should have a unique identifier for a security throughout its history, and cross reference other commonly used identifiers such as ticker symbols, cusips, etc. for different periods that various other datasets may use. The goal is to link all the information related to this security from different datasets together *for the same date*, and use that as features or labels for ML.

A related problem is how we should handle securities that are highly correlated. By that we don’t mean Exxon vs Chevron stocks. We mean companies that issue multiclass stocks such as BF.A and BF.B. They are almost identical in their daily returns. If one construct features from their returns, we would get highly redundant features. Of course, redundant features wouldn’t necessarily be a problem if we use ML algorithms like CART, or feature selection techniques such as CMDA. But still, it is good hygiene to eliminate such

duplications to avoid confusion downstream in the investment process.

3.4.2 Dividend and Split Adjustments

In general, we should treat returns of an asset as features or label, and not prices. The price level is arbitrary and doesn't add much information. It is also nonstationary and price features at one time cannot in general be used to predict labels at another time in the future. You might think that if there is a stock split, it might be a bullish signal, and a reverse split is a bearish signal, but you can code such events as categorical features separately. Similarly, you may think that a large adjustment of prices due to a large dividend is bullish. But again, you can use dividend yields as a standalone feature. There is no need to infer that from the price level. We have discussed in [Chapter 1](#) already how to compute returns that take into account such corporate actions. More discussion can be found in Chan (2021).

3.4.3 Survivorship Bias

Survivorship bias is a data problem that you can't discover by examining the data itself. You won't know what you are missing. For example, in testing a trading strategy on a portfolio of stocks with market cap greater than \$1B, you would need a historical database of market cap of all the stocks that existed at a certain time (called a "Point-in-Time" database), not just a database of market cap of stocks that exist now. Similarly, if you want to test this strategy on S&P 500 stocks, you will need a historical database of the S&P 500 index constituents. The consequences of having survivorship bias in your training data and what data vendors provide survivorship-bias-free data is discussed in Chan (2021).

3.4.4 Look-ahead Bias

If someone mis-specified the timestamp of a *feature* obtained at time t to be t' , which is earlier than t , we have a look-ahead bias if we use it to predict a label at time τ such that $\tau < t$. Equivalently, if someone mis-specified the timestamp of a *label* obtained at time t to be t' , which is later than t , we have a look-ahead bias if we use a feature obtained at time τ such that $t < \tau$. There are subtle ways that look-ahead bias can creep into data, even those from highly reputable data vendors. For example, if a company announced earnings e at time t , but subsequently revised the earnings to e' at time $t' > t$, the data must not contain the entry (t, e') , though some datasets assuredly do. Similarly, if the nonfarm payroll p was announced by the Bureau of Labor Statistics at time t , but subsequently revised to p' at time $t' > t$, the data must not contain only the revised entry (t, p') .

There is another interesting case of look-ahead bias in data. Some event-driven trading strategies are based on the earning announcement date t (not the announced earning itself). For example, it may specify that “At the market close before the earning announcement at time t , buy the stock if it has positive 1-day return.” If you backtest such a strategy based on the earnings announcements calendar you scrape from Yahoo Finance or the like, you will probably have introduced look-ahead bias. The reason is that most databases only capture the *actual* announcement date, not the *expected* announcement date, whereas the expected date is what’s needed to backtest an event-driven strategy like the one we described. Companies often revise their expected announcement date or time, and it is crucial to capture the Point-in-Time expected announcement date/time. As far as we know, such data are only available from Wall Street Horizon.

One other interesting case of look-ahead bias occurs in futures price data. But in this case, the fault lies with the interpretation of the data, not with the data itself. For example, our favorite futures data vendor CSIdata provides a futures settlement price as well as the spot price of the underlying. To be specific, let's say on 2019-06-06, the 2019 June contract of 6M future has settlement price = 0.05082, and spot price = 0.05086. Is it correct to conclude that this future is in backwardation (see Chan, 2013 for the definition) on that date? No! The futures settlement price is obtained at 15:00 ET at the CME, while the spot price is obtained (by CSIdata) at 17:00 ET.

Perhaps you think that by using intraday data, we can safely overcome any look-ahead bias issues with futures or other price data. That may not be enough. Suppose we want to backtest a futures calendar spread strategy using 1-minute OHLC trade bars. That still won't work because the last trade of one leg does not usually occur at the same time as the last trade of the other leg. Backtesting such trade bars in a calendar spread essentially assumes you know the later trade price when we execute the earlier trade. To overcome this problem, we have to use best bid-offer quotes instead of trade prices to form the minute bar of the spread, since the last quote of each leg remains valid till the end of that minute bar.

Such subtleties will surely escape your well-paid data scientist unless they are steeped in financial data science. Even if the look-ahead bias is recognized, it is often hard to find archived data that is bias-free. And even if found, their histories tend to be short. Nevertheless, that's the only data that is fit to be used for ML.

3.5 Feature Engineering

We already wrote that unlike ML applications in images, videos, speech, and texts, raw price data won't work as

features for applications in finance. Sometimes extensive features engineering is required. Firstly, we need to ensure that a feature is *stationary*. This usually means time-differencing a time series instead of using its levels, such as taking differences of the logarithm of a price series to create a log returns series, taking the percentage differences of a GDP series to create a GDP-changes series, and so on. Secondly, we need to properly merge time series with different sampling frequencies. Thirdly, there are different ways to handle similar features from a large pool of similar assets. Lastly, if we purchase such features from a vendor and do not create them from raw data ourselves, extra precautions are needed. We will drill down on these issues next.

3.5.1 Stationarity

It is obvious that we cannot use price levels of stocks as either features or labels. The price may be \$10 in year 2000, and \$10,000 in 2024 if we adjust for splits, and we simply cannot train a ML model based on 2000 prices and expect inferences to work with 2024 prices. We should convert price to return series. But there are subtler time series where it is not clear if they are stationary. For example, is the spread between WTI crude oil futures and Brent crude oil futures stationary? Is the spread between MSFT and GOOG stationary? To determine those questions, we have to run a stationarity test such as the **augmented Dickey-Fuller test** (ADF) test to find out. If not, we can either time-difference these spreads or construct a spread that is stationary to begin with. Ernie has discussed many ways to construct a stationary time series out of multiple nonstationary time series in Chan (2013).

3.5.2 Merging Time Series with Different Frequencies

The frequency that the ML model should make predictions is up to the user. For example, do they want to trade at daily intervals or weekly intervals? Once this frequency is determined, we have to convert features data that may have lower or higher frequency to conform to the label frequency. The former is usually done by either filling the features with NaN when they don't have an updated value, or filling forward with their last updated value. The former may be appropriate if it is a categorical event feature such as "Was the nonfarm payroll announced yesterday?" and the latter may be appropriate if it is a numerical feature such as, "What was the most recent percentage change in nonfarm payroll?" Which method to adopt needs to be determined on a case-by-case basis. When the features frequency is higher than the label frequency, we must use a down-sampling method. The default is just using the last updated value of the feature just before the time stamp of the label. However, in some situations it may be better to use some kind of average across all the values during the intervals between two consecutive labels. For example, if we are using a price series as raw data, the former choice means sampling the last prices, while the latter choice may mean taking the Volume-Weighted Average Price (VWAP). After down-sampling these prices, we can proceed to compute return features. In addition to taking averages, other down-sampling techniques include taking order statistics, like the max or min, familiar to investors as taking the high or low of a price bar. There are many other signal processing techniques one can apply. See López de Prado (2018) for more details. But how do we down-sample categorical features? To down-sample binary categorical features with values 0 or 1, we can either treat them as real-valued features and apply the aforementioned techniques or we can apply the logical AND, OR, XOR operators. To down-sample a multiclass categorical feature, we can first convert

it into a binary feature via one-hot encoding. For example, if we have a three-class categorical feature, it can be converted to three binary features, the first feature is whether the original feature is of class A, the second feature is whether the original feature is of class B, and the third feature is whether the original feature is of class C. A k -class categorical feature can thus be represented by k binary features.

3.5.3 Time-series Versus Cross-sectional Features

If we want to build a ML model to predict returns or other attributes of several assets in the same category (e.g., S&P 500 stocks), we can surely use features that are specific to each of these assets (e.g., past earnings yield of a stock). These features are called cross-sectional features because they span the cross-section of the asset universe. However, what if we want to build a ML model to predict returns or other attributes of the portfolio of assets as a whole? While you may be right to think that MSFT's earnings would portend the returns of the SPY, in general each of these cross-sectional features will be an extremely weak predictor, and it is very inefficient to use so many of them as features. It is better to aggregate these cross-sectional features into something like a summary feature, similar to the way we down-sample high-frequency features to low-frequency features in the previous section. The aggregation method isn't new: it was based on Fama-French's seminal paper Fama and French (1995). First, we sort the assets based on the cross-sectional feature in question (e.g., earnings yield). Then we form a long-short portfolio based on this ranking (e.g., the long portfolio contains stocks in the top decile of earnings yield, and the short portfolio contain stocks in the bottom decile.) This is called the "hedge" portfolio. The resulting time-series feature is simply the return of this

portfolio over some past period. This way, we have collapsed 500 cross-sectional features (for an S&P 500 portfolio) into just one time-series feature. The most famous of these time-series features are none other than the Fama-French factors discussed in [Chapter 2](#). For more discussions, see Pik et. al. (2025).

3.5.4 Validating Third-party Features

Sometimes a feature series is created by a third-party vendor and not by ourselves from raw data. How would we know if the feature is computed correctly? Case in point: we once purchased historical and live sentiment data on financial news articles from a vendor. The scores look reasonable, and we proceeded to train a ML model using this data. Strangely, upon further examination of the time-series data by taking their min and max, we found the chart shown in [Figure 3.12](#).

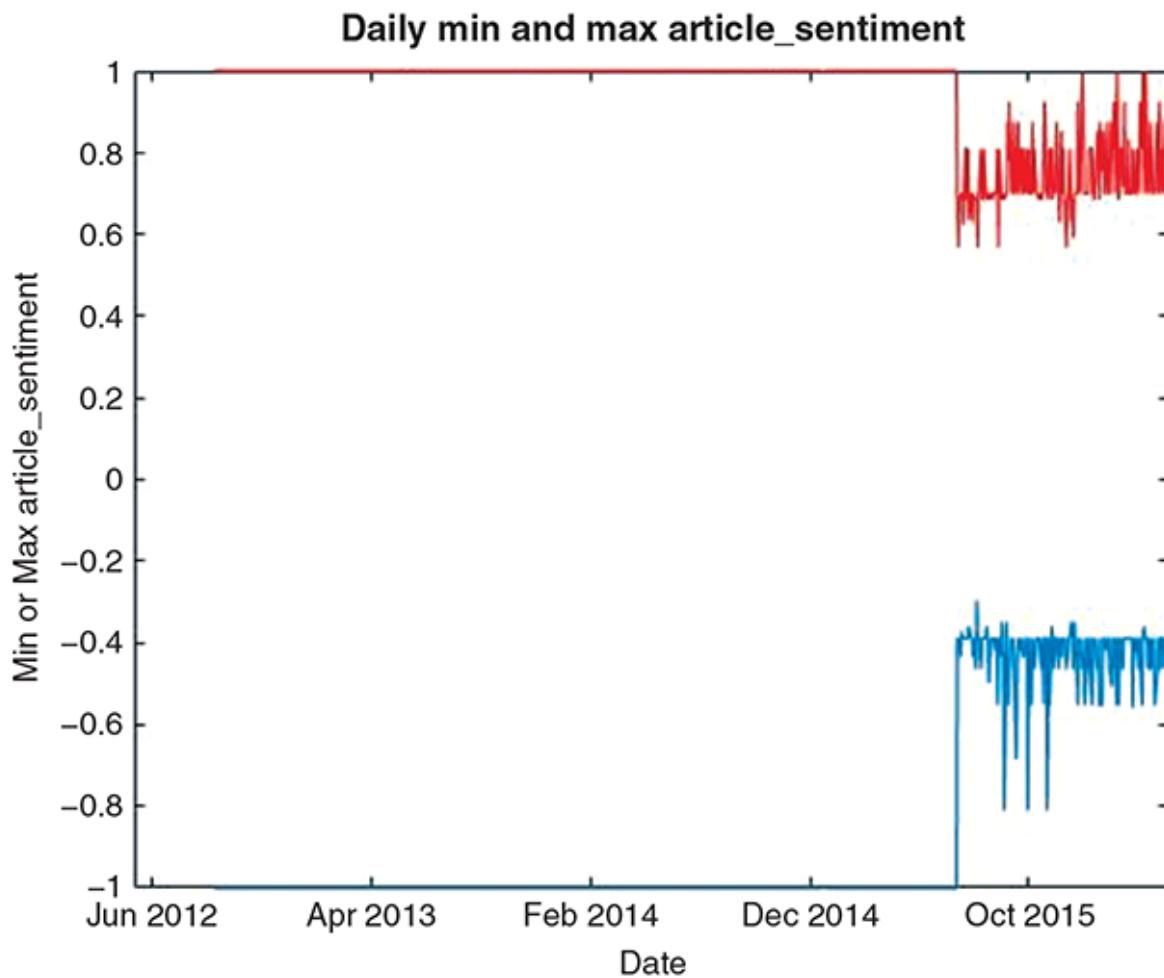


Figure 3.12 Daily minimum and maximum sentiment scores that show structural breaks in the data generating process.

There is a clear structural break in the data in 2015. The vendor admitted that they have changed their methodology for computing sentiment scores. Of course, that rendered all the historical data prior to 2015 useless. But more ominously, how would we know if they won't change the methodology again going forward?

The lesson from this experience is that careful and extensive EDA (Exploratory Data Analysis) is needed when we are purchasing features from third parties—ideally before you commit to an annual contract.

3.5.5 Generative AI as a Feature Generator

The example we quoted above in relying on third-party vendor for features is sobering. Back in the days, we purchased those sentiment data because it wasn't easy to process unstructured data such as text and turn them into numerical/structured data such as sentiment scores. However, with Generative AI, this has become much easier. We used this as example of Generative AI in [Chapter 9](#) on how we can compute sentiment scores, and whether they add any value to an investment process.

Of course, whether or not the features are created by ourselves or by a vendor, we can always use Generative AI to conduct no-code EDA. Further, as demonstrated in the previous chapters, we can even ask Generative AI to create ML codes that can merge the features and labels into a file, build a ML model, and assess the performance.

3.5.6 Features Importance Ranking and Selection

Recall that one of benefits of L1 regularization is that some features will be eliminated altogether as input to a ML model. Similarly, tree-based models typically only select a small subset of features for their nodes before stopping at the leaves. If one is merely concerned about test set performance, we typically do not need to reduce the number of features any further for CART. However, there is sometimes a need for features importance ranking and selection not for performance improvement, but for model interpretability to humans and for features validation (i.e., sanity checks). The power of ML is that we can start with a large features set, without regard to whether these features are insignificant or redundant. But this inclusiveness is at the expense of interpretability because humans will have no idea how the model arrives at its inferences. Features importance ranking alleviates that problem. For a real financial example,

see Table 7 of Chan et al. (2023). Sometimes, when the top feature makes no sense whatsoever to us, we should investigate whether that feature has embedded look-ahead bias or is otherwise corrupted. Hence features importance ranking also serves as a validation method.

As we discussed before, cMDA is our favored features importance ranking method. We have explained the *c*—clustering—part. The MDA (Mean Decrease in Accuracy) part is better known. The idea is that if we randomly permute the samples of a feature (i.e., randomly permute the rows of a column of a data/design matrix), the model should generate lower predictive accuracy on a validation set. (We specified validation set because it is almost guaranteed to generate lower predictive accuracy if we corrupt the train set data *whether or not* that feature is really important.) If we make several of these permutations and compute the mean of the decrease in accuracy (or log likelihood), we can use this mean as a metric to rank the features' importance. The bigger the decrease, the more important is the feature.

There are other features importance ranking methods besides MDA or cMDA, notably Shapley Additive Explanations (SHAP), and Local Interpretable Model-Agnostic Explanations (LIME) (Man and Chan, 2021a), but none of them produces as interpretable results as cMDA. Cong et al. (2021) also invented another interpretable AI technique that approximates the ML model by a polynomial function of the most important input features.

3.6 Conclusion

We have presented almost everything you need to know about the various techniques of machine learning as relevant to asset management but are afraid to ask (because some of the details seem very trivial but are very critical for success). The jury is still out on whether tree-based models, deep

learning, or reinforcement learning is the best method for financial features and applications.

Can Generative AI help in a different way? Can it create trading strategies unlike (i.e., uncorrelated with) those we have seen before using expert systems and conventional machine learning? Can it become a discretionary trader, a “George Soros on a chip?” We will find out in the rest of the book.

Part II

Deep Generative Models for Trading and Asset Management

Chapter 4

Understanding Generative AI

Generative models represent a fascinating frontier in the advance of artificial intelligence. These models have the ability to understand and generate complex, high-dimensional data. This capability not only opens the door to numerous practical applications but also offers deeper insight into the underlying structure of data. This chapter explores the essence of generative models, their applications, and the differences with discriminative models that many quant finance professionals have been familiar with. We will also provide a taxonomy to navigate the landscape of generative models ([Figure 4.1](#)).

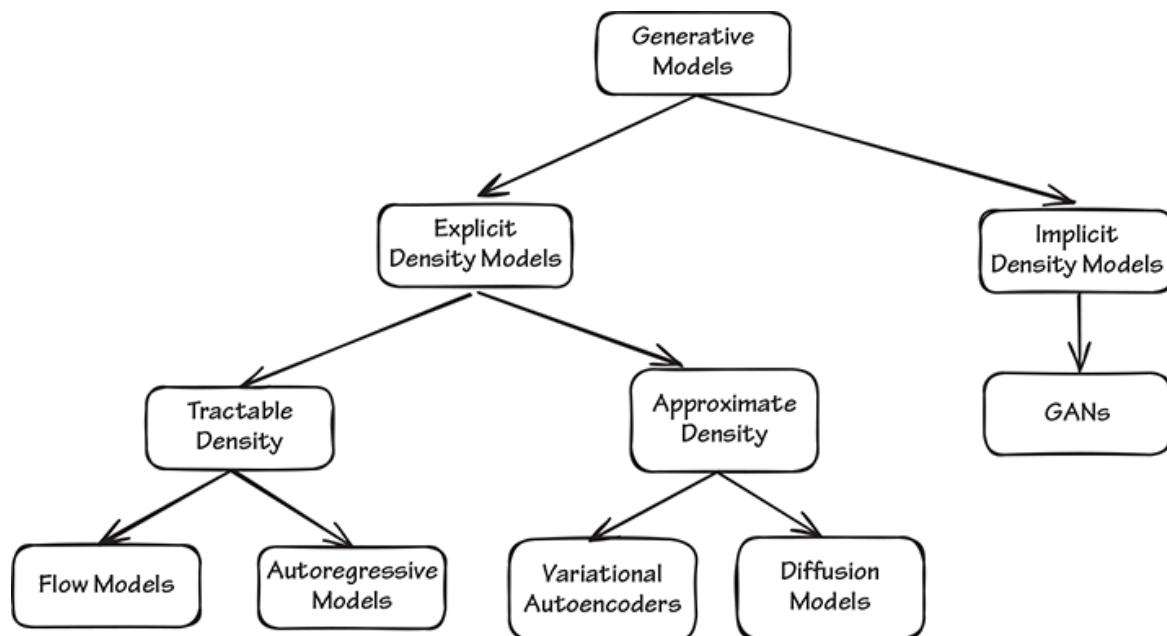


Figure 4.1 Model taxonomy.

4.1 Why Generative Models

Generative models are pivotal for several reasons. Firstly, they enable machines to generate new data instances that resemble the data they have encountered during training, effectively learning the distribution of the dataset. This ability goes beyond merely replicating observed data; it's about understanding the underlying structure of the data to generate new instances. Secondly, generative models facilitate unsupervised learning tasks, where the machine learns common patterns in the data without needing labeled data. This aspect is crucial for exploiting the vast amounts of unlabeled data available today and avoiding the costly and time-consuming process of manual data labeling. Lastly, they aid in understanding data distribution, enabling applications for anomaly detection, data compression, and more.

We want to disabuse the reader of the notion that Generative AI is only useful for understanding unstructured data such as text or images, or that it is only useful for generating synthetic data for simulation purposes. As the following section illustrates, it can be crucial for the usual supervised learning tasks using discriminative models that most quant finance professionals are primarily interested in.

4.2 Difference with Discriminative Models

At the heart of machine learning lies the quest to model and make predictions from data. Discriminative and generative models represent two approaches to this quest, differentiated by the type of probability distribution they aim to learn from the data.

Discriminative models focus on learning the probability of a label or target y given an input x , denoted as $p(y|x)$. These

models excel in classification or regression tasks, where the goal is to correctly predict the target. They are closely associated with supervised learning tasks, where the models are trained on a labeled dataset, learning to map inputs to targets. Despite their powerful ability to determine the most probable target for a given input, a critical limitation of discriminative models is their inability to assess the likelihood $p(x)$ of the input data itself. This design limitation makes them susceptible to adversarial attacks or misinterpretations of novel or outlier data.

Generative models, on the other hand, aim to model the distribution of data itself, learning either the joint probability distribution $p(y,x)$ or the unconditional probability distribution $p(x)$. Unconditional generative models attempt to learn the probability distribution of the input $p(x)$, enabling them to generate new data instances that resemble the training data. As an example, let's suppose we want to model the SPY ETF return data and decide to approximate its distribution using a Gaussian Mixture Model (GMM), one of the simplest generative models. This model not only allows us to generate new data instances but, as we will see in [Chapter 6](#), it also captures some underlying structure of the data that can be used for applications like regime detection. In general, generative models provide a deeper understanding of the data's structure, enabling applications like data generation and outlier detection.

Conditional generative models represent a hybrid approach, learning to generate data x conditioned on specific values of y , such as class labels, denoted as $p(x|y)$. These models combine the generative capability to produce new instances with the discriminative power to condition these instances on particular attributes or classes, making them more robust to adversarial attacks or misinterpretations of novel or outlier data. As an example, commonly encountered, consider models like ChatGPT, which can generate new data like text,

images, or source code conditioned on input prompts provided by the user.

To illustrate the difference between discriminative and generative models, let's consider the classic dog vs. cat classifier example. A discriminative model learns $p(y|x)$ —the probability of a label y (dog or cat) given an image x . At inference time, it takes an input image and predicts the probability that it belongs to each class. A conditional generative model, on the other hand, learns $p(x|y)$, modeling the probability of an image x given a specific class y .

This model can also be used as a classifier, known as a generative classifier, by comparing the probability of an image x under $p(x|y = \text{dog})$ versus $p(x|y = \text{cat})$. However, unlike discriminative models, it can also generate new images conditioned on a label. For example, when conditioned on *dog* ($y = \text{dog}$), the model can generate an image of a dog by sampling from $p(x|y = \text{dog})$.

In this sense, a discriminative model maps the input image x to the output class label y , while a generative model works in reverse, generating x from a given y . Additionally, in generative modeling, y is not restricted to class labels—it can also represent attributes or features.

The distinction between generative and discriminative models, which may initially seem somewhat disconnected, is rooted in Bayes' rule of probability. This relationship is illustrated by the equation

$$p(x|y) = \frac{p(x)p(y|x)}{p(y)} \quad (4.1)$$

where $p(x)$ represents the unconditional generative model, $p(y|x)$ corresponds to the discriminative model, $p(y)$ is the marginal probability over the labels, and $p(x|y)$ is a conditional model of x given y . By combining the strengths of

both unconditional generative models and discriminative models, we can construct conditional generative models, which can be used not only for conditional data generation but also for supervised learning tasks, such as classification.

4.3 How Can We Use Them?

These models are designed with some key core abilities or objectives in mind, which open a world of possibilities for innovation, creativity, and efficiency across various domains. Following is a summary of these characteristics, which we will explore further in next chapters.

4.3.1 Probability Density Estimation

These models can approximate *high-dimensional*, multimodal distributions of data. Once we have access to this versatile object, we can develop and solve a wide range of applications. Notable examples include multivariate time series forecasting, as illustrated in [Chapter 3](#), Value at Risk (VaR) applications for risk management; anomaly detection, and classification tasks.

Finance professionals are already familiar with high dimensional probability distributions of returns, but they are typically modeled as the multivariate versions of simple parametric models such as the multivariate Gaussian or t-distribution. More advanced models such as copulas attempt to capture the nonlinear nature of the co-dependency between variables, but at the heart of them are still simple parametric models. Generative AI, on the other hand, allow much more complex, nonlinear, distributions that cannot be expressed in simple parametric forms.

4.3.2 Generating New Data

At the heart of generative models' capabilities is their power to create. They can produce highly complex, high-dimensional objects, like realistic images, videos, text, computer code, multivariate time series, etc. This makes them invaluable across applications like entertainment, marketing, productivity tools, and risk management. Their ability to generate data is often indistinguishable from humans. This ability goes beyond simply crafting content that is often indistinguishable from human-created work, as seen in widely available examples in the media at the time of writing.

New generated data can be used for the following:

- Improving data quality and availability: Generated data can address challenges related to data scarcity and quality. These models can augment existing datasets, especially in fields where data collection is expensive or impractical, by creating new data instances and enhancing datasets for downstream learning tasks.
- Facilitating discovery and decision-making: one of the most promising applications of generative models lies in their ability to simulate and predict complex, high-dimensional structures and phenomena. In the pharmaceutical industry, for example, these models can predict molecular structures, potentially leading to groundbreaking drugs. Additionally, they can simulate environments for AI training, such as for reinforcement learning applications which are known to be very data hungry, therefore reducing the need for costly and time-consuming real-world data collection, accelerating research and development across fields as diverse as autonomous driving and urban planning.

4.3.3 Learning New Data Representations

The goal of finding new data representations, or representation learning, is to discover an effective way to represent data. Ideally, this new representation should possess desirable properties, such as lower dimensionality compared to the data space or “independent” axes of variation. This new representation serves as a compressed form of the data that can be used for solving downstream tasks such as prediction, anomaly detection, or data generation (see Torralba et al., [2024]).

Many finance professionals are already familiar with representation learning disguised in other names: factor models, PCA, etc. Some of which we already discussed in [Chapter 3](#), but we will reframe them within the Generative AI framework. This allows us to potentially improve on them at a fundamental level.

The power of representation learning allows you to combine multiple models as building blocks to develop complex applications. We will explore examples of using them, particularly in [Chapters 9](#) and [10](#), where they are applied to train a model specialized in sentiment analysis for financial data.

4.4 Illustrating Generative Models with ChatGPT

To better understand how we can use generative models for computing probability densities and generate new data, let’s examine a concrete example in Natural Language Processing (NLP). For simplicity, we’ll set aside specific model details and instead focus on a pre-built model, ChatGPT, examining its density estimation, sampling, and representation learning capabilities.

Let's consider a sequence of random variables where each variable represents a word or token in a sentence. (For this example, we'll ignore the distinction between words and tokens, which will be covered in [Chapter 5](#).) Each element in the sequence takes values from a finite vocabulary—a discrete set. In ChatGPT-4o, for example, the vocabulary consists of 16,384 unique tokens (See OpenAI documentation at <https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>), which include common words, word fragments, and characters.

A probabilistic model for a sequence is defined by its joint probability distribution: $p(x_1, \dots, x_N)$

4.4.1 Language Modeling

Modeling this joint probability over a sequence of words/tokens is what is known as the Language Modeling (LM) task in NLP. One of the most straightforward ways to express this joint distribution is by applying the product rule of probability:

$$\begin{aligned} p(x_1, \dots, x_N) &= p(x_1) && (4.2) \\ &\quad \times p(x_2|x_1) \\ &\quad \times p(x_3|x_1, x_2) \\ &\quad \dots \\ &\quad \times p(x_N|\mathbf{x}_{1:N-1}) \end{aligned}$$

Models that exploits the right-hand side of [Equation 4.2](#) to represent joint distributions as a sequence of conditional probabilities like this are known as autoregressive models. As it turns out, ChatGPT is an instance of an autoregressive model, specifically a transformer-based model, which we will explore in [Chapter 5](#).

From [Equation 4.2](#), we see that each conditional probability models the likelihood of the next word/token given the previous ones. In other words, ChatGPT performs probabilistic “one-step-ahead” forecasts, where each prediction is based on past values, or past context.

A key question is: How many previous words can ChatGPT consider when making predictions?

This is defined by what is known in NLP as the context window. As of the time of writing, the context window of ChatGPT-4o is 128,000 tokens, meaning it can analyze up to 128,000 previous tokens to estimate the probability of the next one (<https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo>). This significantly enhances its ability to capture long-range dependencies in text.

4.4.2 Sampling: How Generative Models Create New Data

Language models are generative models, meaning they can generate new sequences by sampling from the learned probability distribution. The process follows these steps:

- Generate the first token, $x_1 \sim p(x_1)$
- Generate the second token, $x_2 \sim p(x_2|x_1)$
- Continue generating tokens, $x_n \sim p(x_n|x_1, \dots, x_{n-1})$

Since sampling is sequential, each newly generated token depends on previously generated ones, making the process relatively slow for autoregressive models.

4.4.3 Conditional Language Generation: Asking ChatGPT a Question

Just as we can model a sequence of words/tokens with a language model, we can also generate text conditionally. That is, we can condition the generated output on some given context, such as a question or a prompt, which we will denote in the following example as y . This process is used in question answering, where the model generates responses conditioned on user input.

Mathematically, the conditional version of the product rule is given by:

$$\begin{aligned} p(x_1, \dots, x_N | y) &= p(x_1 | y) \\ &\quad \times p(x_2 | x_1, y) \\ &\quad \times p(x_3 | x_1, x_2, y) \\ &\quad \cdots \\ &\quad \times p(x_N | \mathbf{x}_{1:N-1}, y) \end{aligned} \tag{4.3}$$

Sampling follows the same sequential process described in [Equation 4.3](#):

- Generate the first token $x_1 \sim p(x_1 | y)$
- Generate the second token $x_2 \sim p(x_2 | x_1, y)$
- Continue generating tokens $x_n \sim p(x_n | x_1, \dots, x_{n-1}, y)$

To illustrate this, we first ask ChatGPT to generate humorous responses to our prompts. Now, let's consider the following user prompt:

User Prompt

What do Mark Zuckerberg and Elon Musk have in common?

When ChatGPT generates a response to given prompt, it follows the process outlined above, predicting one word at a time based on the probabilities it assigns to possible next tokens conditioned on previous tokens and the prompt (y).

Here's an example response generated by ChatGPT-4o:

ChatGPT Response

They both have enough money to buy a small country ... or at least a really nice pizza place!  

Now, how did ChatGPT-4o generate this response?

To understand its output, we used the OpenAI API to access both the sequence of tokens generated by ChatGPT-4o and the probabilities associated with each token in the vocabulary. Since ChatGPT-4o has a vocabulary size of 16,384 unique tokens, visualizing the entire probability distribution over the next word would be impractical. Instead, we display only the top 20 most probable tokens based on their probabilities.

Step-by-Step Breakdown: How the AR Model Generates a Response

1. Computing the first word's conditional probabilities
 - Conditioned on the prompt, the model calculates the probabilities of the first word.
 - As shown in [Figure 4.2](#), the top choices include "They", "They're", and "Both".

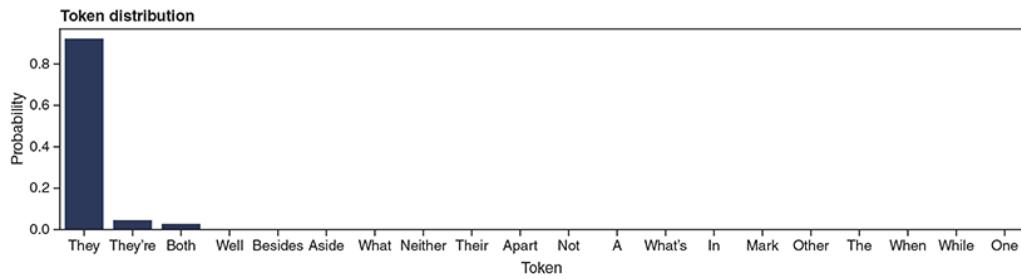


Figure 4.2 Conditional probability of the first token given the prompt y : $p(x_1|y)$.

- The model samples “They” (which happens to be the most probable word in this case).
2. Computing the second word’s conditional probabilities
- Next, the model predicts the probabilities of the second word, conditioned on the prompt and the first word (“They”).
 - As shown in [Figure 4.3](#), the top choices include is “both”.

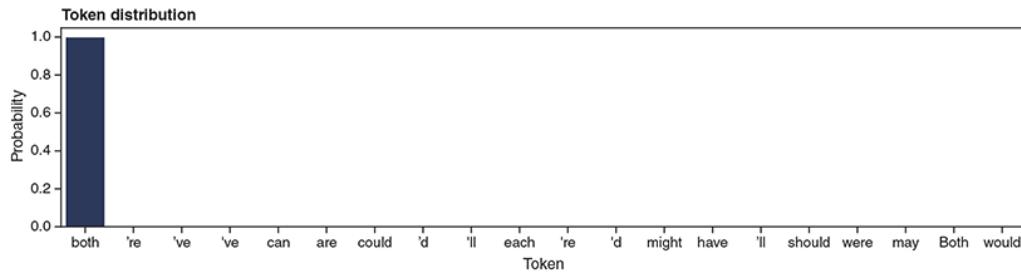


Figure 4.3 Conditional probability of the second token given the prompt y and previous token x_1 : $p(x_2|x_1, y)$.

- The model samples “both.”
3. Computing the third word’s conditional probabilities
- Now, the model predicts the next word, conditioned on the prompt, “They” and “both”.

- As shown in [Figure 4.4](#), the most probable words are “have” “now” and “think.”

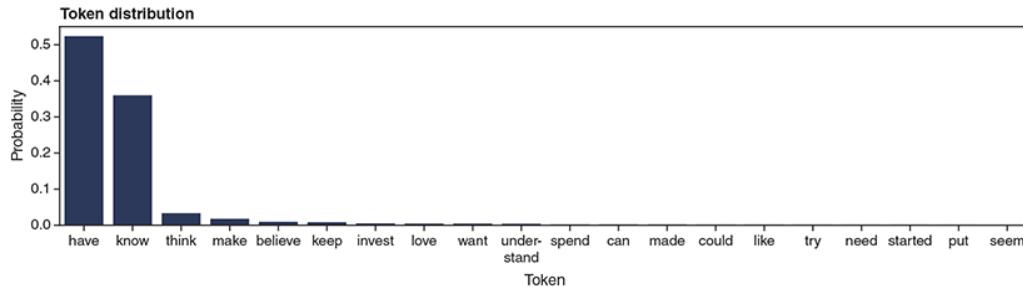


Figure 4.4 Conditional probability of the third token given the prompt y and previous tokens

$$x_1, x_2: p(x_3|x_1, x_2, y).$$

- The model samples “have”.
4. Computing the fourth word’s conditional probabilities
- The model then predicts the next word, given the prompt, “They”, “both”, and “have”.
 - As shown in [Figure 4.5](#), the top choices include “enough”, “more”, “a”, and “billions.”

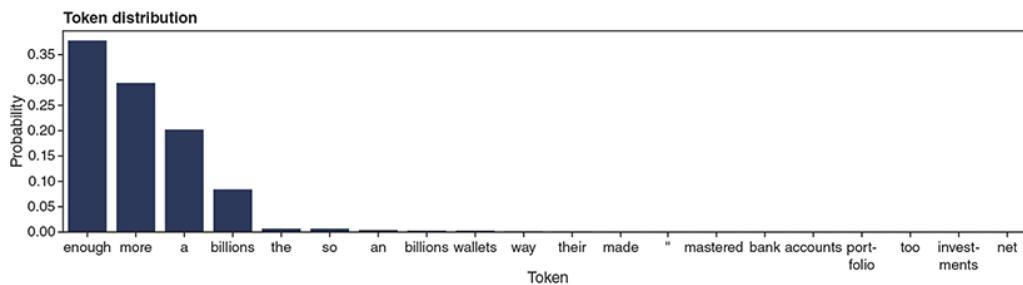


Figure 4.5 Conditional probability of the fourth token given the prompt y and previous tokens $x_1, x_2, x_3: p(x_4|x_1, x_2, x_3, y).$

- The model samples “enough.”
5. Computing the fifth word’s conditional probabilities

- The model continues, now predicting the next word given the prompt, “They”, “both”, “have”, and “enough”.
- As shown in [Figure 4.6](#), the most probable words include “money”, “wealth”, and “cash”.

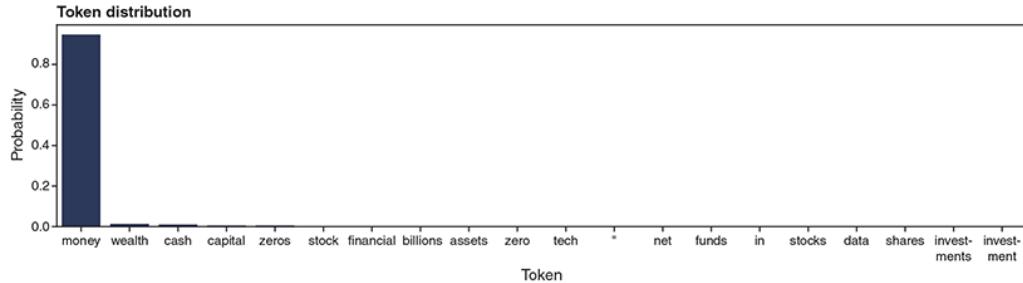


Figure 4.6 Conditional probability of the fourth token given the prompt y and previous tokens x_1, x_2, x_3, x_4 : $p(x_5|x_1, x_2, x_3, x_4, y)$.

- The model samples “money.”

6. The process continues

- The model sequentially predicts each word based on the previous ones and the original prompt until it forms a coherent response.

Each word generated depends on the probability distribution estimated by the model at that step. In this case, data generation is performed using the probability density function. This process illustrates how autoregressive generative models perform sampling in a real-world scenario.

Beyond data generation, probability density estimation can also be applied to forecasting the next most probable token or answering questions about the probability of rare words in a text. This becomes even more relevant when shifting from modeling a sequence of words to modeling a sequence of financial returns, a topic we will explore in [Chapter 5](#). These techniques can be used for applications such as anomaly

detection or for computing key financial metrics like Value at Risk (VaR).

As you can see, generative models can function as simulators, which are widely used in finance. A well-known example is option pricing, especially for complex option payoffs using Monte Carlo techniques.

In the example above, the sampling process selects the word/token from the vocabulary with the highest probability at each step. However, in practice, ChatGPT may use more complex sampling strategies (such as temperature scaling or top-k/top-p sampling) to generate more diverse and contextually relevant responses.

If you prefer video lectures, a tutorial based on this section, presented by the authors of this book, can be found on YouTube. You can search for: Trading Using LLM | Generative AI & Sentiment Analysis for Finance | Webinar on the QuantInsti Quantitative Learning YouTube channel.

As you might have noticed with this example, even though these models are highly effective at generating new data, a key question arises: how can we evaluate the quality of the generated data if every run produces different results? As we saw in [Chapter 3](#), the answer is clear for applications like forecasting, where you can compare the true realized values against your predictions using a chosen metric, such as mean squared error.

In our experience developing applications, one of the best ways to evaluate the quality of generated data is to use it for training a downstream model to solve a specific task. By combining the real data with the newly generated data, you can train the model and assess whether its performance improves. However, many metrics for evaluating the quality of generated samples are available in the literature, and we will use several of them throughout the book to assess the performance of these models in data generation.

In the accompanying notebook [*Synthetic Data Evaluation—Notebook*], we will walk through an example to explain the different metrics we will use in our experiments in the book.

This example highlights how ChatGPT, as an autoregressive model, generates text by modeling probability distributions over sequences of tokens.

4.4.4 A Few Words on Representation Learning with ChatGPT

As mentioned earlier, representation learning is about discovering effective ways to encode data that facilitates solving downstream tasks such as prediction, anomaly detection, data generation, among others. Surprisingly, ChatGPT also has the capability to compute new representations of input data in the form of vectors, also known as embeddings or vector embeddings in the NLP literature. These embeddings capture semantic information about words and concepts, enabling us to manipulate and “understand language” by performing mathematical operations with these objects. We will explore embeddings in more details in [Chapter 5](#).

OpenAI provides access to ChatGPT embeddings through its API, allowing users to use these representations for solving applications.

As a use case, imagine you are an investor or a discretionary trader tracking a set of companies where you either hold positions in your portfolio or are considering an allocation. Let’s say that you frequently analyze 10-K and 10-Q filings, earnings transcripts, and financial analysis reports to make informed decisions. However, retrieving relevant information across multiple documents can be very time consuming.

Wouldn’t it be useful to have a way to extract specific insights from these sources? For example, imagine asking a question

like “What are the risks associated with Company XYZ?”, and having a system that retrieve relevant documents where these risks are mentioned. To solve this kind of problem, text representations might come to rescue.

To approach this, the first step is to build a knowledge database using vector embeddings. Here’s how:

1. Chunking the Documents: Break each financial report or document into manageable chunks (e.g., sentences, paragraphs, or pages).
2. Generating Embeddings: Use OpenAI’s Embeddings API to generate vector representations for each document chunk. Store these embeddings in a database.
3. Performing Semantic Search: When a user submits a query (e.g., “What are the risks associated with Company XYZ?”), converts the query into an embedding. This embedding is then compared against stored embeddings in the database using a similarity measure, such as cosine similarity. Then, you can retrieve the top K most relevant text chunks, such that you get the most contextually relevant information for your query.

This embedding-based retrieval approach is a core building block for more advanced techniques like Retrieval-Augmented Generation (RAG), see

<https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts>.

RAG enables Large Language Models (LLMs) to generate responses based on external knowledge by enriching the input query, allowing you to customize your LLM for your specific domain without fine-tuning the model. More details on this topic can be found in [Chapter 10](#).

4.5 Hybrid Modeling: Combining Generative and Discriminative Models

To illustrate the benefits of combining both modeling approaches, let's consider an example from commodity trading.

Commodity trading involves buying and selling raw materials such as agricultural products, oil, and natural gas, etc. Success in this field often depends on our ability to accurately predicting supply and demand. For instance, in energy trading, investors and traders would like to track global oil and gas production, which is distributed across different regions of the world.

A particularly useful application of machine learning in this field is leveraging satellite imagery to monitor fracking operations—specifically, detecting well pads (prepared sites where the drilling process occurs). This use case is mentioned in SkyFi's blog, at <https://skyfi.com/en/blog/synmax-case-study>. SkyFi is a company that provides satellite imagery data and analytics.

Let's say we have access to this data, and we are considering how it can be used to make informed trading decisions. One idea could be to build a classifier that takes satellite images as input and outputs the probability of a well pad being present. This model could be used to estimate the number of active well pads, providing insights into natural gas production and enabling investors and hedge funds to improve their decision making. In the case of fully systematic strategies, for example, the output of such a model could be incorporated into more complex factor-based trading strategies.

Now, let's imagine the process of actually building this model. We'll denote the input image as x and the binary label as y , indicating whether a well pad is present. A common

approach is to use a discriminative model that directly estimates $p(y|x)$ —the probability of a well pad being present given the satellite image.

We can assume we've collected a high-quality dataset of satellite images, all labeled with precision—either by human annotators or another reliable method. Let's say we use a Deep Neural Network model to solve this problem. The model takes an input image x and directly outputs the probability of y , indicating the presence of a well pad. If the model performs well on test data—measured using common classification metrics like AUC, precision, or recall—we might be tempted to think the problem is solved.

However, real-world deployment presents challenges. One key issue is that unexpected noise in the inputs caused by multiple factors might affect performance. Let's consider what happens when our model encounters an image that is significantly different from anything it has seen before. What will $p(y|x)$ output?

What if an input image has been affected by severe weather and environmental factors? For example, clouds, which cover around 70% of the Earth's surface at any given time, according to NASA's Earth Observatory (<https://earthobservatory.nasa.gov/features/CloudsGallery>), can obscure key features. What happens if an input image is completely covered by clouds?. Or, instead of clouds, what if geopolitical crises destroy major pipelines or natural disasters drastically alter the landscape? In real-time production environments, the input images may differ drastically from those in our training set. What will $p(y|x)$ output in such cases?

Ideally, in situations of extreme uncertainty, we'd want our model to simply say, "I don't know," and return $p(y = \text{well pad presence}|x) = 0.5$. But in reality, that's often not the case. Instead, the model may still produce an overconfident prediction.

This problem is well-documented in deep learning, and uncertainty in deep learning remains an active area of research. These models often achieve impressive performance, but they are unable to recognize "when they don't know". For example, in the paper "Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images" by Nguyen, Yosinski, and Clune (2015), the authors conducted experiments where state-of-the-art classifiers were presented with input images consisting of random noise that were unrecognizable for humans. Despite of the input been random noise, deep neural networks (DNNs) trained on ImageNet classified these inputs with $\geq 99.6\%$ confidence as belonging to familiar object categories, see Nguyen, Yosinski, and Clune (2015). This experiment provides empirical evidence on how modern deep learning models can be overconfident in their predictions, even when the input is entirely noise. Ideally, we would want our model to be robust to such inputs.

This issue arises when the distribution of the training data differs significantly from the distribution encountered during at inference time, a problem known in the machine learning literature as the out-of-distribution (OOD) problem or dataset shift. In the time series and finance literature, this is commonly referred to as the non-stationarity problem. Without going into too much detail, two common scenarios encountered in practice are covariate shift, where the distribution of input features $p(x)$ changes at inference time, and open set recognition, which occurs when new, unseen classes appear at inference time that were not present during training.

One fundamental limitation of modeling $p(y|x)$ alone is that, since it represents a probability distribution over y , the model is forced to assign probability mass over of the known classes, regardless of the input. In a binary classification scenario, for instance, the model must satisfy:

$$p(y = \text{well pad presence}|x) + p(y = \text{no well pad presence}|x) = 1$$

As a side note regarding satellite imagery and geopolitical crisis, SkyFi mentions in its blog how satellite imagery was used in 2022 to track energy infrastructure in Ukraine, including power plants, pipelines, and fuel storage facilities.

Well, let's get back to our problem. The question is: how can we solve or mitigate issues like this?

A common solution used in practice is to introduce a new class, typically called an unknown class, where images that do not belong to any of the target categories are labeled as unknown. However, this introduces another challenge: how many classes that do not belong to the target categories should we label as "unknown"? How many examples should we include for each of these classes to achieve good coverage of all possible out-of-distribution (OOD) inputs? If we rely on this method alone, we might find ourselves endlessly adding new categories without truly solving the problem. So, can we tackle this issue using a different approach?

One approach to solving this problem could be combining generative and discriminative models. The main issue with the previous approach is that, at training time, the new image was not present in the training set. But what if we had a way to quantify the probability that x comes from the training distribution? What if, instead of modeling $p(y|x)$, we modeled $p(x, y)$ instead?

By applying the product rule, we know that:
$$p(x, y) = p(x)p(y|x).$$

This means that if we can build a model for $p(x)$ (a generative model for the input features) and a separate discriminative model for $p(y|x)$, we could potentially provide a solution to our problem. Since our $p(x)$ model aims to approximate the probability distribution of the data, it should assign high probability density to familiar images (inputs) and low probability density to those that are out of distribution. Consequently, inputs that are out of distribution relative to the training set should have low probability density under $p(x)$.

For an OOD input x during training, $p(x)$ should be low, which in turn attenuates the product $p(y|x)p(x)$, making it also low. We can then use a probability threshold to decide whether to trust the prediction. For example, if $p(y|x)p(x)$ falls below a certain value, it indicates too much uncertainty, and the model should avoid making a confident prediction.

Thus, this hybrid modeling approach allows us to combine our preferred discriminative model with a generative model, improving robustness to OOD inputs and uncertainty estimation, leading to better decision-making.

Another use case, distinct from the satellite imagery example, is the well-known application in finance for predicting market regimes. Suppose we have collected a dataset consisting of pairs of inputs x and targets y , where y is a binary variable indicating whether the next market state corresponds to a bull or bear market. The inputs x represent features or characteristics of the current market state that we believe provide valuable information for predicting the next market regime state.

Similar to the previous case on satellite imagery, if we collect data only relevant for bull and bear markets—each with specific characteristics encoded in x —what happens when we encounter a new input that is very different from those observed in either regime? As before, if we model $p(y|x)$, the

system will still provide a high-confidence prediction for either bull or bear, even if the input does not fit either category well.

However, modeling $p(x)$ could offer additional insights into the novelty of a given input. This can be useful in two ways:

1. As a standalone anomaly detection system, identifying inputs that deviate significantly from the training distribution.
2. In combination with a discriminative model, improving forecasts by accounting for out-of-distribution (OOD) inputs and uncertainty.

Now, the question is: if we take this approach, how can we build $p(x)$ and $p(y|x)$? There are multiple ways to do this, but two main approaches are described below.

One straightforward solution is to build them independently and then attempt to combine them as $p(y|x)p(x)$. However, there is no guarantee that this will work. In practice, sometimes it performs very well, and other times it does not. This approach may also require numerous iterations over both models, as they are trained independently and decoupled from each other by design.

We can consider an example in finance. You may be building a ML-based trading model to predict the returns of some volatility futures. But there is a great risk of overfitting due to the limited historical data. It may therefore be advantageous to learn from the vast amount of historical data from across all financial series, futures, stocks, FX, or otherwise, and use that as prior probability of the volatility futures returns. We may also be using certain technical indicators as input to this supervised learning model. Computing the marginal probability of these indicators may inform us if certain input are outliers and may not generalize.

Another way to approach this problem is by making the models dependent on each other. This idea is explored in the paper ‘‘Hybrid Models with Deep and Invertible Features’’ by Nalisnick et al. (2019). The key idea is to use a latent variable model that not only captures the distribution of the data but also serves as an input to the discriminative model. This way, both models are tightly coupled and depend on the same latent representation. The training of $p(x)$ and $p(y|x)$ is now coupled.

In the paper, they use a flow model for $p(x)$ (a type of generative modeling you’ll learn more about in [Chapter 7](#)) and a Generalized Linear Model (GLM) for $p(y|x)$. This setup makes it possible to apply the approach to both general classification and regression problems.

4.6 Taxonomy of Generative Models

Generative models can be broadly categorized into two main types based on how they model data distribution: Explicit Density Models and Implicit Density Models. Explicit Density Models explicitly define and calculate the probability distribution of the data. Examples of Explicit Density Models include the following:

- Autoregressive Models: These models generate data sequentially, predicting each piece of the data based on what has been generated so far.
- Variational Autoencoders (VAEs): VAEs learn to encode input data into a lower-dimensional space and then decode it back, allowing for efficient data generation.
- Flow Models: This family of models transforms simple distributions into complex ones, maintaining the ability to calculate the exact likelihood of the data.

- Diffusion Models: These models gradually add noise to data and learn to reverse this process, generating data by denoising. (We don't have space to discuss diffusion models in this book, but we will cite references to it.)

On the other hand, Implicit Density Models aim to generate data by capturing the data distribution without explicitly defining it. A well-known example of these models is:

- Generative Adversarial Networks (GANs): GANs involve a dual-network architecture where one network generates data and the other evaluates it, leading to highly realistic outputs.

See [Figure 4.1](#) for a categorization of the main types of generative models.

Each model family, as specified in the model taxonomy, has its own advantages and disadvantages for abilities mentioned in the previous section. In the accompanying notebook [*Model Family Comparison—Notebook*], we present a list of the main model families and compare their efficiency. This table is something we hope to keep updating as new models are released or better techniques are developed.

4.7 Conclusion

Generative models represent a profound shift in AI's capabilities. By learning to understand and manipulate data, they enable us to categorize the world as it is and to imagine and create what it could be. From creating art to advancing scientific discovery, Generative AI is not just a tool for innovation but a foundation for future advancements in the field. In the following chapter, we will dive deeper into each of the key model families for generative models.

Chapter 5

Deep Autoregressive Models for Sequence Modeling

Autoregressive (AR) models are a familiar presence in the quantitative investor's toolbox. After all, the most basic financial data is a price series (sequence), and who doesn't want to predict the next price? In Chan (2017), Ernie discussed the various linear predictive AR models such as AR(p), ARMA(p,q), VAR, VEC, and state space models such as the Kalman filter. In this chapter, we will find out whether we can push the envelope by adopting nonlinear techniques from deep learning for such a "sequence modeling" problem. Beyond predictions, AR can be used for anomaly detection and data generation applications as well. It falls under the category of explicit density models, as illustrated in [Figure 5.1](#). This means it proposes a tractable probability density model to approximate the data distribution p_{data} .

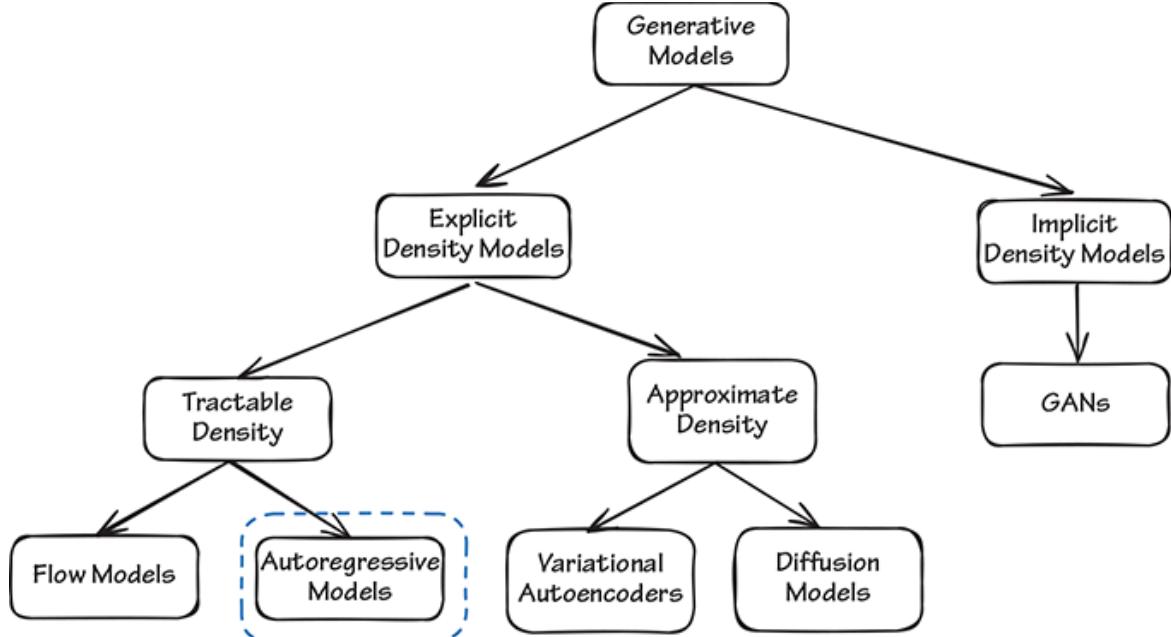


Figure 5.1 Model taxonomy: Autoregressive models.

Sequential data means that data appear in sequences where a natural or artificial order is present. This principle enables the construction of probabilistic models for any variable in the sequence by conditioning on its preceding values.

This feature makes them useful for applications involving inherently ordered data, such as audio signals, meteorological phenomena (e.g., precipitation patterns), text data for Natural Language Processing (NLP) applications, or human-created time series like the ones found in financial markets. Remarkably, AR models also apply to datasets lacking of inherent order, such as images. By imposing an order among the pixels within the image, AR models can generate complete images one pixel at a time, as demonstrated with the PixelCNN architecture, see Oord et al. (2016b) for more details.

5.1 Representation Complexity

Consider a sequence of random variables x_1, x_2, \dots, x_N . A probabilistic model of the sequence is specified by its joint probability distribution, $p_{X_1, X_2, \dots, X_N}(x_1, x_2, \dots, x_N)$. Through the application of the product rule in probability, we can decompose the joint distribution into a product of conditionals:

$$\begin{aligned} p_{\mathbf{X}}(\mathbf{x}) = p_{X_1, X_2, \dots, X_N}(x_1, \dots, x_N) &= p_{X_1}(x_1) \\ &\quad \times p_{X_2|X_1}(x_2|x_1) \\ &\quad \times p_{X_3|X_1, X_2}(x_3|x_1, x_2) \\ &\quad \dots \\ &\quad \times p_{X_N|X_1, \dots, X_{N-1}}(x_N|\mathbf{x}_{1:N-1}) \end{aligned} \tag{5.1}$$

AR models simplify the modeling of high-dimensional distributions by exploiting the right-hand side of [Equation 5.1](#), modeling a series of conditional distributions.

These distributions provide probabilistic descriptions for each variable based on its predecessors in the sequence, transforming complex high-dimensional modeling into a series of more manageable conditional probabilities that are easier to represent and estimate.

Despite AR models' simplification of high-dimensional problem into more manageable one-dimensional conditional distributions, the computational cost of inference and storage remains a challenge. For example, in financial time series like the S&P500, discretizing values into a finite states, let's say, five states—a common practice in financial applications, such as the NUMERAI Hedge Fund competition—illustrates the complexity. As an example, consider modeling the conditional distribution $p_{X_{n+1}|\mathbf{x}_{1:n}}(x_{n+1}|\mathbf{x}_{1:n})$ given the previous $n = 21$ values in the sequence. Think of this example

as forecasting the next day's returns of a security based on the last month of daily returns, similar to trading applications where the past month's data is used for prediction. This approach would require a huge probability table with 5^{21} entries, which is clearly impractical. Instead of specifying conditional probabilities through table lookups, a parameterized model is necessary. However, even with a parameterized approach, modeling interactions between all variables can result in a prohibitive number of parameters. Therefore, we need to find ways to reduce the number of parameters.

5.2 Representation and Complexity Reduction

One way to reduce the number of parameters in the model is by making assumptions about the structure of the joint distribution in [Equation 5.1](#). These assumptions might include specifications about variable interactions, density forms, etc.

One approach to reduce complexity is to impose conditional probability assumptions, such that to model x_n we do not need the knowledge of the entire sequence of previous values x_1, \dots, x_{n-1} . For example, we could say that to model x_n , all we need to know is the value of x_{n-1} , suggesting that all relevant information is contained in this variable. (There is no loss of generality in assuming that x_n only depends on the previous state x_{n-1} , because we could just redefine a state to include information from a longer lookback period.)

Mathematically, we express this as

$p_{X_n|X_1, \dots, X_{n-1}}(x_n|x_1, \dots, x_{n-1}) = p_{X_n|X_{n-1}}(x_n|x_{n-1})$, also known as the Markov Property. We touched on this in [Chapter 3](#) when we discussed the Hidden Markov Model.

Another technique involves weight sharing, where, in the case of the previous example, the conditional probability $p_{X_n|X_{n-1}}(x_n|x_{n-1})$ is independent of the sequence position n , imposing a kind of invariance or stationarity across the sequence.

These simplifications also reduce the expressiveness of the model by narrowing the types of probability distributions that can be represented.

To illustrate the concept of conditional independence and model stationarity in sequence modeling, let's consider a practical example involving an AR model of order 2, denoted as AR(2). An AR(2) model factorizes the joint probability of a sequence x_1, \dots, x_N as follows:

$$p(x_1, \dots, x_N) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)p(x_4|x_2, x_3)\dots p(x_N|x_{N-2}, x_{N-1}) \quad (5.2)$$

In this scenario, we assume that x takes values in the space $\chi = \mathbb{R}$ and that we can further model each conditional probability using a Gaussian distribution. Specifically, for any given x_n , the conditional probability $p(x_n|x_{n-2}, x_{n-1})$ is expressed as:

$$p(x_n|x_{n-2}, x_{n-1}) = \mathcal{N}(w_0 + w_1 x_{n-1} + w_2 x_{n-2}, \sigma^2) \quad (5.3)$$

here $\mathcal{N}(\cdot, \sigma^2)$ denotes a Gaussian distribution with the mean determined by the linear combination of the two preceding values in the sequence and a constant term, and with constant variance σ^2 . As an example of using a linear AR models, the accompanying notebook [*AR Notebook*] demonstrates forecasting the quarterly growth rate of the US gross national product (GNP).

This Gaussian assumption is for illustration only. We do not need to constrain ourselves to any particular form of the density function $p_{\theta}(\cdot|\cdot)$. We could even choose to model variables taking values in either continuous or discrete spaces. The choice of the density function may determine whether you frame the task as a classification problem or a regression problem, each with its own distinct characteristics. These representations will have their advantages and disadvantages in terms of efficient training and model representation, expressiveness and generalization, sampling quality and speed, and compression rate.

In this chapter, we will explore various model classes that are commonly used to model conditional probabilities, starting from traditional to more cutting-edge architectures. We will start with model families such as Logistic Regression to lay the groundwork and clarify essential concepts. Next we explore Neural Networks and Recurrent Neural Network leading to more complex families. We then advance to innovations that serve as the cornerstone of modern architectures, including Causal Masked Neural Networks, specifically WaveNet, and transformers. (The discussion on Causal Masked Neural Networks is inspired by insights gained from the lecture slides of Abbeel et al. [2020]).

To illustrate main concepts and techniques for image applications, we will use the standard Binary MNIST dataset (Larochelle and Murray, 2011), a canonical example in Computer Vision. Even though we use simple datasets, these models have the potential to extend far beyond these basic applications.

These models can be used to develop trading strategies utilizing alternative data. For instance, analyzing satellite images of crop fields can help predict commodity prices (Guida, 2019; Denev and Amen, 2020), while observations of retail parking lots can forecast retail company earnings (Guida, 2019), among other applications. Of course, we can

also apply these models to price, or more appropriately, returns series.

By the end of the chapter, we will provide examples of how these models can be applied for asset management and trading, while also exploring how Transformer models can be adapted for time-series forecasting tasks. In [Chapter 9](#), “Leveraging Large Language Models (LLMs) for Text Data Analysis in Trading,” we will demonstrate how to construct a simple high-frequency trading strategy using Auto Regressive Generative Models to process speech signals from Federal Reserve announcements to inform trading decisions.

5.3 A Short Tour of Key Model Families

5.3.1 Logistic Regression Model

The binary MNIST dataset (Larochelle and Murray, 2011) is a simplified version of the original MNIST dataset, which has been widely used as a benchmark in computer vision.

This dataset consists of 28×28 pixel images, with each pixel falling into $\{0, 1\}$, representing the color black or white at that pixel location. The binary constraints means that there are $D = 2^{764}$ possible images, highlighting the fact that any practical training set can cover only a very small portion of all potential image configurations. Indeed, the dataset contains only 60,000 images for training and 10,000 images for testing.

In this context, an image \mathbf{x} is an element of the space $\mathcal{X} = \{0, 1\}^D$.

As typical examples of digits in the dataset, see [Figure 5.2](#).



Figure 5.2 Sample from the Binarized MNIST dataset.
Larochelle and Murray (2011).

For a given image \mathbf{x} , we can write its joint probability as follows:

$$\begin{aligned}
 p_{\mathbf{X}}(\mathbf{x}) = p_{X_1, X_2, \dots, X_{768}}(x_1, \dots, x_{768}) &= p_{X_1}(x_1) \\
 &\quad \times p_{X_2|X_1}(x_2|x_1) \\
 &\quad \times p_{X_3|X_1, X_2}(x_3|x_1, x_2) \\
 &\quad \dots \\
 &\quad \times p_{X_{768}|X_1, \dots, X_{767}}(x_{768}|\mathbf{x}_{1:767})
 \end{aligned} \tag{5.4}$$

This factorization is presented verbosely to highlight two key aspects of this model:

- Lack of parameters sharing
- No assumptions of conditional independence

In this model, each conditional probability is modeled using a logistic model defined as:

$$p(x_d | \mathbf{x}_{1:d-1}) = f(\mathbf{x}_{1:d-1}; \theta^{(d)}) = \text{sigmoid}(\mathbf{x}_{1:d-1}^T \mathbf{w}^{(d)} + b^{(d)}) \quad (5.5)$$

(Recall from [Chapter 3](#) that the logistic model is defined in terms of the sigmoid function.) Here, a unique set of parameters $\boldsymbol{\theta}^d = \{\mathbf{w}^d, b^d\}$ is assigned to each conditional. This model is known as a *fully visible sigmoid belief network* (FVSBN); see Frey et al. (1995).

So far, we have discussed several model representations and will introduce a few more. Keep in mind, however, that model training and parameter estimation will be covered in detail in [Section 5.4](#) of this chapter.

5.3.1.1 Sampling from FVSN.

The sampling process involves generating one pixel at a time, sequentially. For example, the first pixel is sample from its distribution, $x_1 \sim p_{X_1}$. Subsequent pixels are generated conditioned on the pixel values realized previously, e.g., $x_2 \sim p_{X_2|X_1}(\cdot, x_1)$, continuing in this manner for the rest of the pixels. This method is known as Ancestral Sampling (Bishop, 2006).

Instead of modeling the conditionals using a logistic model, we could explore using a different function to enhance model expressiveness, such as a Neural Network. Here,

$$p(x_d | \mathbf{x}_{1:d-1}) = f(\mathbf{x}_{1:d-1}; \theta^{(d)}) \quad (5.6)$$

where now $f = \text{NeuralNetwork}(\mathbf{x}_{1:d-1}; \boldsymbol{\theta}^{(d)})$. Of course, modeling with a Neural Network will increase the number of parameters to estimate.

5.3.2 Masked Autoencoder for Density Estimation (MADE)

The Masked Autoencoder for Density Estimation (MADE), see Germain et al. (2015), represents a significant advancement in the field of Generative AI by enhancing the capabilities of traditional autoencoders. Traditional autoencoders compress input data into a latent representation and then attempt to reconstruct it, introducing and denoising noise in the process. However, they are limited in their ability to generate new data points or to model the probability distribution of the data.

MADE addresses these limitations by introducing a sequential dependency among the input variables, following the chain rule of probability. This method ensures that the generation of any data point in the sequence does not depend on future data points, adhering to a predefined ordering. This innovation allows MADE to output probabilities and sample new data points in a way that respects the underlying data distribution, marking a departure from traditional autoencoders.

Key features of MADE include its flexibility in handling input variables of varying dimensionality and its use of a softmax output for each variable, representing a probability distribution from which new data points can be sampled. The model's structure, which incorporates masked connections between nodes, enables it to maintain the autoregressive property essential for accurate density estimation.

Illustrating the application of MADE, experiments on datasets like MNIST demonstrate its capability to generate digit images that increasingly resemble authentic data points with extended training. [Figure 5.3](#) shows an image generated in the left and its nearest neighbor in the right, which is a very good practice to verify the model is not just learning the

images of the dataset, and that it is actually generating new ones.

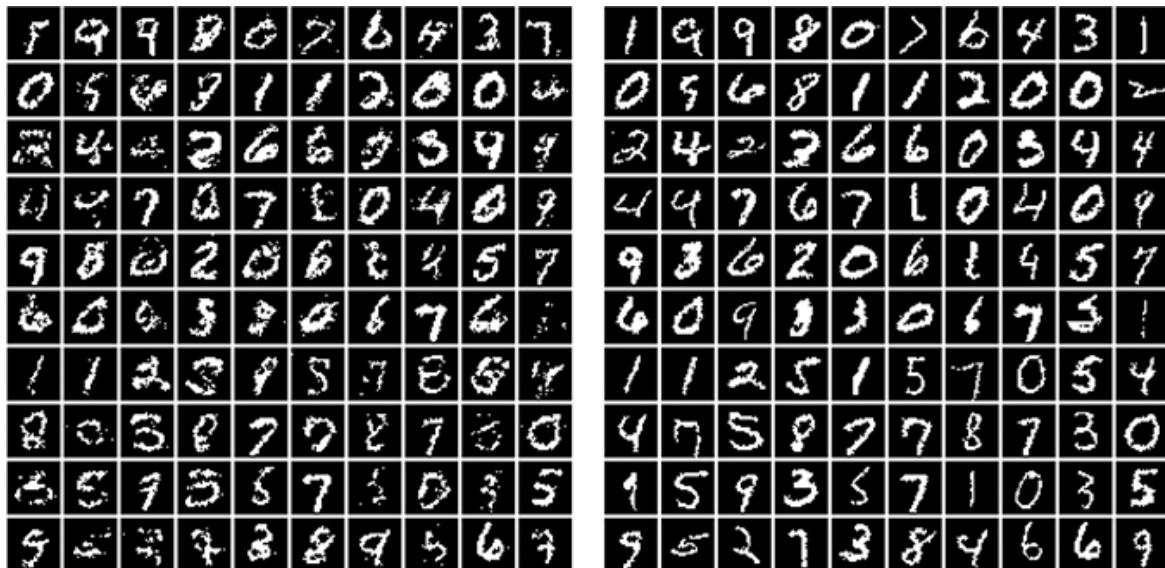


Figure 5.3 MADE Generation on MNIST. Left: samples from a MADE model. Right: Nearest neighbor in binarized MNIST.

Source: Germain et al. (2015).

In conclusion, MADE's innovative approach to combining autoencoder frameworks with probabilistic modeling techniques has established it as a foundational model in Generative AI. Its contribution lies in its ability to generate new data points and model data distributions more accurately, paving the way for further advancements in the field.

5.3.3 Causal Masked Neural Network Models

Causal Masked Neural Networks have emerged as a leading approach in the field of Generative AI, building on the foundational concepts introduced by models like MADE. This approach has advanced the state of generative modeling by implementing efficient parameter sharing and introducing

coordinate coding to incorporate location information about data sequences.

At the heart of Causal Masked Neural Networks is the principle of parameterizing conditionals with a neural network, similar to MADE, but with a crucial innovation: the use of the same neural network parameters across different stages of the generation process. This approach is akin to applying a “sliding window” over the data, allowing the repetitive use of a set of parameters, significantly increasing model efficiency by reducing the total number of parameters needed while still modeling high-dimensional data.

To address the challenges associated with parameter sharing, particularly the potential loss of positional information in data such as images, these networks incorporate coordinate coding. This technique involves feeding the location coordinates of data points (e.g., pixels in an image) into the model along with the data itself, enabling the network to retain spatial awareness despite the uniform use of parameters across different positions. Coordinate coding can be implemented in various forms, such as one-hot encodings or relative position encodings, adding a layer of flexibility to the model’s design.

5.3.3.1 WaveNet.

One of the first and most successful implementations of this approach was WaveNet (see Oord et al., 2016a), a model primarily designed for generating speech. Its influence extends beyond academic circles, with variations of the model being implemented in real-world systems such as personal assistants on smartphones. WaveNet is recognized for its superior audio generation quality, making it a cornerstone in the evolution of audio synthesis technologies.

WaveNet utilizes 1D convolutional layers and introduces additional techniques like dilation in convolutions to capture

information from further back in the sequence, see [Figure 5.4](#). This means WaveNet can capture longer-range dependencies within the audio data more efficiently, improving the quality and naturalness of the generated audio. This model has showcased the potential of this approach not only for audio data but also for a wide range of sequential data types.

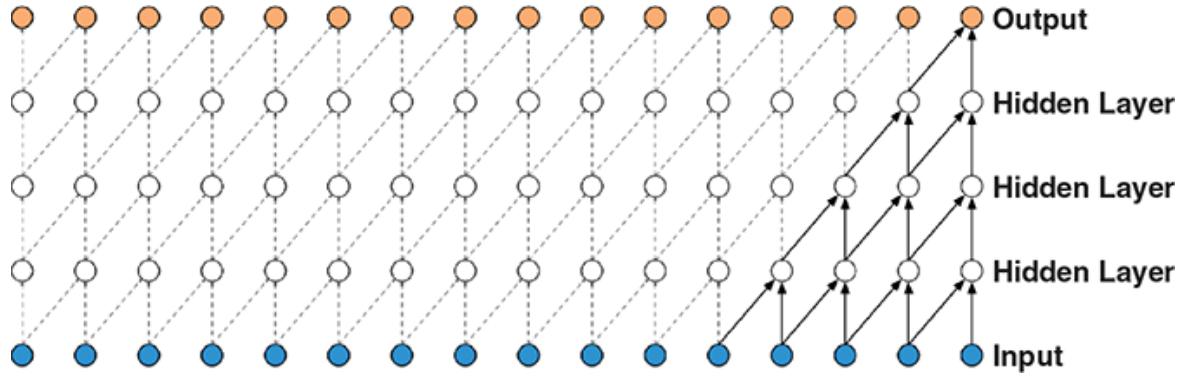


Figure 5.4 Visualization of a stack of causal convolutional layers. Figure 2 from Oord et al. (2016a).

In practice, WaveNet utilizes a stack of dilated causal convolutional layers to enable networks to have very large receptive fields, thereby allowing the network to consider inputs located far away in the sequence. For example, [Figure 5.5](#) illustrates dilated causal convolutions with dilations of 1, 2, 4, and 8, as shown in the paper by Oord et al. (2016a).

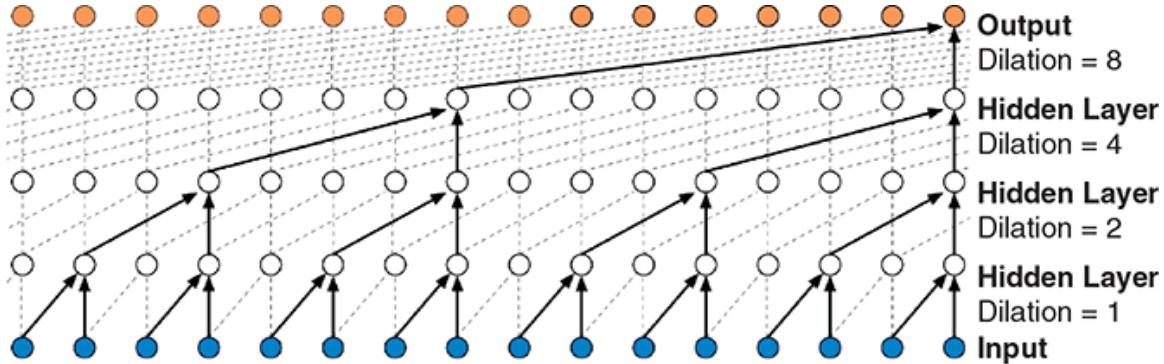


Figure 5.5 Visualization of a stack of dilated causal convolutional layers. Figure 3 from Oord et al. (2016a).

As an example of using WaveNet, the accompanying notebook [*WaveNet Notebook*] demonstrates its application for forecasting financial data, providing an opportunity for hands-on learning with this type of models

Despite the efficiency and expressiveness of Causal Masked Neural Network Models, challenges remain, particularly in the context of sampling. Generating new data points requires processing each element sequentially, which can be slow. However, the training process benefits from parallelism, significantly speeding up model learning. Moreover, concerns about the finite context window—how far back the model can look when generating the next part of a sequence—have been mitigated by advancements in attention mechanisms.

5.3.4 Recurrent Neural Networks (RNN)

As discussed in [Chapter 3](#), Recurrent Neural Networks (RNNs) are well known for their ability to process sequential data with an effectively infinite look-back capability. This feature distinguishes them from other models that are limited to fixed-size inputs, enabling RNNs to dynamically incorporate all prior information in a sequence. This capability makes RNNs exceptionally well-suited for complex

tasks such as speech recognition, natural language processing, and other time-series related applications.

To recap, at the core of an RNN is the concept of a hidden network state, often referred to simply as the “state.” This state acts as the network’s memory, theoretically holding all necessary past information to make future predictions. In a typical RNN, the hidden state at any sequence index t , denoted by h_t , is updated based on the previous state h_{t-1} and the current input x_t , according to a function g . The output y_t at each timestep depends on the current state h_t using a typical nonlinear function f ,

$$\begin{aligned} h_t &= g(h_{t-1}, x_t) \\ y_t &= f(h_t) \end{aligned} \tag{5.7}$$

Here, f and g are typical nonlinear neural network layers that are independent of the specific index location t . Consequently, RNNs effectively employ parameter sharing across the sequence.

This recursive nature allows RNNs to handle variable input lengths and provides them with the theoretical capacity to manage an infinite window of context, which is ideal for generating coherent and contextually rich sequences over time.

However, the practical application of RNNs is often challenged by issues related to their deep and recursive structure. Notably, the training of RNNs can be subject to the phenomena of vanishing and exploding gradients. These occur when the gradient signals, essential for learning through backpropagation, diminish to insignificant levels (vanish) or grow too large (explode) as they are propagated back through each “timestep” in the sequence. Such issues can severely limit the RNN’s ability to learn long-term dependencies within the data affecting the overall efficacy of the model in real-world applications.

In generative tasks, RNNs operate by predicting the next item in a sequence based on the history it has processed, with each output dependent on the preceding computational state. For example, when trained on specific datasets like the works of Shakespeare or encoded Wikipedia text, RNNs can generate text that mimics the style of the training data, capturing nuances such as punctuation, grammar, and even the specific formatting of text.

In the accompanying notebook [*RNN Notebook*], we demonstrate an experiment using RNNs to forecast price movements and returns of the S&P 500.

5.3.4.1 Practical Considerations and Limitations.

While theoretically capable of handling sequences of arbitrary length and complexity, RNNs in practice face significant operational challenges:

- **Training Speed:** Training RNNs is slow due to the sequential nature of their computations, which prevents parallel processing. Each step depends on the completion of the previous step, leading to longer training times.
- **Gradient Issues:** The training of RNNs is notoriously difficult due to vanishing and exploding gradient problems, which can severely impact learning effectiveness. Techniques like Long Short-Term Memory (LSTM) units have been developed to mitigate these issues but do not entirely eliminate them.
- **Memory Requirements:** Due to their recursive nature, RNNs often require substantial memory for training, as they need to maintain information across many time steps.

5.3.5 Transformers

Transformer models have revolutionized both industry and academia in recent years. Since their introduction in the highly influential paper *Attention Is All You Need* by Vaswani et al. (2023), they have been successfully applied to a wide range of data types, including text, audio, time series, image, and video. As autoregressive models, Transformers offer several advantages over traditional RNNs. One of the most significant is their ease of training and the ability to leverage parallelization, which accelerates both training and inference. Additionally, Transformers mitigate issues such as vanishing and exploding gradients, which are more prevalent in earlier models like RNNs and LSTMs.

In this section, we will explore the Transformer architecture in detail, breaking down its core components. By the end of this section, we will have covered all the essential building blocks necessary to implement two of the most influential Transformer models: BERT, introduced by (Devlin et al. 2019) and developed by Google, and GPT, introduced by Radford et al. (2018) and developed by OpenAI. BERT, which stands for Bidirectional Encoder Representations from Transformers, revolutionized natural language processing, redefining how both industry and academia tackle a wide range of NLP tasks. Similarly, the GPT family of models (Generative Pre-trained Transformers) by OpenAI has been groundbreaking, pushing the boundaries of what's possible with language generation and understanding.

For clarity of exposition, let's focus on text data, as it makes these models easier to understand. Keep in mind, however, that the same framework may be applicable to numerical time series, as both text and time series are considered "sequences." We will elaborate on this point further in [Section 5.3.6](#). We will briefly cover the essential steps involved in a typical preprocessing pipeline for Natural Language Processing (NLP) applications, providing just

enough information to understand Transformers. To do this, we will use a combination of code snippets and mathematical notation to illustrate how these models work.

5.3.5.1 Attention Mechanism.

To understand the concept of attention, let's consider the following two sentences:

- *The startup achieved a new **round** of funding to expand operations.*
- *He invited his friends for another **round** of drinks at the bar.*

In the first sentence, the word “round” refers to a phase of funding for a company, while in the second sentence, it refers to a sequence of drinks being ordered. Intuitively, these different meanings suggest that the word “round” should be represented by different vectors, also known as vector embeddings—or simply embeddings in the NLP literature—depending on the context.

However, in the initial steps of the NLP pipeline—when raw text is transformed into token embeddings—the model assigns the same vector representation to the words “round” in both sentences, regardless of the context. This is because token embeddings, at this point, are **context-independent** and rely solely on the token ID assigned to the word.

The NLP Pipeline: From Raw Data to Token Embeddings:

1. **Tokenization:** The first step in modern NLP pipelines involves converting raw text into smaller units called **t**okens****. Tokens can be full words, subwords, or even characters, depending on the tokenizer. Each token is mapped to a unique token ID.

2. **Token Embeddings:** The model assigns an initial embedding to each token based on the token ID. At this stage, the embeddings are purely **context-independent** —the same word will always have the same embedding, regardless of the surrounding text.

To illustrate this, we will use the Hugging Face (<https://huggingface.co>) `transformers` python library, which simplifies working with Transformer models like BERT. In the following code, we examine the tokenization process:

```
from transformers import BertTokenizer, BertModel
import torch

# Initialize BERT tokenizer and model
model_ckpt = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)

# Two sentences with different meanings for "round"
sentence_1 = "The startup achieved a new round of funding to expand operations."
sentence_2 = "He invited his friends for another round of drinks at the bar."

# Tokenize the sentences
inputs_1 = tokenizer(sentence_1, return_tensors='pt')
inputs_2 = tokenizer(sentence_2, return_tensors='pt')

# Get the token ID for "round"
token_id = tokenizer.convert_tokens_to_ids('round')
print(f"Token ID of word round: {token_id}")

# Print token IDs to show tokenization process
print("Token ids for sentence 1:", inputs_1.input_ids)
print("Token ids for sentence 2:", inputs_2.input_ids)

# Access the token embeddings directly from BERT's embedding matrix
# The embedding matrix is model.embeddings.word_embeddings
token_embedding_matrix = model.embeddings.word_embeddings.weight

# Extract the token embedding for "round"
token_embedding_matrix[token_id]

# Print results to verify that token embeddings are identical
```

```
print(f"Token embedding for 'round': {token_embedding_for_round_1[:5]}...")  
# First 5 values  
>>>  
Token ID of word round: 2461  
Token ids for sentence 1: tensor([[  
    101, 1996, 22752, 4719,  
    1037, 2047, 2461, 1997,  
    4804, 2000, 7818, 3136,  
    1012, 102]])  
Token ids for sentence 2: tensor([[  
    101, 2002, 4778, 2010,  
    2814, 2005, 2178, 2461,  
    1997, 8974, 2012, 1996,  
    3347, 1012, 102]])
```

Here, the Token ID for the word “round” corresponds to the ID 2461, which is located at position 6 and position 7 (0-indexed) in `inputs_1.input_ids` for sentence 1 and `inputs_2.input_ids` for sentence 2, respectively.

To determine the token embeddings assigned to the word “round” let’s access the token embedding matrix of BERT:

```

# Access the token embeddings directly from BERT's embedding matrix
# The embedding matrix is model.embeddings.word_embeddings
token_embedding_matrix = model.embeddings.word_embeddings.weight

# Extract the token embedding for "round"
token_embedding_for_round = token_embedding_matrix[token_id]

print(f"Embedding matrix size: {token_embedding_matrix.size()}")
>>>
Embedding matrix size: torch.Size([30522, 768])

print(f"Token embedding for 'round': {token_embedding_for_round[:5]}...")
>>
Token embedding for
'round': tensor([ 0.0109,  0.0577,
 0.0310, -0.0615,  0.0571], grad_fn=<SliceBackward0>)...

```

The embedding matrix size for the BERT model is `torch.Size([30522, 768])`, meaning that each token ID is mapped to a 768-dimensional vector representation, and the vocabulary—the number of unique tokens for this specific model—consists of 30,522 tokens. More details about how these embedding representations are created and learned will be provided later. Additionally, more information about this process and the typical NLP pipeline can be found in the Hugging Face tutorial, located in the [chapter 10](#) folder of the book repository.

The purpose of the attention mechanism is to refine the representation of each word in a sentence, making it more context-dependent. After applying attention, we obtain a more refined embedding that captures the context in which

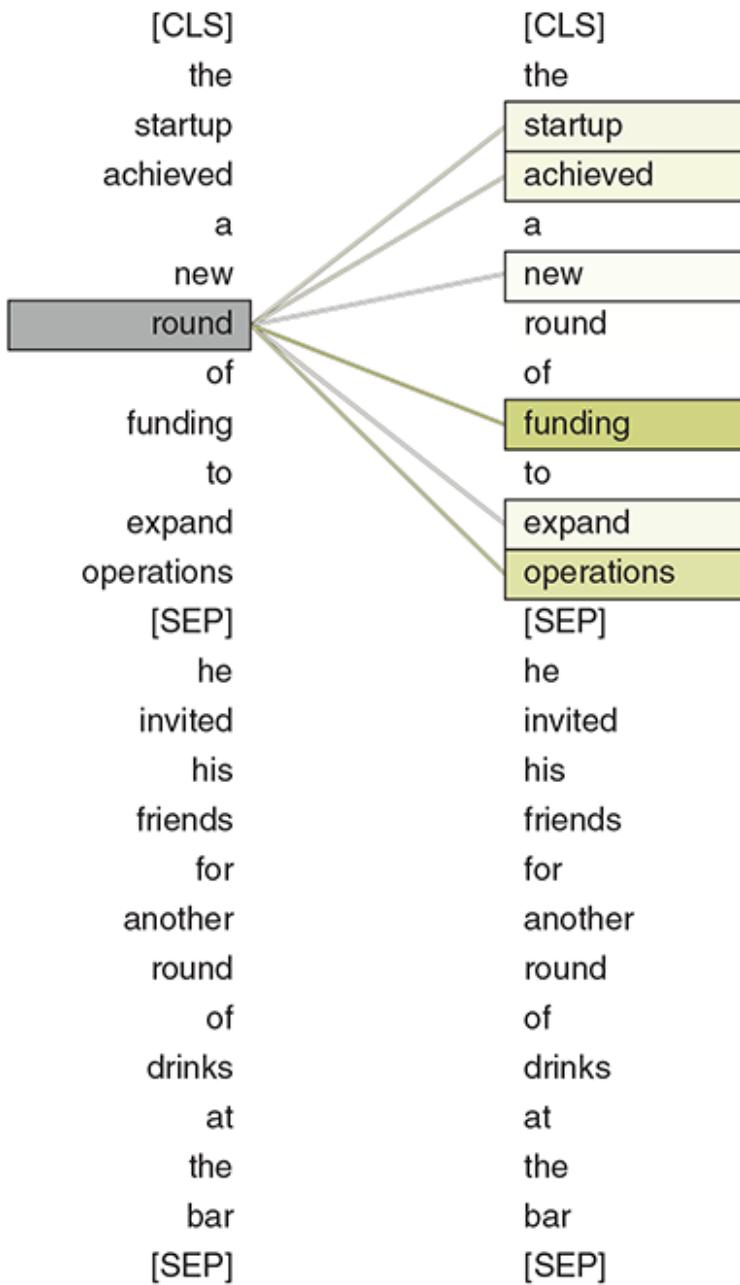
the word appears. The importance of the attention mechanism is so profound that the authors of the influential paper “Attention Is All You Need” (Vaswani et al., 2023) emphasized it in the very title. But how exactly does attention work to refine these embeddings?

Before diving into the mathematical details, let’s explore some insightful visualizations using BertViz, a tool introduced by Vig (2019), which helps us understand how models like BERT, GPT2, or T5 update word representation through attention. This visualization will help us to grasp the concept before looking at the equations.

Most attention mechanisms refine embeddings by computing a linear combination of all input embedding using specific “attention weights.” The attention mechanism makes these “attention weights” or “attention scores” dependent on the input, allowing the model to create context-dependent representations. To clarify this concept, we will use BertViz to see how BERT assign attention weights across different words in a sentence.

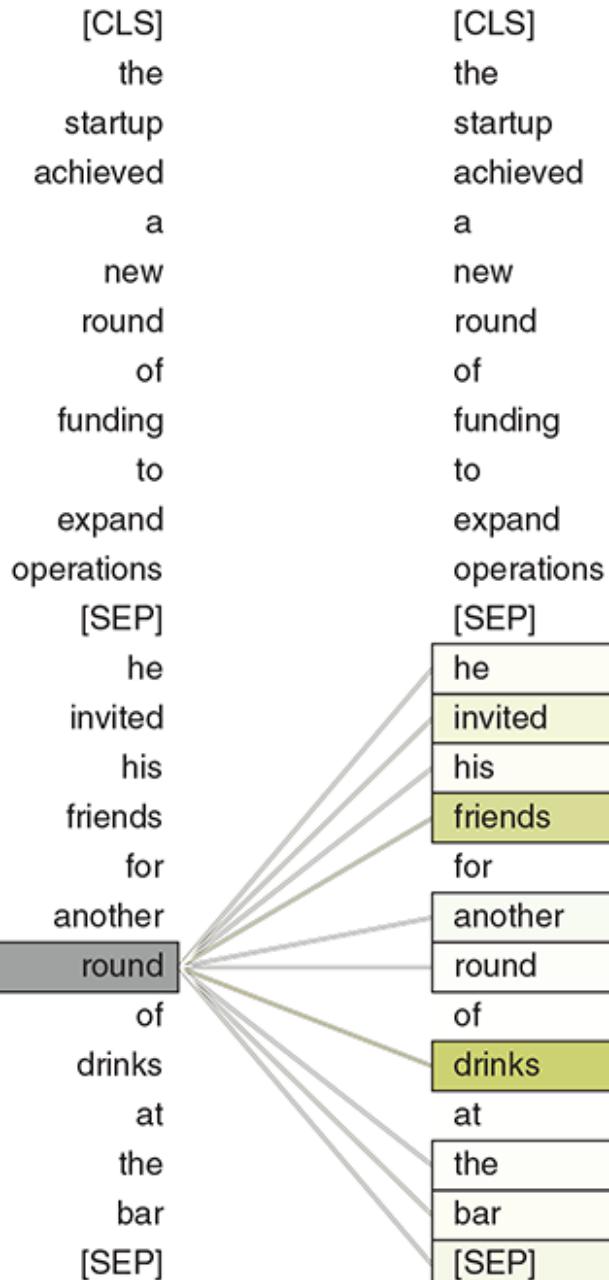
In the visualizations generated by the following code, namely [Figures 5.6a](#) and [b](#), it is displayed the attention scores for two input sentences:

Layer: 0 ▾ Attention: All ▾



(a)

Layer: 0 ▾ Attention: All ▾



(b)

Figure 5.6 Visualizing attention.

```

from transformers import AutoTokenizer
from transformers import AutoModel
from bertviz import head_view

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = AutoModel.from_pretrained(model_ckpt, output_attentions=True)
sentence_a = 'The startup achieved a new round of funding to expand operations'
sentence_b = 'He invited his friends for another round of drinks at the bar'

viz_inputs = tokenizer(sentence_a, sentence_b, return_tensors='pt')
attention = model(**viz_inputs).attentions
sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1)
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids[0])

head_view(attention, tokens, sentence_b_start, heads=[8])

```

Here, we instructed BERT to process the two input sequences simultaneously. Both sentences are shown in the left and right column, separated by the special [SEP] token.

In these figures, the right column represents the initial input embeddings, while the left column shows the output embeddings after applying the attention mechanism. Although the visualizations do not show the final word embeddings, they display the attention scores between words. By selecting any word in the left column, highlighted with a gray background, we can see lines connecting this word to all words in the right column. The intensity of each line represents the magnitude of the attention score, ranging from 0 to 1, indicating how much influence each word in the right column has on the selected word in the left column.

Let's look at [Figure 5.6a](#), where we focus on the word **round** in the first sentence, "*The startup achieved a new **round** of funding to expand operations.*" The visualization shows that the words "funding," "operations," "startup," and "achieved" have the highest attention scores, contributing the most to refining the representation of "round." Notably, none of the words from the second sentence are considered important in updating this specific representation.

On the other hand, in [Figure 5.6b](#), we select the word **round** from the second sentence, "*He invited his friends for another **round** of drinks at the bar.*" In this context, the words "drinks," "friends," and "invited" are most significant in updating the representation of "round." Similarly, no significant attention is assigned to words from the first sentence.

These visualizations illustrate how attention helps the model focus on relevant parts of the context when refining word representations. Now, let's see how this attention update is computed.

If we are dealing with a stock's returns series instead, we might imagine that a return at time t may depend significantly on some past return $t' = t - \Delta$ that was particularly significant, perhaps because there was an earnings announcement or economic news at t' (Guizarro-Ordonez et al., 2021).

Now, let's see how this attention update is already computed. Let's say we have some embedding representation for each token in the input sequence, denoted as $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N$, each of dimension \mathbb{R}^{d_v} . (In the case of a returns series, each v_t might be an embedding representation of the stock returns at time t in a portfolio.) As mentioned, most attention mechanisms create refined embeddings by computing a linear combination of all input embeddings, such as the \mathbf{v}_s in our example, which, as we will see next, are linear combinations of the input embeddings \mathbf{X} . To find a refined

representation for each input vector, we compute a weighted sum. For example, the refined representation, or hidden representation, \mathbf{h}_1 for \mathbf{v}_1 is computed as:

$$\begin{aligned}
 \mathbf{h}_1 &= a_1 \mathbf{v}_1 + a_2 \mathbf{v}_2 + \dots + a_N \mathbf{v}_N \tag{5.8} \\
 &= \sum_n a_n \mathbf{v}_n \\
 &= \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \\
 &= \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_1 \\ | \end{bmatrix}
 \end{aligned}$$

where $\mathbf{a} = [a_1, a_2, \dots, a_N]^T$ is the attention vector, also known as attention scores or attention weights, used to compute \mathbf{h}_1 . Similarly, for the refined representation or hidden representation of the third vector, we have:

$$\mathbf{h}_3 = \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_3 \\ | \end{bmatrix} \tag{5.9}$$

In general, the refined representation for the i -th input is:

$$\mathbf{h}_i = \begin{bmatrix} | & \cdots & | \\ \mathbf{v}_1 & \cdots & \mathbf{v}_N \\ | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ \mathbf{a}_i \\ | \end{bmatrix} \tag{5.10}$$

This can be represented in matrix form as:

$$\begin{bmatrix} \cdots \mathbf{h}_1^T \cdots \\ \cdots \mathbf{h}_2^T \cdots \\ \vdots \\ \cdots \mathbf{h}_N^T \cdots \end{bmatrix} = \begin{bmatrix} \cdots \mathbf{a}_1^T \cdots \\ \cdots \mathbf{a}_2^T \cdots \\ \vdots \\ \cdots \mathbf{a}_N^T \cdots \end{bmatrix} \begin{bmatrix} \cdots \mathbf{v}_1^T \cdots \\ \cdots \mathbf{v}_2^T \cdots \\ \vdots \\ \cdots \mathbf{v}_N^T \cdots \end{bmatrix} \quad (5.11)$$

$$\mathbf{H} = \mathbf{A}\mathbf{V}$$

where \mathbf{A} is the matrix of attention weights. In short, we update each token through a linear combination of all other tokens. In a financial time-series application for forecasting returns, to avoid look-ahead bias, we should only include past returns when calculating the current return. This approach is known as causal masked attention in NLP terminology. The question now is, how do we determine this matrix \mathbf{A} ? Well, these weights are learned from data during training.

Before explaining how the attention matrix is computed, let's clarify the difference between *hard attention* and *soft attention*.

- **Hard Attention:** Each attention vector \mathbf{a}_n satisfies the property $\|\mathbf{a}_n\|_0 = 1$, meaning it is a one-hot vector where only one component is 1 and the rest are 0. This means the new representation will be exactly one of the input vectors.
- **Soft Attention:** Each attention vector \mathbf{a}_n satisfies $\|\mathbf{a}_n\|_1 = 1$, meaning the new representation is a weighted combination of the input vectors.

Although there are several methods to compute the attention matrix and the matrix of embedding vectors \mathbf{V} , in this book, we will focus on the most well-known approach, Scaled Dot-Product Attention, as introduced in the influential paper by Vaswani et al. (2023).

5.3.5.2 Scaled Dot-Product Attention.

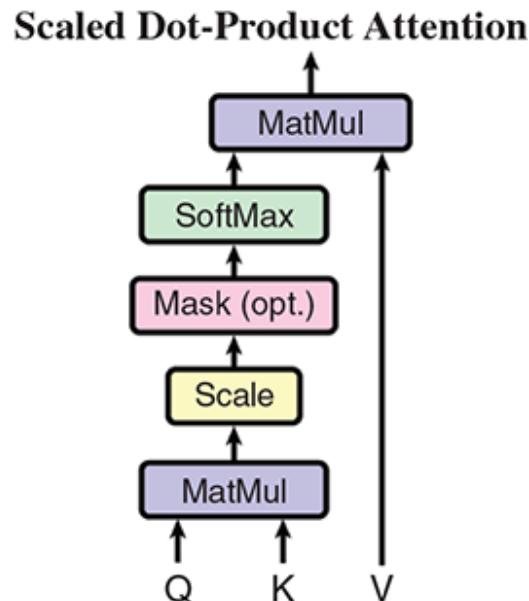
In Scaled Dot-Product Attention, the attention scores \mathbf{A} are computed as:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \quad (5.12)$$

and so, the hidden representation is computed as:

$$\begin{aligned} \mathbf{H} &= \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \\ \mathbf{H} &= \mathbf{AV} \end{aligned} \quad (5.13)$$

The operations of the Scaled Dot-Product Attention are illustrated in [Figure 5.7](#), serving as a useful reference for understanding the process.



[Figure 5.7](#) Scaled Dot-Product Attention. [Figure 2 \(left\)](#) from Vaswani et al. (2023).

Let's break down the equation for Scaled Dot-Product Attention.

- **\mathbf{Q}** (Query Matrix): This matrix is computed from the input embeddings \mathbf{X} as $\mathbf{Q} = \mathbf{X}\mathbf{W}_q$. For now, consider the input embeddings \mathbf{X} as those generated from the basic NLP pipeline explained earlier, where they are assigned based on token ID. In [Section 5.3.5.4](#), we will discuss how additional sequence information is incorporated into the token embeddings to fully create \mathbf{X} . The query matrix \mathbf{Q} contains N row vectors, each representing a query vector \mathbf{q}_n^T of dimension d_q . The matrix \mathbf{W}_q is a learnable parameter of size $d \times d_q$.
- **\mathbf{K}** (Key matrix): Similarly, the Key matrix is computed as $\mathbf{K} = \mathbf{X}\mathbf{W}_k$. Each row of \mathbf{K} corresponds to a key vector \mathbf{k}_n^T of dimension d_k . The matrix \mathbf{W}_k is also learnable parameter of size $d \times d_k$.
- **\mathbf{V}** (Value matrix). The Value matrix is computed as $\mathbf{V} = \mathbf{X}\mathbf{W}_v$. It contains N row vectors, each representing a value vector \mathbf{v}_n^T of dimension d_v . The matrix \mathbf{W}_v is a learnable parameter of size $d \times d_v$.

For Scaled Dot-Product Attention to work, the dimensions d_q and d_k must match, and a common choice is $d_q = d_k = d$.

The attention mechanism used here is a soft-attention mechanism, enforced by applying the softmax function to ensure that the rows of \mathbf{A} sum to 1. The scaling factor $1/\sqrt{d_k}$ was introduced by the authors of the *Attention Is All You Need* paper to prevent the argument of the softmax from becoming too large, thereby avoiding regions of the function with very small gradients (Vaswani et al., 2023).

To better understand Scaled Dot-Product Attention, let's focus on the computation of the unnormalized attention scores:

$$\widetilde{\mathbf{A}} = \mathbf{Q}\mathbf{K}^T \quad (5.14)$$

We can call this matrix $\widetilde{\mathbf{A}}$ the unnormalized attention scores. For every input vector in the sequence $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$, we compute its query representation $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_N$, its key representation $\mathbf{k}_1, \mathbf{k}_2, \dots, \mathbf{k}_N$, and its value representation $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_N$, and using this information, we compute the attention weights.

For example, to compute the attention weights $\tilde{\mathbf{a}}_1$ needed to the hidden representation \mathbf{h}_1 of \mathbf{x}_1 , we take the dot product of its query vector \mathbf{q}_1 with the key vectors in the sequence:

$$[\dots \tilde{\mathbf{a}}_1^T \dots] = [\dots \mathbf{q}_1^T \dots] \begin{bmatrix} | & | & | & | \\ \mathbf{k}_1 & \mathbf{k}_2 & \dots & \mathbf{k}_N \\ | & | & | & | \end{bmatrix} \quad (5.15)$$

The terminology and concepts used in the *Attention Is All You Need* paper by Vaswani et al. (2023) draw from information retrieval, where we have a database or key-value store indexed by keys with corresponding values. To retrieve a value, we compute the similarity between a query and all the keys in the database. The retrieved value can be the one with the maximum score or a weighted sum of values based on these scores, which is the essence of self-attention.

After computing the unnormalized attention scores $\tilde{\mathbf{a}}_1$, we normalize them using the softmax function

$$\mathbf{a}_1 = \text{softmax} \left(\frac{\widetilde{\mathbf{a}}_1}{\sqrt{d_k}} \right) \quad (5.16)$$

Now, we can compute the hidden representation \mathbf{h}_1 by taking a linear combination of all the values in the key-value store, following the same key-store analogy:

$$\left[\begin{array}{c} \text{--- } \mathbf{h}_1^T \text{ ---} \\ \vdots \end{array} \right] = \left[\begin{array}{c} \text{--- } \mathbf{a}_1^T \text{ ---} \\ \vdots \end{array} \right] \left[\begin{array}{c} \text{--- } \mathbf{v}_1^T \text{ ---} \\ \text{--- } \mathbf{v}_2^T \text{ ---} \\ \vdots \\ \text{--- } \mathbf{v}_N^T \text{ ---} \end{array} \right] \quad (5.17)$$

This produces a refined representation \mathbf{h}_i for each input \mathbf{x}_i , incorporating information about the whole sequence.

The combination of Scaled Dot-Product Attention with the three linear transformations for the query, key, and value vectors is what's known as an Attention Head in Transformer models. (Multiple heads can be used to capture different relationships between words, as will be explained later.) Let's implement this concept in PyTorch to gain a better understanding. The following implementation is adapted from the book *Natural Language Processing with Transformers*, by Tunstall et al. (2022).

To implement Scaled Dot-Product Attention, we need to pass in the query, key, and value matrices. Here is a simple implementation of the function in PyTorch:

```
import torch
from math import sqrt
import torch.nn.functional as F

def scaled_dot_product_attention(query, key, value):
    dim_k = query.size(-1)

    unnormalised_scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    scores = F.softmax(unnormalised_scores, dim=-1)
    return torch.bmm(scores, value)
```

In this function:

- We use `torch.bmm` for batch matrix multiplication, which efficiently handles the batch dimension during computation.
- The scores are scaled by dividing by the square root of the dimension of the key vectors (`sqrt(dim_k)`) to stabilize the dot products and prevent them from becoming too large.
- We apply the softmax function to the scores to obtain attention weights and then use these weights to compute a weighted sum of the value vectors.

Next, we'll implement the full attention head. This requires creating the matrices \mathbf{W}_k , \mathbf{W}_q , and \mathbf{W}_v for the key, query, and value transformations, respectively. Each of these matrices transforms the input embeddings into the required dimensions query, key, and value dimensional spaces.

We define these matrices in the following `AttentionHead` class. For simplicity, we assume $d_k = d_q = d_v = \text{head dimension}$, which we refer to as `head_dim` in the code:

```

from torch import nn

class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = scaled_dot_product_attention(
            self.q(hidden_state),
            self.k(hidden_state),
            self.v(hidden_state)
        )

        return attn_outputs

```

In this implementation:

- The `__init__` method initializes three linear transformations for the query, key, and value matrices. These transformations project the input embeddings into the `head_dim` dimensional space.
- In the `forward` method, the hidden states are passed through the linear transformations to obtain the query, key, and value matrices.
- The `scaled_dot_product_attention` function is called with these matrices to compute the attention outputs.

5.3.5.3 From Self-attention to Transformers.

In this section, we will explain how to transition from the self-attention mechanism discussed earlier to the complete Transformer architecture introduced in the influential paper by Vaswani et al. (2023). The architecture of the Transformer model can be visualized in [Figure 5.8](#).

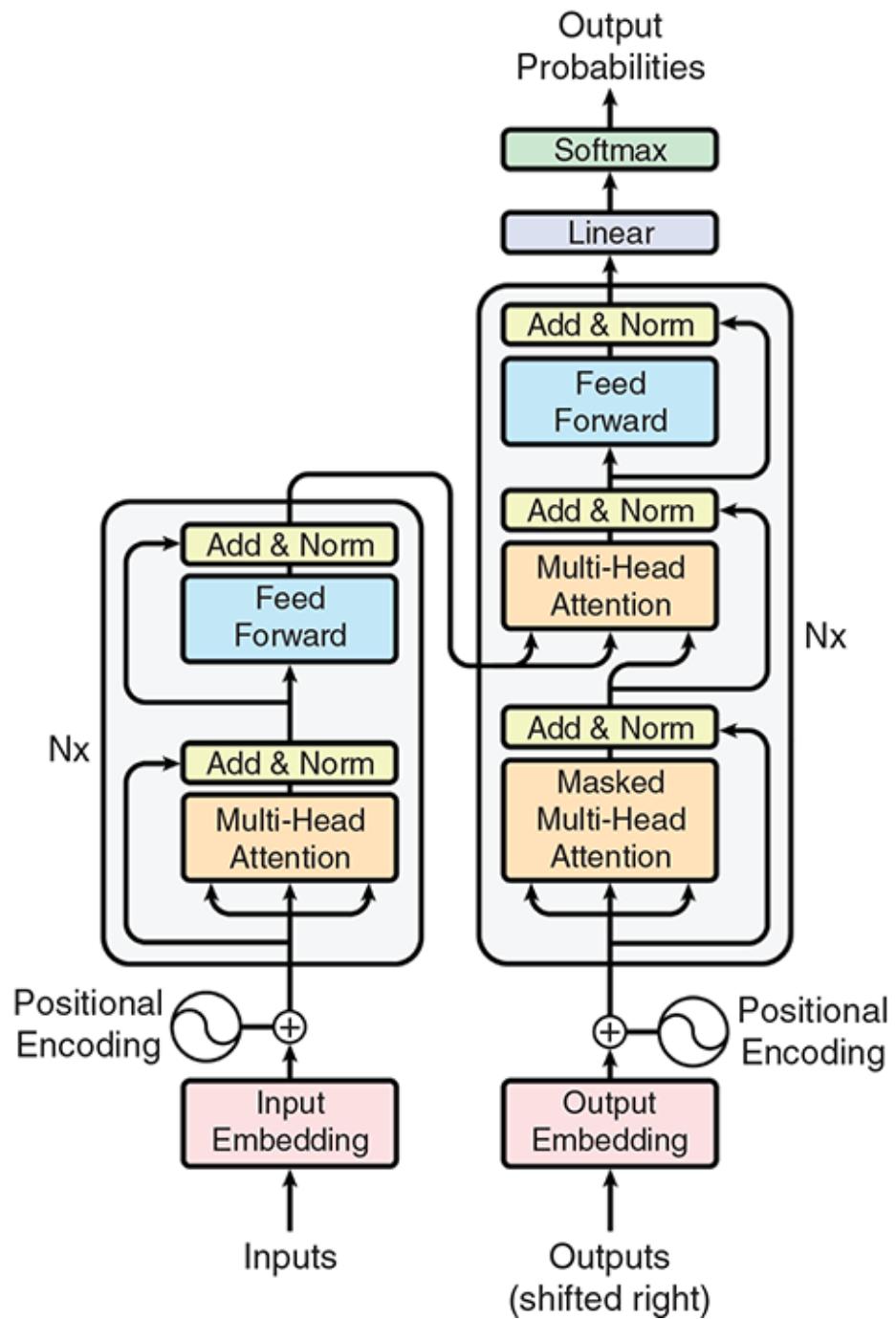


Figure 5.8 The Transformer encoder-decoder architecture, developed for machine translation, processes an input sequence—such as a sentence in the source language—into continuous representations via the encoder. The decoder then generates the translated output sequence, producing one token at a time. Input embeddings correspond to the representation of the source language, while output embeddings correspond to the representation of the target language. The output embeddings are offset by one position to ensure that the prediction for position i depends only on the known outputs at positions less than i . **Figure 1 from Vaswani et al. (2023).**

The Transformer model was originally developed for machine translation tasks, which traditionally follow an encoder-decoder structure. The role of the encoder is to map the input sequence—such as the sentence in the source language—into a sequence of hidden representations. These hidden representations are then used by the decoder to generate the translated output sequence, one token at a time. This process is autoregressive, meaning the decoder generates each token by considering the previously generated tokens as additional inputs. In the Transformer architecture, the encoder is depicted on the left side of [Figure 5.8](#), while the decoder is shown on the right.

As seen in the figure, we need several components for building the complete Transformer model, including Input Embeddings, Multi-Head Attention, and Feed-Forward blocks. These components are fundamental to both the encoder and decoder sections. To build a comprehensive understanding of the Transformer model, we will explore each of these components in details by providing a code implementation of the encoder section.

Additionally, it's important to note that self-attention allows the model to attend to all tokens simultaneously, which is not ideal for tasks such as sequence generation, where the model needs to distinguish between past and future tokens, such as in financial time series. To address this, the self-attention mechanism can be modified to create what is known as causal masked self-attention, which ensures that each token only attends to previous tokens in the sequence. This modification preserves the causal structure needed for generating text in an autoregressive manner.

5.3.5.4 Positional Encodings.

Positional encoding enables self-attention to consider the order of tokens in the sequence, addressing a limitation of the self-attention mechanism, which does not account for positional information in the sequence.

Positional encodings are combined to token embeddings so that the self-attention mechanism can incorporate the order of tokens in the sequence. These encodings can be created using a fixed mathematical formula or learned from data. Common methods for integrating token embeddings with positional encodings include concatenation and addition. The most widely used approach is to add the positional encodings to the token embeddings directly. It's important to note that positional encodings depend only on the position within the sequence.

With positional encodings, the input to the AttentionHead is no longer just token embeddings, but the addition of token embeddings and positional information.

The following code implements a class that creates both token and positional embeddings and adds them together.

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.token_embeddings = nn.Embedding(config.voca
b_size,
                                            config.hidd
en_size)
        self.position_embeddings=
nn.Embedding(config.max_position_embeddings,
                                            config.h
idden_size)

        self.layer_norm = nn.LayerNorm(config.hidden_siz
e, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # Create position IDs for input sequence
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length,
dtype=torch.long).unsqueeze(0)
        # Create token and position embeddings

        token_embeddings = self.token_embeddings(input_i
ds)

        position_embeddings = self.position_embeddings(p
osition_ids)
        # Combine token and position embeddings

        embeddings = token_embeddings + position_embeddi
ngs
        embeddings = self.layer_norm(embeddings)
        embeddings = self.dropout(embeddings)
        return embeddings
```

In this class:

- The `__init__` method creates both token and position embeddings, as well as a layer normalization and dropout layer.
- The `forward` method combines token and position embeddings by addition, normalizes them, and applies dropout.

5.3.5.5 Multi-headed Attention.

The self-attention mechanism involves a single set of projections for generating the query, key, and value representations. In practice, the authors in Vaswani et al. (2023) found it beneficial to use multiple sets of these transformations, known as attention heads. These heads are computed in parallel and then combined by concatenation, which joins the vectors end-to-end, to create a richer representation of the input, see [Figure 5.9](#) for a schematic representation of this process. Multi-head attention enables the model to focus simultaneously on different aspects of the input.

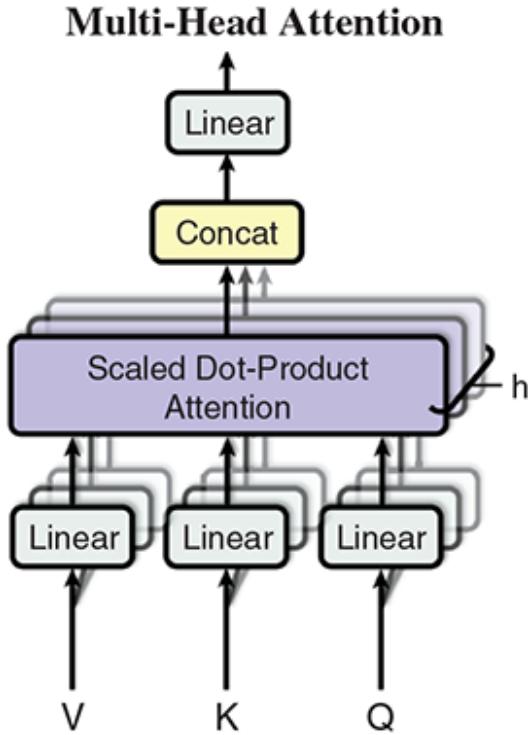


Figure 5.9 Multi-head Attention. Figure 2 (right) from Vaswani et al. (2023).

Multi-head attention allows for different dimensions for the query, key, and value projections. However, in practice, each attention head typically has the same dimension. In addition to the dimensions of these projection matrices, we must specify the number of attention heads. For example, in the original Transformer architecture, the input embedding dimension is 512, and there are 8 attention heads. This configuration sets the dimension for each head as $d_v = d_k = d_q = d_{\text{head}} = 512/8 = 64$.

An implementation of the following `MultiHeadAttention` class demonstrates this concept:

```

class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads
        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for
 _ in range(num_heads)])
        self.output_linear = nn.Linear(embed_dim, embed_
dim)

    def forward(self, hidden_state):
        x = torch.cat([h(hidden_state) for h in
self.heads], dim=-1)
        x = self.output_linear(x)
        return x

```

In this implementation:

- The `__init__` method creates multiple attention heads using the `AttentionHead` class, based on the specified number of heads.
- The `forward` computes attention heads outputs and concatenate them, which is then passed through a final linear layer.

5.3.5.6 The Feed-forward Layer.

The feed-forward layer adds nonlinearity to the model and is applied to the output of the multi-head attention layer. It consists of two linear transformations with a nonlinear activation function in between, as shown here:

```

class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()

        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x

```

This class performs the following steps:

- The `__init__` method defines two linear layers, a GELU activation function, and dropout for regularization.
- The `forward` method applies the first linear transformation, the GELU activation, the second linear transformation, and finally dropout before returning the output.

In the original paper, the dimensionality of the input and output is 512, referred in the code as `hidden_size`, while the inner layer has a dimensionality of 2048, referred in the code as `intermediate_size`.

The multi-head attention, feed-forward layers, and positional encodings together form the building blocks of the Transformer model. They enable the model to capture

complex dependencies and learn rich representations from the input data.

5.3.5.7 Add and Norm Blocks.

As depicted in [Figure 5.8](#), there are Add & Norm blocks integrated into the Transformer architecture.

These blocks perform two key operations: **Add**, which applies a residual connection (i.e., adding a layer's input to its output to improve gradient flow and facilitate training), and **Norm**, which involves layer normalization (see Goodfellow et al., 2016, for more details). As shown in the figure, the authors use a residual connection around both Multi-head Attention and feed-forward blocks.

5.3.5.8 The Transformer Encoder Layer.

Now that we've covered the core components of the Transformer architecture, we are ready to implement the Transformer encoder layer, as shown in the left of [Figure 5.8](#). The Encoder layer, also known as Encoder block, consists of an embedding layer, a multi-head attention mechanism, and a feed-forward network. As described, batch normalization and residual connections are also included.

Following is the implementation of the `TransformerEncoderLayer`, incorporating layer normalization and skip connections for better gradient flow:

```

class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # Apply layer normalization and then copy input
        # into query, key, value
        hidden_state = self.layer_norm_1(x)
        # Apply attention with a skip connection
        x = x + self.attention(hidden_state)
        # Apply feed-
        # forward layer with a skip connection
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x

```

A note on regularization: As described by the authors of the Transformers paper, and as you may have noticed in the code implementation, dropout (see [Chapter 3](#)) is used for regularization. Dropout is applied around the Add & Norm blocks, as well as to the sum of the embeddings and positional encodings in both the encoder and decoder stacks (Vaswani et al., 2023).

5.3.5.9 The Complete Transformer Encoder.

The Transformer model is constructed by stacking multiple TransformerEncoderLayer modules. Each layer processes the input embeddings and refines them further, allowing the model to learn complex relationships within the data.

Here's how we can implement the complete Transformer encoder:

```
class TransformerEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = Embeddings(config)

        self.layers = nn.ModuleList([TransformerEncoderLayer(config)
                                    for _ in range(config.num_hidden_layers)])

    def forward(self, x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

5.3.5.10 Model Objective.

Now that we have defined the Transformer encoder architecture, the next question is: What tasks can we solve with it? how do we structure the model's output for solving these tasks and train the model?

The typical approach is to use the Transformer encoder model as the core *body* of our model and attach either a decoder, as in language translation tasks, or a specific *head* that is suited to the task at hand. For instance, for a classification task, we can attach a specific head, a network layer that takes the output of the Transformer and generates a probability distribution over the possible classes. This allows us to use an optimization algorithm to find the model parameters that best explain the relationship between inputs and targets in our dataset.

Let's consider a few examples to understand this concept better:

- **Sentiment Analysis:** For this task, we would use a classification head that processes the output of the Transformer model and predicts the sentiment (for example, positive, negative, or neutral) of a given input text. The model learns to associate specific patterns in the text with specific sentiments.
- **Language Modeling:** The BERT model, introduced by (Devlin et al. 2019), is a good example of this approach. BERT stands for Bidirectional Encoder Representations from Transformers and uses the Transformer encoder architecture. One of BERT's primary training objectives is the Masked Language Model (MLM) task, where certain words in a sentence are randomly masked, and the model is trained to predict these missing words. Specifically, 15% of the words in a sentence are masked. Of these, 80% are replaced with the [MASK] token, 10% remain unchanged, and the final 10% are substituted with a random word. To solve this task, we would use a classification head that processes the output of the Transformer and generates a probability distribution over the input's vocabulary.

To adapt the Transformer encoder for a classification task like sentiment analysis, we add a classification head on top of the encoder. As an example, the head implemented here consists of a dropout layer followed by a fully connected layer that maps the encoded representation to the desired number of labels:

```

class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)

        self.dropout = nn.Dropout(config.hidden_dropout_
prob)

        self.classifier = nn.Linear(config.hidden_size,
config.num_labels)

    def forward(self, x):
        x = self.encoder(x)
        [:, 0, :] # select hidden state of [CLS] token
        x = self.dropout(x)
        x = self.classifier(x)
        return x

```

- `self.encoder`: This is the Transformer encoder that processes the input sequence.
- `self.dropout`: This layer is used to prevent overfitting during training.
- `self.classifier`: This fully connected layer maps the encoder's output to the desired number of classes.

As just shown, the classification model leverages the Transformer encoder to create powerful representations of the input text. By attaching a classification head, we can train this model for various tasks, such as sentiment analysis, language modeling, question answering, and more. However, the typical way these models are trained to solve downstream tasks, such as sentiment analysis, involves a two-step training process. First, in the pre-training phase, the Transformer encoder is trained on a language modeling task. Then, in the second step, known as fine-tuning, the model is further trained on a specific dataset related to the task, and a classification or regression head is added to solve specific

problems. More details on this process, along with practical examples, can be found in [Chapters 9](#) and [10](#).

In fact, the structure described here is essentially an implementation of the BERT model, which has become a foundational tool for many NLP applications due to its versatility and powerful performance across a wide range of tasks.

For complementary purposes, the accompanying notebook [*DecoderOnly Transformers—Notebook*] showcases an implementation of a decoder-only transformer model, specifically GPT, to complement the discussion of decoder-only transformers presented here.

5.3.6 From NLP Transformer to the Time-series Transformers

Given the great success of Transformer models in capturing long-range dependencies in sequential data—such as text, audio, and video—it is natural to wonder if they can also be applied to time-series applications, as time-series data, like text, is inherently sequential.

Transformers are known for several advantages, which we will discuss in more detail in [Chapter 10](#). To mention a few, these models improve their performance in NLP tasks as model size, training data volume, and training time increase (see Kaplan et al., 2020). They are also highly parallelizable and adaptable for multiple downstream tasks, as demonstrated by Devlin et al. (2019). These qualities are motivating the industry to explore whether Transformers can achieve a similar level of success in modeling time-series problems.

For language modeling tasks, Transformers are trained with the objective of predicting the next token in a sequence based on the previous tokens. This is conceptually similar to

the time-series forecasting task, where we want to predict the next element of the series based on previous values, or perform what is called a “one-step-ahead forecast.” In time-series terminology, we could say that the language modeling task is akin to forecasting a discrete time series where the elements in the sequence take values from a finite vocabulary, with a time horizon of one. Given this similarity in objectives, we might wonder what modifications are needed to apply Transformers to time series.

As hinted, one key difference is that most of the time, the values in a time series are unbounded, continuous quantities, unlike in a text sequence, where elements (called tokens) take values from a finite vocabulary. A straightforward idea for adapting Transformers to time-series data is to convert the continuous values of the series into a finite set of values, or tokens, drawing an analogy to text applications, similar to how words are represented in language tasks. This approach is taken in *Chronos: Learning the Language of Time Series* (Ansari et al., 2024).

5.3.6.1 Discretizing Time-series Data: The Chronos Approach.

In the paper *Chronos: Learning the Language of Time Series* (Ansari et al., 2024), the authors propose an approach for using Transformers on time-series data by **tokenizing** time-series values. This is achieved by first normalizing and then quantizing the time series, transforming continuous values into discrete “tokens” that can be processed by standard Transformer architectures.

1. Normalization: Each time series is normalized to mitigate scaling differences between series and improve model optimization. The authors use mean scaling, in which each series is normalized by the mean absolute historical

value within the “historical context”—the number of inputs to the Transformer.

2. Quantization: The normalized values are then quantized into a fixed number of bins, converting continuous values into a discrete set of values, or tokens. These discrete tokens represent the time series in a way similar to how text is represented by a sequence of tokens.

Once each value has been tokenized, **token embeddings** and **positional embeddings** are created in the same way as for language modeling tasks and combined by addition to form the input embeddings for the Transformer. At inference time, since the model outputs tokens, a process of **dequantization** and **denormalization** is applied to convert discrete token predictions back into continuous values.

This method allows the use of standard Transformer architectures, such as T5 (Raffel et al., 2020; encoder-decoder) and GPT-2 (Radford et al., 2019; decoder-only), without architectural modifications. The authors of Chronos demonstrated that this approach is effective across a wide range of time-series data, achieving strong results on out-of-sample data. To train their models, the authors compiled time-series data from various domains, including retail, energy, finance, healthcare, and climate science, enhancing generalization by incorporating synthetic data generated through data augmentation techniques.

In the accompanying notebook [*Chronos Notebook*], we explore the use of Chronos on the Exchange Rate dataset, which contains daily exchange rates for currencies of eight countries (Australia, United Kingdom, Canada, Switzerland, China, Japan, New Zealand, and Singapore) from 1990 to 2016 (Exchange Rate data). Part of this data was used to train Chronos, and according to the references provided in their repository and paper, we evaluate Chronos’s

performance on the unseen part of the data, highlight some of its limitations, and compare it with some simple baselines.

5.3.6.2 Continuous Input for Transformers: The Lag-Llama Approach.

An alternative approach to discretizing the time series is to treat the data as it is, using continuous values so that the network takes continuous inputs and outputs continuous values. This approach not only requires preprocessing the input to create “token embeddings,” but also adjustments to the output, as we are now solving a regression task rather than a classification task, as in language modeling. This is the approach taken in the paper *Lag-Llama: Towards Foundation Models for Probabilistic Time Series Forecasting* (Rasul et al., 2024).

In this work, the authors adapt the Llama model (Touvron et al., 2023), a decoder-only Transformer architecture, to output continuous values rather than discrete tokens by introducing a final layer that predicts the parameters of continuous distributions, specifically a t-distribution, rather than the parameters of a categorical distribution typical in multiclass classification problems.

The modifications in Lag-Llama include the following:

1. Probabilistic continuous output layer: Instead of predicting the parameters of a categorical distribution that allocates probability mass over a finite vocabulary of tokens, Lag-Llama outputs the parameters of a t-distribution, allowing the model to make probabilistic forecasts over continuous values.
2. Lagged and date-time features: In addition to using the current time step, Lag-Llama incorporates lagged features—previous values at various intervals (e.g., quarterly, monthly, weekly)—to enrich the input and

capture seasonality and periodic patterns. The model also includes features such as the hour, day, and month, etc., derived from the timestamp, which provide additional contextual information about each time point. The authors also include summary statistics of the series to provide additional context.

3. Normalization and outlier control: Lag-Llama normalizes each time series and mitigates outliers using the interquartile range (IQR).

By enriching the current input with lagged and date-time features, the authors provide the Llama model with more contextual information to capture long-term and short-term dependencies effectively. In our experience with forecasting tasks involving large datasets of multivariate time series, this type of feature engineering helps models capture seasonal characteristics and improves forecasting performance.

In the Lag-Llama approach, the current and lagged values of the series, along with engineered date-time features, are concatenated into a vector. See [Figure 5.10](#) for an illustration of how these features are constructed. This vector is then projected into the model’s “token embedding” space, creating an analogous “token representation” as in NLP applications, which serves as the input to the Transformer model, as shown in [Figure 5.11](#).

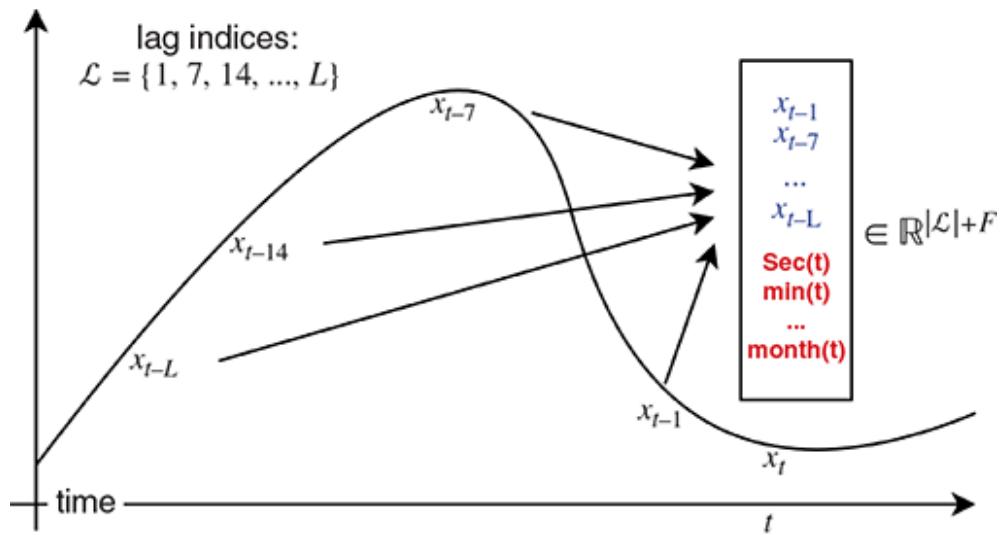


Figure 5.10 An illustration showing how the current and lagged values of the series, combined with engineered date-time features, are concatenated into a vector. Each component of the vector represents $|\mathcal{L}|$ past value of x_t (shown in blue), along with F temporal covariates (date-time features) derived from the timestamp (shown in red). Figure 1 from Rasul et al. (2024).

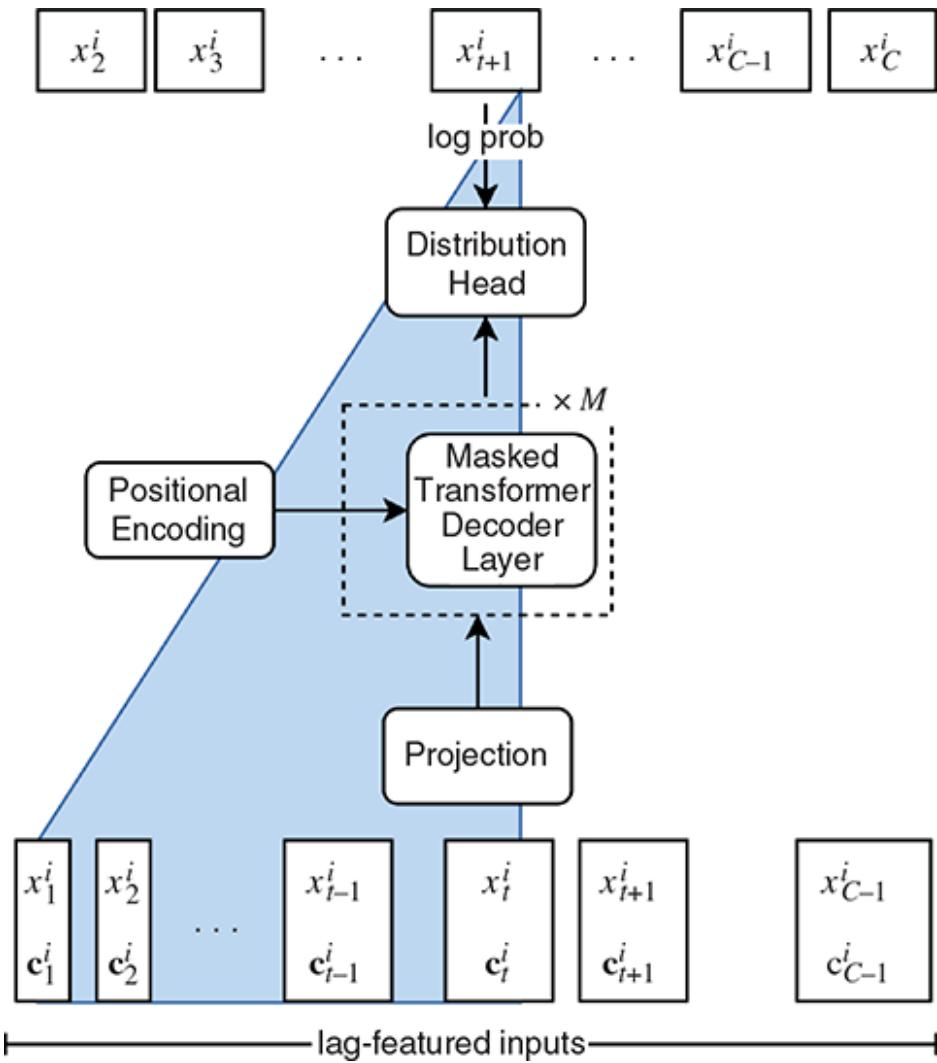


Figure 5.11 An illustration showing how the model input, comprising the time-series value at timestep t with $|\mathcal{L}|$ lags, F date-time features, and summary statistics, is projected into the transformer’s “token embedding” space, creating a “token representation” similar to NLP applications. Figure 2 from Rasul et al. (2024).

In the accompanying notebook [Lag-Llama Notebook], we explore the use of Lag-Llama on the same Exchange Rate dataset. For Lag-Llama, since this dataset has not been used for training the model according to the authors, we will

evaluate its “zero-shot” performance. As an additional exercise, we will fine-tune the model on this dataset and observe how its performance is affected. Similar to Chronos, we will highlight some of its limitations and compare its performance with simple baselines.

5.3.6.2.1 Innovations and Approaches.

The examples of Chronos and Lag-Llama illustrate some of the most common techniques for adapting Transformer architectures for time-series applications, which in these cases required few to no modifications to existing Transformer architectures. In these cases, much of the work revolves around techniques for preprocessing time-series data and enriching contextual information, using methods such as normalization, quantization, and the addition of lagged values and datetime features. We also saw how different preprocessing choices, such as applying quantization or not, affect how the problem is framed: as a classification problem (as in Chronos, which requires no changes to standard Transformer architectures and treats the problem similarly to text tasks) or as a regression problem (as in Lag-Llama, where the authors modified the last layer of the architecture to model a probability distribution over continuous values).

While these examples showcase a few interesting adaptations, they are by no means representative of the full range of approaches that can be applied to improve model performance. In particular, there are approaches that focus on improving the space and time complexity of Transformers, which we have not yet discussed.

Over the past few years, numerous Transformer architectures for time series have been proposed. To the best

of our knowledge, the main areas of work are around the following:

1. Input Normalization and Contextual Information: This area includes scaling time-series data to help model optimization, applying various normalization techniques, handling outliers, and quantizing the series. Context can be enriched by incorporating lagged values or creating datetime features to capture important seasonal and calendar effects, or incorporating macroeconomic indicators, as well as indicators from markets or indices that are not part of the target time series—some of which were outlined in [Chapter 3](#), among others. This information is typically projected to create “token embeddings,” which serve as inputs to the Transformer. For example, some models use one-dimensional convolutional layers as projection layers, as seen in the Informer architecture (Zhou et al., 2021).
2. Positional Encoding: Transformers are permutation-invariant by nature, posing a challenge when handling sequential data like time series. Positional encoding is crucial to convey order information in NLP tasks and becomes even more vital for time-series applications, where the sequence order is essential. Solutions range from fixed positional encodings, as used in the original Transformer (Vaswani et al., 2023), to learnable positional embeddings like the one discussed in [Section 5.3.5.4](#).
3. Attention Layer efficiency: Improving the efficiency of the attention mechanism is an active area of research, especially for time-series data, which often involves long historical sequences and forecasting over extended horizons. Improving computational complexity of this layer can help reduce inference latency and help us to meet the system requirements of specific applications. Recall from our earlier discussion in [Section 5.3.5.2](#) that

the vanilla Transformer’s attention mechanism (Vaswani et al., 2023), computes a dot product between each query and all keys, resulting in a time complexity of $\mathcal{O}(N^2)$ for N inputs. To reduce this complexity, several methods have been proposed. For example, the Informer model (Zhou et al., 2021) reduces the computational cost of the attention layer by computing attention for a subset of $\log N$ “active queries,” rather than for all N queries, thus, achieving a time complexity $\mathcal{O}(N \log N)$.

4. New transformers architecture variants: Researchers have developed architecture variants to account for the multi-resolution nature of time series. For example, the Informer architecture down-samples the series by inserting max-pooling layers between attention blocks (Zhou et al., 2021). The Pyraformer model (Liu et al., 2021) introduces a pyramidal attention module that summarizes features at different resolutions, enabling it to model temporal dependencies over varying time scales effectively.

For a more detailed overview of the most well-known Transformer architectures for time series at the time of writing, we recommend the excellent resource *Transformers in Time Series: A Survey* (Wen et al., 2023). Additionally, for an application of transformers as building blocks in portfolio construction, see Cong et al. (2021).

5.4 Model Fitting

Autoregressive models are often referred to as Maximum Likelihood models, since they primarily rely on Maximum Likelihood Estimation (MLE) for parameter estimation. All the models discussed in this chapter use MLE as the main method for parameter learning. Once we define a model class parametrized by θ , denoted as p_θ , it becomes essential to

establish a metric that evaluates the model's ability to approximate the data probability distribution, denoted as p_{data} . Of course, we discussed MLE fitting already in [Chapter 3](#), but here is a new way to think about it.

Considering that our goals include computing probability densities, determining hidden structures, and generating new samples, determining a single model scoring metric that fulfills all these objectives can be challenging. Intuitively, if a model can closely approximate the distribution p_{data} , it should theoretically perform well in the other tasks. Therefore, our objective is to align the model distribution p_{θ} as closely as possible with p_{data} .

To quantify the closeness between two distributions, we use the Kullback-Leibler divergence, $\text{KL}(P\|Q)$, defined as:

$$\text{KL}(P\|Q) = \mathbb{E}_{\mathbf{x} \sim P} \left[\log \frac{P(x)}{Q(x)} \right] \quad (5.18)$$

Our goal is to minimize the Kullback-Leibler divergence between the empirical data distribution and our model's distribution:

$$\begin{aligned} \text{KL}(p_{\text{data}}\|p_{\theta}) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\theta}(\mathbf{x})} \right] \\ &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log p_{\text{data}}(\mathbf{x}) - \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log p_{\theta}(\mathbf{x}) \end{aligned} \quad (5.19)$$

The term $\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \log p_{\text{data}}(\mathbf{x})$ does not depend on θ and is thus irrelevant to the optimization process. Hence, the objective function, cost function, or score to minimize is given by:

$$\underset{\theta}{\operatorname{argmin}} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} - \log p_{\theta}(\mathbf{x}) \quad (5.20)$$

Given that we only have access to samples from p_{data} , the empirical estimation is:

$$\operatorname{argmin}_{\theta} \mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_n \in \mathcal{D}} -\log p_{\theta}(\mathbf{x}_n) \quad (5.21)$$

This formulation corresponds to Maximum Likelihood Estimation, or minimizing the Negative Log Likelihood, which aims to increase the probability mass at points present in our training dataset while assigning lower probabilities to those that are absent.

To minimize this objective function, the typical optimization algorithm used is Stochastic Gradient Descent (SGD):

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \widetilde{\mathcal{L}}(\theta) \quad (5.22)$$

Here, $\widetilde{\mathcal{L}}(\theta)$ represents the loss computed using a batch or subset of the data, facilitating efficient gradient computation and model updates.

5.5 Conclusions

As we have explored in this chapter, autoregressive models are powerful tools for modeling high-dimensional sequences. We began with classical approaches such as linear AR models, progressed to RNNs, and concluded with the groundbreaking Transformers that have reshaped the fields of NLP, speech, computer vision, and more. However, at the time of writing, Transformers for time-series applications have not yet reached the level of performance observed in other domains such as text and vision tasks.

Adapting Transformers for time series may require further optimization and empirical work to achieve the same level of success. Throughout this chapter, we have highlighted points that could provide pathways for improvement. By addressing the unique challenges posed by time-series data, future research and development can help bridge the current performance gap. Based on findings in the literature, simple

customized solutions, such as solutions based on linear AR models, can still outperform them, particularly when forecasting financial time series. However, Transformer models offer many advantages, and their use depends on your main objective—whether it is performance, scalability for handling multiple heterogeneous time series, custom versus general solutions, or other factors.

When it comes to applications of Transformers for developing trading strategies, they are powerful tools for preprocessing alternative data, providing valuable insights for systematic strategies or discretionary investment managers. As a use case, in [Chapter 9](#), we will demonstrate how to leverage this powerful model for processing alternative data and develop systematic trading strategies.

Chapter 6

Deep Latent Variable Models

In the previous chapter, we discussed how autoregressive models leverage previous values in a sequence to represent highly complex, high dimensional probability distributions. In contrast, latent variable models adopt a distinct approach to approximating the probability distribution p_{data} . Latent variable models do not use previous values in the sequence as AR models do; instead, they assume the existence of an unobservable hidden variable that influences every data point in the dataset.

In this chapter, we focus on models where the knowledge of the hidden or latent variables **partially explains the observed data**. Here, the relationship between the hidden variables and the observations is not fully deterministic—we introduce some uncertainty in the modeling step regarding how the latent variables generate the observed data. On the other hand, models where the latent variables completely determine the observed data—where a deterministic, invertible mapping exists between the hidden variables and the data—will be the subject of [Chapter 7](#). These are known as flow models or invertible models. In [Chapter 8](#), we will introduce another type of latent variable model: Generative Adversarial Networks (GANs). Unlike the models discussed here or in the next chapter, GANs do not explicitly model the probability distribution of the data.

In this chapter, we focus on models where the dimensionality of this latent variable is significantly smaller than that of the observed data. Introducing hidden variables in our models helps model a high-dimensional

multimodal distribution as the integral of the product of two simpler distributions. Commonly, these simpler distributions belong to the exponential family, which simplifies the overall modeling process while retaining flexibility in capturing the underlying data structure. The Hidden Markov Model that we introduced in [Chapter 3](#) is obviously one simple example of such latent variables model (where there is some dynamic between the latent variables), but we shall see that Principal Component Analysis (PCA) introduced there can also be viewed as an example of latent variables model. Deep latent variable models, such as the well-known variational autoencoders discussed in this chapter, belong to the category of explicit models that estimate densities approximately, as shown in [Figure 6.1](#). This chapter will leverage the full machinery of deep learning to create a much richer variety of these models.

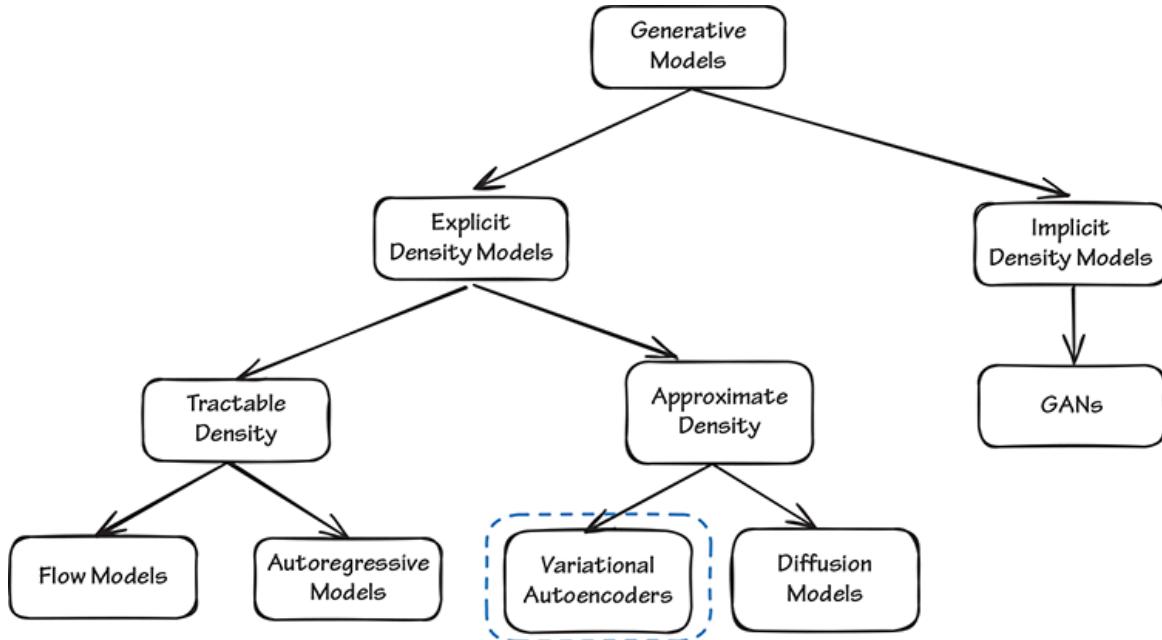


Figure 6.1 Model taxonomy: variational autoencoders.

6.1 Introduction

To illustrate the utility of latent variable models, consider their application in finance, particularly for modeling and estimating the covariance matrix of multivariate time series. Covariance matrices are fundamental in finance, utilized for tasks such as regression analysis, risk estimation, portfolio optimization, and scenario simulation through Monte Carlo methods (López de Prado, 2020).

Let's assume we have a lengthy history of returns from D time series, each representing an asset's historical returns. A straightforward approach to modeling these multivariate time series is to ignore individual time-series serial correlation and assume a constant covariance across assets over time. At any time index t , the returns could be modelled as a draw from a constant probability distribution, for example, $\mathbf{x}_t \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\mathbf{x}_t \in \mathbb{R}^D$

represent the return at time index t , μ is the mean return vector, and Σ is the covariance matrix of returns.

To provide context, consider modeling the daily returns of the S&P 500 index components, thereby setting the number of time series to $D = 500$. Here, the simplest estimator for the covariance matrix would require computing $D(D + 1)/2 = 500(501)/2 = 125,250$ covariance values. This large number of parameters can become problematic if the available daily data are limited.

Another way to think about modeling \mathbf{x}_t is to assume it is influenced by a smaller set of underlying, unobserved factors that partially explain the observations.

These factors may look similar to traditional multi-factor models, such as the Fama-French three-factor model we discussed in [Chapter 2](#), which explains asset returns using factors like the market, Small Minus Big (SMB), and High Minus Low (HML) factors. However, a key difference between these traditional models and the ones we will explore in this chapter is that, in the traditional case, the factors are observable—they can be measured, and often, they can even be traded, providing significant value to traders. In contrast, the models introduced in this chapter deal with unobservable factors. These are hidden, or latent, variables, terms we will use interchangeably in this chapter, and which are common in the literature.

For instance, Factor Analysis (FA), a type of latent variable model, decomposes the covariance matrix as $\Sigma = \Lambda\Lambda^T + \Sigma_0$, where $\Lambda \in \mathbb{R}^{D \times M}$ is known as the loading matrix, M being the number of factors (a hyperparameter in FA) with $M \ll D$, and Σ_0 is a $D \times D$ diagonal matrix. In financial terminology, we can think of $\Lambda\Lambda^T$ as the systematic covariance matrix, representing the part of the asset's returns variance explained by the factors, and Σ_0 as the

idiosyncratic covariance matrix—the variance of asset's returns not explained by the factors. This type of variance decomposition is particularly useful for risk modeling in portfolio construction applications, where we might be interested in reducing the exposure of our strategy to certain factors or placing an upper limit on systematic variance, etc. This formulation of variance decomposition allows constraints like these to be easily incorporated into the portfolio optimization process.

This method significantly reduces the number of parameters to estimate, from $D(D + 1)/2$ to $D \times M + D$, from $\mathcal{O}(D^2)$ to $\mathcal{O}(D \times M)$. For example, if $M = 10$, the parameter count drops to 5,010 compared to 125,250—a 25-fold reduction. Because this modeling approach assumes that observations are explained by a smaller set of factors, the covariance matrix is inherently low rank. This not only reduces the number of parameters to estimate but also decreases the memory required to store the covariance matrix compared to simpler modeling approaches.

What is fascinating about latent variable models is that, even though the factors are not directly observed, by making probabilistic assumptions about how they behave, we can infer their values and estimate the associated model parameters. Hamlet remembers when he first studied these kinds of models; I was absolutely fascinated by them. In the context of FA, things get even more interesting when we introduce dynamics into the latent variables, leading to models such as the well-known Kalman Filter.

As we will see in this chapter, when we consider Deep Latent Variable Models, the algorithms used to estimate these models become even more interesting. The concepts and tools required to develop them are applied in many fields, including statistics, machine learning, optimization,

information theory, etc., and have applications across many areas.

6.2 Latent Variable Models

Latent variable models aim to represent high-dimensional data distributions using two simpler distributions. The model involves a latent variable \mathbf{z} and an observed variable \mathbf{x} , where \mathbf{z} is an M -dimensional random vector and \mathbf{x} is a D -dimensional random vector. By design, we choose $M < D$ to ensure a lower dimensionality for the latent space.

A latent variable model is defined by the joint probability distribution of the latent and observed variables as $p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$. The model is specified by two components: a probability distribution over the latent variables, $p(\mathbf{z})$, called the prior distribution, and a conditional distribution over the observed data given the latent variables, $p(\mathbf{x}|\mathbf{z})$, which maps \mathbf{z} from the latent space to the data space. Sampling in latent variable models is relatively straightforward, as it follows the data generation process described by:

$$\begin{aligned}\mathbf{z}_t &\sim p(\mathbf{z}_t) \\ \mathbf{x}_t | \mathbf{z}_t &\sim p(\mathbf{x}_t | \mathbf{z}_t)\end{aligned}\tag{6.1}$$

The challenge lies in choosing the prior and conditional distributions to represent our hypotheses about the data generation process. These choices also determine the specific methods used for performing latent variable inference and parameter estimation.

As examples of these particular choices, we will briefly examine traditional latent variable models such as Factor Analysis (FA), Probabilistic Principal Component Analysis

(PPCA), and Gaussian Mixture Models (GMM) before moving on to the main topic of this chapter: Deep Latent Variable Models.

6.3 Examples of Traditional Latent Variable Models

6.3.1 Factor Analysis

As we have already seen, Factor Analysis (FA), see Bishop (2006), is a foundational model that expresses the modeling assumption that data can be explained by a reduced number of factors. This approach has interesting applications, such as covariance matrix estimation and dimensionality reduction we just saw, among others. Similarly, PPCA, which we will also examine later, builds on this probabilistic framework. In this section, we will go into the probabilistic formulation of these methods.

In FA, we assume that both the marginal and conditional distributions are Gaussian and are defined as follows:

$$\begin{aligned} p(\mathbf{z}_t) &= \mathcal{N}(\mathbf{z}_t; \mathbf{0}, \mathbf{I}) \\ p(\mathbf{x}_t | \mathbf{z}_t) &= \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu} + \boldsymbol{\Lambda}\mathbf{z}_t, \boldsymbol{\Sigma}_0) \end{aligned} \tag{6.2}$$

where $p(\mathbf{z}_t)$ is the prior distribution over the latent variable, and \mathbf{z}_t , and $p(\mathbf{x}_t | \mathbf{z}_t)$ represents the conditional probability over the latent variable \mathbf{z}_t given the observed data \mathbf{x}_t .

In the FA literature, the matrix $\boldsymbol{\Lambda} \in \mathbb{R}^{D \times M}$ is referred to as the loading matrix, and it captures the correlations between the observed variables. The vector of means $\boldsymbol{\mu} \in \mathbb{R}^D$ can be thought as a bias term. The matrix $\boldsymbol{\Sigma}_0$ is

diagonal with D elements and represents the noise covariance matrix. The values along the diagonal are the independent noise variances for each variable. In finance, this is commonly referred to as idiosyncratic variance, as described in Bodie et al. (2018).

To obtain the marginal distribution over \mathbf{x}_t , we integrate out all possible values of the latent variable \mathbf{z}_t :

$$p(\mathbf{x}_t) = \int p(\mathbf{x}_t, \mathbf{z}_t) d\mathbf{z}_t = \int p(\mathbf{x}_t | \mathbf{z}_t) p(\mathbf{z}_t) d\mathbf{z}_t$$

In FA, where both $p(\mathbf{x}_t)$ and $p(\mathbf{x}_t | \mathbf{z}_t)$ are Gaussian, this integral has a closed-form solution. However, a simpler way to compute the marginal distribution and to avoid the previous integral is to express \mathbf{x}_t as:

$$\mathbf{x}_t = \boldsymbol{\mu} + \Lambda \mathbf{z}_t + \boldsymbol{\epsilon}_t$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_0)$.

Using the property that Gaussians are closed under addition and multiplication, the marginal distribution over \mathbf{x}_t is also Gaussian. Therefore, we only need to compute the mean and covariance of \mathbf{x}_t , which are given by:

$$\begin{aligned}\mathbb{E} [\mathbf{x}] &= \boldsymbol{\mu} \\ \text{Cov} [\mathbf{x}] &= \boldsymbol{\Sigma} = \Lambda \Lambda^T + \boldsymbol{\Sigma}_0\end{aligned}$$

Thus, the marginal distribution over \mathbf{x}_t is:

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}),$$

where $\boldsymbol{\Sigma} = \Lambda \Lambda^T + \boldsymbol{\Sigma}_0$, retrieving the formulation presented in the introduction of this chapter.

Parameter estimation in this model follows a well-known approach: the parameters $\boldsymbol{\mu}$, Λ , and $\boldsymbol{\Sigma}_0$ are estimated from data using the Expectation-Maximization (EM) algorithm.

For more details on parameter estimation for FA, see Bishop (2006).

6.3.2 Probabilistic Principal Component Analysis

Probabilistic Principal Component Analysis, or Probabilistic PCA (PPCA), is the probabilistic version of PCA proposed by Bishop and Tipping (2001). As we know, PCA is a widely used technique in data analysis and processing, yet it does not rely on a probabilistic model. Particularly in finance, PCA is employed to identify statistical factors that can be utilized for asset prediction (Chan, 2017) or risk modeling, where these factors have shown performance comparable to fundamental ones (see Paleologo [2021]).

This model is closely related to the Factor Analysis. In PPCA, both the prior and conditional distributions are Gaussian and are defined by the same equations as in [Equation 6.2](#). However, the key difference between the two methods is that PPCA assumes Σ_0 is a constant diagonal matrix, denoted as $\Sigma_0 = \sigma^2 \mathbf{I}$.

This assumption provides a computational advantage while preserving the interpretability of Factor Analysis.

6.3.2.1 Example: Comparing PCA and Factor Analysis for Latent Space Recovery.

As an illustrative use case for Factor Analysis (FA) versus Probabilistic PCA (PPCA), let's explore an example adapted from the sklearn documentation (https://scikit-learn.org/stable/auto_examples/decomposition/plot_pca_vs_fa_model_selection.html). As we have seen, FA and PPCA are closely related: both assume Gaussian prior and

Gaussian conditional distributions. The key difference lies in their treatment of **idiosyncratic variance**.

- PPCA assumes homoscedastic noise, meaning the variance is the same for all variables (identity covariance matrix).
- Factor Analysis assumes heteroscedastic noise, where each variable has its own variance (diagonal covariance matrix).

This example illustrates the ability of PCA (the limiting case of PPCA with zero variance) and Factor Analysis to recover the correct latent dimension in a low-rank simulated dataset with additive noise. We examine two cases, one with homoscedastic noise, with the same variance for each variable (identity covariance), and a second case with heteroscedastic noise, with different variance for each variable (diagonal covariance).

For the experiment, we generate 100-dimensional data with an underlying low-rank structure (latent dimension = 10).

Results regarding model selection for homoscedastic noise are illustrated in [Figure 6.2](#). In this case, the low-rank data has been corrupted with noise that has equal variance across all variables. Both PCA (blue) and Factor Analysis (red) successfully recover the correct latent dimension of 10 when performing model selection.

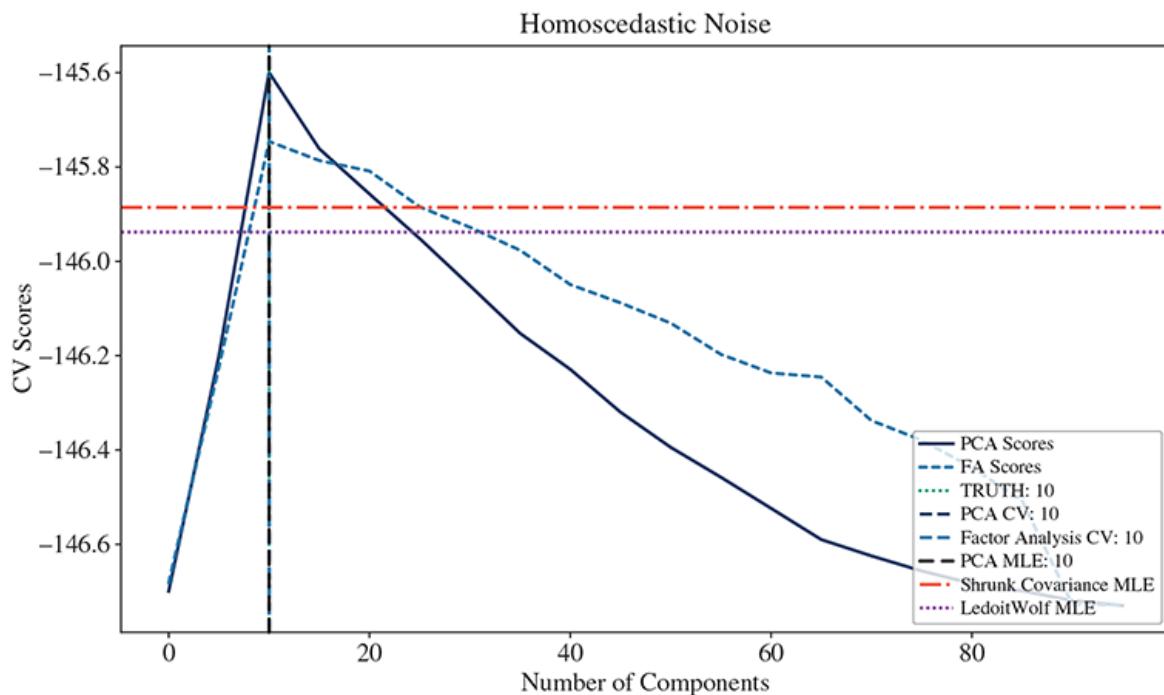


Figure 6.2 Illustration of model selection for homoscedastic noise.

On the other hand, results regarding model for heteroscedastic noise are illustrated in [Figure 6.3](#). Here, the low-rank data are corrupted with noise that has varying variance across variables. This leads to a more interesting result:

- Factor Analysis (red) correctly identifies the latent dimension as 10, aligning with its assumption of heteroscedastic noise.
- PCA (blue), on the other hand, overestimates the latent dimension.

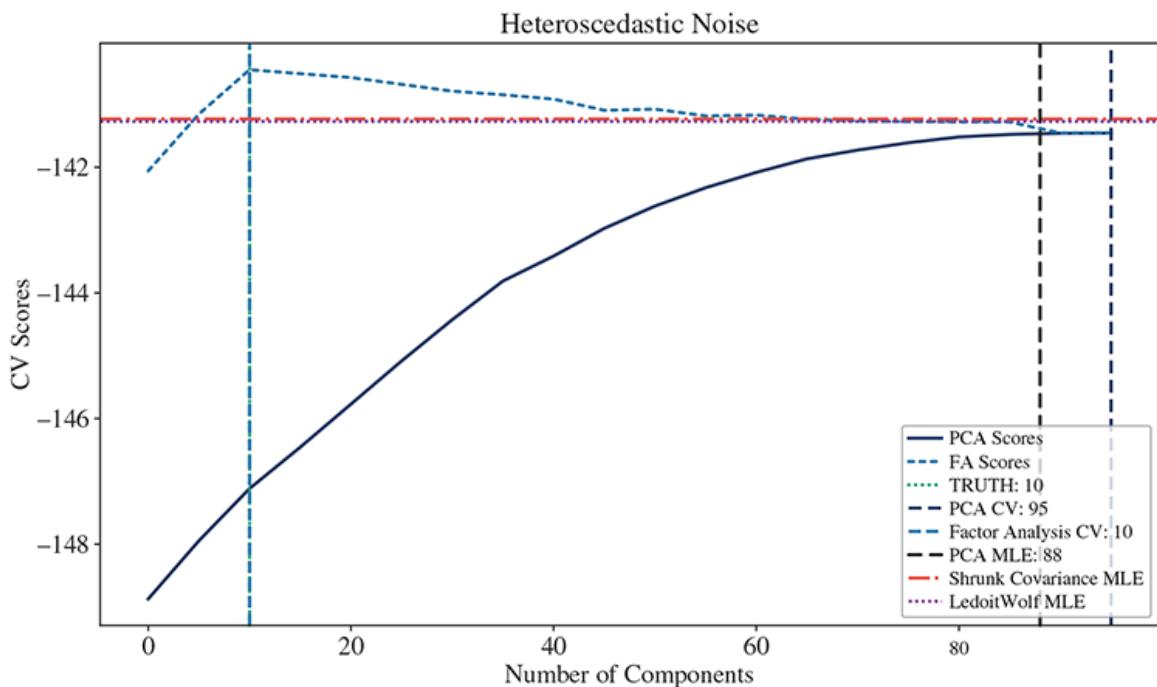


Figure 6.3 Illustration of model selection for heteroscedastic noise.

These results demonstrate how the assumptions behind each method influence their ability to capture the true latent structure in the presence of different noise characteristics.

6.3.2.2 Advantages of Probabilistic Approaches (PPCA/FA) over PCA

There are several advantages of using the probabilistic approach, either with PPCA or FA, over PCA, also called the non-probabilistic PCA. Some of the main ones include the following:

- **Estimation with Missing Data:** PPCA and FA provide a natural framework for parameter estimation when some entries of a vector in the dataset $\mathbf{x}_n \in \mathbb{R}^D$ are missing. This contrasts with the non-probabilistic PCA,

where parameter estimation with missing data might require data imputation techniques.

- **Mixtures of Probabilistic Principal Component Analysis Models:**

Sometimes, a single PCA projection is insufficient to explain the data. Why not use a mixture of PCA models? This idea is analogous to Gaussian Mixture Models (GMM), detailed in the next section, where the data distribution is approximated using a mixture of Gaussians rather than a single Gaussian. Similarly, the probabilistic framework allows that a mixture of PPCA models can be easily implemented to better capture the structure of the data, which is an advantage over the non-probabilistic approach.

- **Data Generation and Likelihood Computation:** If your task involves data generation or the computation of densities (e.g., finding the likelihood of a point in your dataset), the probabilistic versions (PPCA/FA) are the way to go, as they explicitly model the data distribution.

For more details on these advantages, as well as the estimation procedures for FA, PPCA, and PCA using the Expectation-Maximization (EM) algorithm, refer to the excellent reference paper by Bishop and Tipping (2001) on PPCA.

In practice, choosing the appropriate method can be done using cross-validation or domain knowledge. Also, you might:

- Use explained variance to determine the dimension of the latent space.
- You might decide between PPCA and FA based on whether you believe the idiosyncratic variance

(residual noise) of each variable should be equal (homoscedastic), requiring a single σ^2 to estimate, or different (heteroscedastic), requiring D variances to estimate $\sigma_1^2, \dots, \sigma_D^2$.

- For initial data exploration or modeling insight, you might use PCA, which is computationally more efficient to estimate than its probabilistic counterpart. Depending on your needs, PCA might offer a good trade-off between performance and computational complexity, especially when the amount of data to process is extremely large.

By understanding or making assumptions about the noise structure in our data we can make a better informed decision about our modeling choices.

6.3.3 Gaussians Mixture Models

In the Gaussian Mixture Model (GMM), the latent variable z is discrete, taking values in the set $\{1, \dots, K\}$, while the observed value \mathbf{x} is continuous, taking values in \mathbb{R}^D . The latent variable follows a Categorical distribution, denoted as $p(z) = \text{Categorical}(1, \dots, K; \boldsymbol{\pi})$, where $\boldsymbol{\pi} = [\pi_1, \dots, \pi_K]$, and, for example, the probability that the variable z takes the value k is given by $p(z = k) = \pi_k$. The conditional distribution of \mathbf{x} given z are Gaussian, defined by $p(\mathbf{x}|z = k) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$.

The mixture of Gaussian becomes apparent when computing the marginal over the observed variable by marginalizing the joint probability between the latent and observed variables over the latent variable, as follows:

$$\begin{aligned}
p(\mathbf{x}) &= \sum_{k=1}^K p(z=k, \mathbf{x}) \\
&= \sum_{k=1}^K p(z=k) p(\mathbf{x}|z=k) \\
&= \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)
\end{aligned} \tag{6.3}$$

The result is simply a weighted sum of Gaussians (hence the adjective “mixture”). We can create different mixture models by altering the probability mass over the latent variables.

The posterior probability distribution of the latent variables given the observations quantifies the probability that the data point \mathbf{x} was generated by the k -th mixture component. It is given by the conditional distribution:

$$p(z=k|\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \tag{6.4}$$

For an illustrative example of GMM applied to clustering, see [Figure 6.4](#). For more details on GMM see the excellent references Bishop (2006) and Murphy (2022).

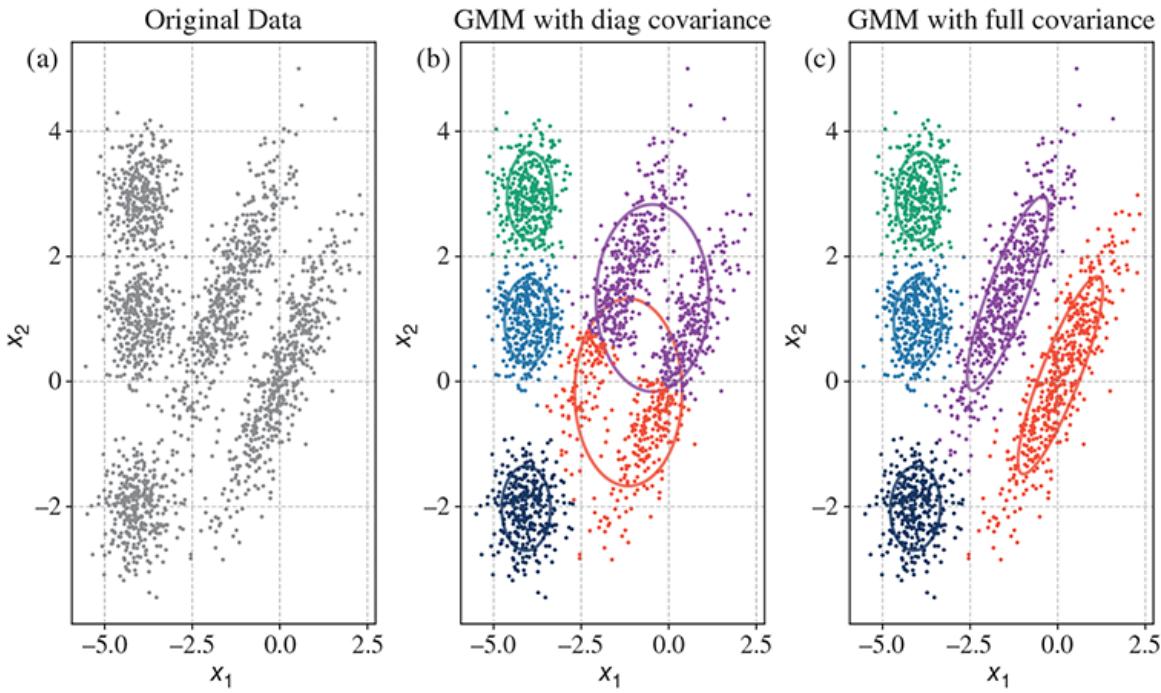


Figure 6.4 Illustration of clustering using Gaussian Mixture Models (GMMs). (a) Generated 2D data samples. (b) A possible clustering of the same data with $K = 5$ using GMMs with a diagonal covariance matrix. (c) A possible clustering with $K = 5$ using GMMs with a full covariance matrix. Cluster assignments (indicated by colors) are determined by selecting the cluster with the highest posterior probability for each point. Illustration based on [Figure 3.12](#) in Murphy (2022), with additional details.

6.3.3.1 Gaussian Mixture Model (GMM) for Market Regime Detection.

Let's explore how Gaussian Mixture Models (GMM) can be applied to a well-known problem: detecting market regimes. As we know, markets change over time, and traders and investors often use domain knowledge to label

certain periods. Common labels we hear in the press, on the news, or in interviews include terms like “bull market,” “bear market,” “trending market,” or “mean-reverting market,” and so on. In this case, however, we are going to take a data-driven approach to estimate market regimes.

Modeling market regimes is crucial for traders and investors, as for example, it might help them deploy the most appropriate strategy from a pool of potential strategies for the current market conditions or incorporate regime information into portfolio construction models for risk management. As we will see, GMM serves as a powerful building block for developing more sophisticated models that can be tailored to your specific market insights.

6.3.3.2 Example: Low and High Volatility Regimes.

In this hypothetical scenario, based on domain knowledge, let's say we are interested in identifying two distinct clusters that may correspond to two different market regimes: one with low volatility and the other with high volatility. These regimes might be identified entirely based on domain knowledge or through exploratory data analysis. For instance, it's unlikely that the returns of an asset like the SPY could be well explained over long periods using a single Gaussian distribution with constant mean and constant variance. Consider recent periods in history, such as the time before, during, and after the COVID-19 pandemic. During the pandemic, volatility was exceptionally high, while the months before and after exhibited much lower volatility. For simplicity, we might build a model to capture this high-volatility period separately from the low-volatility periods, which we could then visualize as “clusters” in the SPY returns time series. From an application point of view, identifying volatility

regimes can help traders make decisions such as adjusting portfolio allocations, modifying leverage levels, or implementing volatility-specific trading strategies.

In the following example, we use the SPY ETF return data for market regime detection. In this setup, the latent variable z represents the market regime and can take two possible values: “low volatility” or “high volatility.”

Conditioned on the market regime (or state of z), we assume that returns follow a Gaussian distribution with regime-specific means and variances.

In this case, the data generation process can be described as follows. Since we have only two states for the latent variable, 0 or 1, at time t we draw the value of the state from a Bernoulli distribution with parameter π , where $P(z = 1) = \pi$:

$$z_t \sim \text{Bernoulli}(\pi)$$

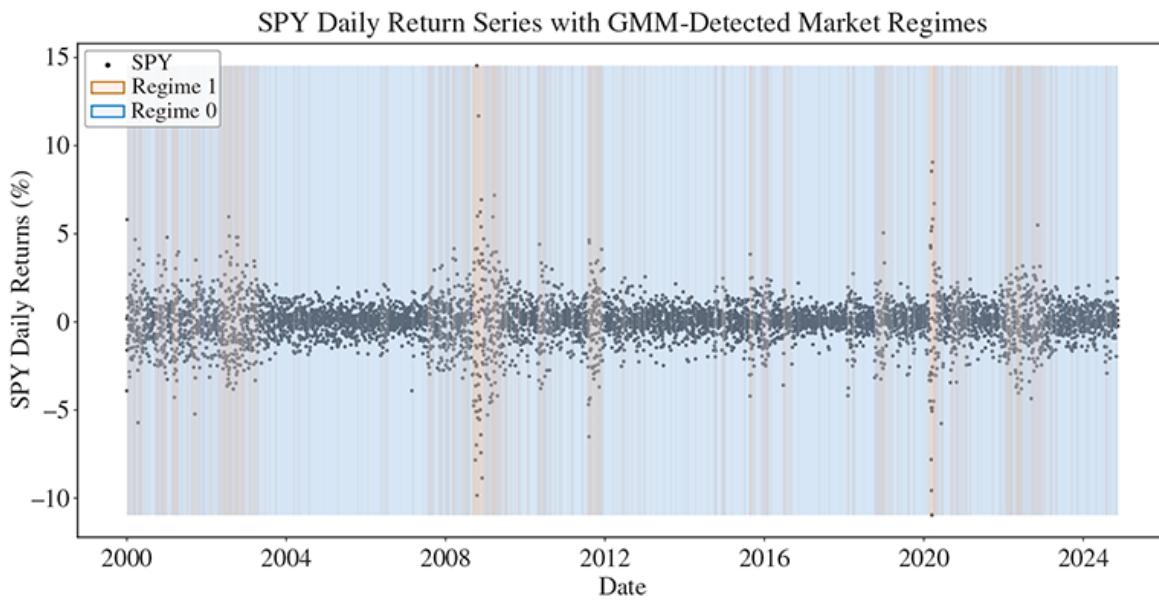
Once the value of the latent variable is determined, at time t we generate data from the corresponding Gaussian distribution, with regime-specific means and variances, as follows:

$$x_t \sim (1 - z_t)\mathcal{N}(\mu_0, \sigma_0) + z_t\mathcal{N}(\mu_1, \sigma_1)$$

Here, the state $z = 0$ corresponds to the first cluster, or low-volatility regime, while $z = 1$ represents the second cluster, or high-volatility regime. In the accompanying notebook [*GMM-Market-Regime-Detection*], we will demonstrate how to estimate the parameters of this model, namely $\boldsymbol{\theta} = \{\pi, \mu_0, \mu_1, \sigma_0, \sigma_1\}$, using data from the SPY ETF.

[Figure 6.5](#) is a time-series plot of SPY returns from January 2000 to March 2024, along with the most probable market regimes estimated from the posterior probability distribution (see [Equation 6.4](#)) for this period. In this plot,

individual points represent SPY returns, while vertical frames indicate the most probable regime at each time point. Light orange corresponds to regime 0 (low volatility), and blue corresponds to regime 1 (high volatility) (see [Figure 6.5](#)). As shown in [Figure 6.5](#), this simple model identifies well-known high-volatility periods over the timeline, such as the 2008 financial crisis and the pandemic period in 2020. It is worth noting that in this example, we are addressing the problem of market regime detection or identification, not market regime forecasting or prediction.



[Figure 6.5 Gaussian Mixture Model for market regime detection.](#)

Two Sigma, one of the largest hedge funds in the world, ranked fourth in Forbes Advisor's "Top 10 US Hedge Funds of December 2024" (Baldridge, 2024) with \$67.47 billion in assets under management (AUM) at the time of writing this chapter, shared a similar approach in their blog post, "A Machine Learning Approach to Regime Modeling," see Botte and Bao (2021). They used GMM to detect market regimes and identified four clusters they linked to different market conditions: Crisis, Steady State, Inflation, and Walking on Ice. They backed these labels with historical

analysis and applied the method to custom-built factors developed in-house.

Even though their setup uses more clusters and proprietary data, the underlying method is very similar. GMM is a strong starting point for building more powerful models, which you can tailor with your unique insights into the market.

There are many ways to improve our modeling approach for regime detection. For example, one limitation of the basic GMM setup is that it assumes the latent variables are independent over time. This assumption ignores the well-known stylized fact of volatility clustering, where periods of high volatility tend to be followed by more high-volatility periods, and the same is true for low-volatility periods.

To capture this, you can add dynamics to the latent variables. This essentially turns GMM into a Hidden Markov Model (HMM), which has been widely used for market regime detection (see Chan [2017] Kinlaw et al., [2021]). Unlike GMM, HMM adds “memory,” allowing it to better reflect how market conditions evolve over time.

As a fun historical note, Leonard Baum (https://en.wikipedia.org/wiki/Leonard_E._Baum), one of the first employees of Monometrics (the precursor to Renaissance Technologies, considered the most successful quantitative hedge fund ever), co-developed the Baum-Welch algorithm. This algorithm, a special case of Expectation-Maximization (EM), is used to estimate the parameters of an HMM.

According to Gregory Zuckerman’s book (Zuckerman, 2019), *The Man Who Solved the Market*, James Simons, founder of Renaissance Technologies, said about his early days at Monometrics: “Once I got Lenny (Leonard) involved, I could see the possibilities of building models.”

Now, in the rest of this chapter, we will focus on cases where the parameters of the conditional distribution of observations given the latent variables are functions of the latent variables and are modeled using Deep Neural Networks.

6.3.4 Deep Latent Variable Models

In this book, we will focus on specific representations for the latent variable model we have discussed so far, namely:

$$\begin{aligned}\mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x}|\mathbf{z} &\sim p(\mathbf{x}|\mathbf{z})\end{aligned}$$

Here, $p(\mathbf{z})$ represents the prior over the latent variable \mathbf{z} . Unlike the earlier examples, the parameters of the conditional probability of \mathbf{x} given \mathbf{z} are determined by Deep Neural Network function of \mathbf{z} . Typically, as before, the relationship is represented as

$p(\mathbf{x}|\mathbf{z}) = \text{ExponentialFamily}(\mathbf{x}; \eta)$, but now η is defined as $\eta = \text{DNN}_{\theta}(\mathbf{z})$.

Note that $p(\mathbf{x}|\mathbf{z})$ is also known as the **decoder**, a term that you will find all over the deep learning and LLM literature. Such models are referred to as Deep Latent Variable Models (DLVM). When the prior over the latent variable is Gaussian, the model is known as a Deep Latent Gaussian Model (DLGM); see Murphy (2023).

Contrary to the examples previously discussed, the posterior computation $p(\mathbf{z}|\mathbf{x})$ is intractable. Likewise, the marginal over \mathbf{x} is also intractable:

$$\begin{aligned}
p(\mathbf{x}) &= \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\
&= \int p(\mathbf{z}) p(\mathbf{x}|\mathbf{z}) d\mathbf{z} \\
&= \int p(\mathbf{z}) \text{ExponentialFamily}(\mathbf{x}; \text{DNN}_{\theta}(\mathbf{z})) d\mathbf{z}
\end{aligned}$$

We will need to approximate this quantity to compute the density of \mathbf{x} and, in turn, use it to estimate the parameters of the model. In the next section, we will explore methods for approximating this integral to effectively train Deep Latent Variable Models.

6.4 Learning

6.4.1 Training Objective

As we discussed in the previous chapter on autoregressive (AR) models, once we define a family of probability distributions parametrized by θ , denoted by $\{p_{\theta}\}_{\theta}$, our goal is to establish an objective function that will allow us to select the best model from this family that most accurately approximates the data distribution, p_{data} .

Similar to AR models, we will use the log-likelihood to define our objective function:

$$\begin{aligned}
\mathcal{L} &= \frac{1}{N} \sum_n \log p_{\theta}(\mathbf{x}_n) \\
&= \frac{1}{N} \sum_n \log \int p(\mathbf{x}_n | \mathbf{z}_n) p(\mathbf{z}_n) d\mathbf{z}_n
\end{aligned}$$

however, as we have observed, this calculation is intractable for most interesting cases.

6.4.2 The Variational Inference Approximation

Given that the exact likelihood is intractable, we will derive an approximation for it and pick the best model by optimizing this quantity. The following derivations were inspired by the exposition style presented in Levine (2021). Let's focus on a *single* observation \mathbf{x}

$$\begin{aligned}\log p(\mathbf{x}) &= \log \int p(\mathbf{x}, \mathbf{z}) d\mathbf{z} \\ &= \log \int \frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} q(\mathbf{z}) d\mathbf{z} \\ &= \log \mathbb{E}_{\mathbf{z} \sim q} \left[\frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right]\end{aligned}\tag{6.5}$$

where $\mathbb{E}_{\mathbf{z} \sim q}$ indicates expectation value over the distribution $q(\mathbf{z})$. In this derivation, we introduced an arbitrary distribution $q(\mathbf{z})$ by applying an identity—multiplying and dividing the integrand by this quantity. This technique allowed us to rewrite the integral as an expectation with respect to arbitrary distribution $q(\mathbf{z})$. This technique is known as importance sampling, and it provides a framework for approximating expectations with respect to distributions that are difficult to sample from but easy to evaluate. It does so by using a distribution that is easier to sample from, such as $q(\mathbf{z})$, which is a design choice (see Bishop [2006]). In the importance sampling literature, $q(\mathbf{z})$ is referred to as the proposal distribution.

Applying Jensen's inequality to [Equation 6.5](#), $\log \mathbb{E}[x] \geq \mathbb{E}[\log x]$, we can push the log inside the expectation, obtaining:

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \mathbb{E}_{\mathbf{z} \sim q} \left[\frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right] \\
&\geq \mathbb{E}_{\mathbf{z} \sim q} \log \left[\frac{p(\mathbf{x}, \mathbf{z})}{q(\mathbf{z})} \right] \\
&\geq \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z})] \\
&\geq \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z})]
\end{aligned} \tag{6.6}$$

Here, the last expression in [Equation 6.6](#) is known as the Evidence Lower Bound (ELBO, pronounced as “elbow”), defined as follows:

$$\mathcal{L}(q, p) = \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z}) - \log q(\mathbf{z})] \tag{6.7}$$

The ELBO can be written or interpreted in two ways (Levine, 2021). The first one uses the entropy formulation:

$$\begin{aligned}
\mathcal{L}(q, p) &= \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z})] - \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) \tag{6.8} \\
&= \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z})] + \mathbb{H}[q]
\end{aligned}$$

here, $\mathbb{H}[q]$ represents the Entropy of the probability distribution q , defined $\mathbb{H}[q] = \mathbb{E}_{\mathbf{z} \sim q} [-\log q(\mathbf{z})]$.

The second commonly used interpretation involves the Kullback-Leibler Divergence:

$$\begin{aligned}
\mathcal{L}(q, p) &= \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z}) + \log p(\mathbf{z})] - \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) \tag{6.9} \\
&= \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z})] - \mathbb{E}_{\mathbf{z} \sim q} \left[\log \frac{q(\mathbf{z})}{p(\mathbf{z})} \right] \\
&= \mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{x}|\mathbf{z})] - \text{KL}(q(\mathbf{z}) \| p(\mathbf{z}))
\end{aligned}$$

This formulation highlights the common interpretation given to this objective function in the variational inference literature: the first term represents the reconstruction

error (log likelihood), and the second term serves as a regularization term. The advantage of this approach is that the KL divergence between $q(\mathbf{z})$ and $p(\mathbf{z})$ often has a closed-form solution, especially when both distributions are Gaussian—a common design choice that also simplifies the gradient computation.

6.4.2.1 How to Choose the Proposal Distribution

One critical decision in this formulation is to pick the proposal distribution, $q(\mathbf{z})$. We could say that it makes sense to align the proposal distribution close to the posterior distribution of the latent variable \mathbf{z} given the observed data \mathbf{x} , represented as $p(\mathbf{z}|\mathbf{x})$. Therefore, we can opt to minimize the KL divergence between both distributions to make $q(\mathbf{z})$ close to $p(\mathbf{z}|\mathbf{x})$, which involves:

$$\begin{aligned}
 \text{KL}(q(\mathbf{z}) \| p(\mathbf{z}|\mathbf{x})) &= \mathbb{E}_{\mathbf{z} \sim q} \left[\log \frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} \right] \\
 &= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z}|\mathbf{x}) \\
 &= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}, \mathbf{z}) + \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}) \\
 &= \mathbb{E}_{\mathbf{z} \sim q} \log q(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{z}) - \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}|\mathbf{z}) + \mathbb{E}_{\mathbf{z} \sim q} \log p(\mathbf{x}) \\
 &= -\mathbb{E}_{\mathbf{z} \sim q} [\log p(\mathbf{z}) + \log p(\mathbf{x}|\mathbf{z}) - \log q(\mathbf{z})] + \log p(\mathbf{x}) \\
 &\quad - \mathcal{L}(q, p) + \log p(\mathbf{x})
 \end{aligned} \tag{6.10}$$

here, minimizing $\text{KL}(q(\mathbf{z}) \| p(\mathbf{x}|\mathbf{z}))$ with respect to q is equivalent to maximizing $\mathcal{L}(q, p)$ with respect to q , since $\log p(\mathbf{x})$ is independent of \mathbf{z} :

$$\operatorname{argmin}_q \text{KL}(q(\mathbf{z}) \| p(\mathbf{z}|\mathbf{x})) = \operatorname{argmax}_q \mathcal{L}(q, p) \quad (6.11)$$

Hamlet remembers the first time he encountered this derivation and how amazed he was by its elegant result. It enables us to design a proposal distribution using tractable quantities, even when the quantities involved in the derivation, such as the posterior distribution, are intractable. In most practical cases, where deep neural networks are used to parameterize conditional distributions, computing the posterior $p(\mathbf{z}|\mathbf{x})$ directly is infeasible. If we had direct access to this posterior, we could use it instead of relying on $q(\mathbf{z})$.

Remarkably, the derivation shows that we do not actually need access to the posterior $p(\mathbf{z}|\mathbf{x})$ to find a proposal distribution $q(\mathbf{z})$ that is close to it, that minimizes the KL divergence between both distributions. By developing the optimization problem, we discover that making $q(\mathbf{z})$ as close as possible to the posterior is equivalent to maximizing [Equation 6.7](#), the ELBO with respect to q . This is a powerful result because the ELBO consists of quantities that are tractable and can be computed efficiently.

What surprised Hamlet most is the realization that by optimizing the same objective—the ELBO—with respect to p , we can simultaneously find the best parameters for approximating the data distribution. In other words, finding the optimal q that best approximates the posterior and the optimal p that best models the data involves alternating between two steps:

1. Maximizing the ELBO with respect to q to make $q(\mathbf{z})$ as close as possible to the posterior $p(\mathbf{z}|\mathbf{x})$ in KL divergence sense.

2. Maximizing the ELBO with respect to p to increase the likelihood of the observed data \mathbf{x} .

This alternating optimization approach unifies the goals of posterior approximation and data likelihood maximization under the same objective function.

Recall that we have constructed all these equations for a single, fixed, observation \mathbf{x} . One of the challenges with this formulation is that for each \mathbf{x}_n in our dataset \mathcal{D} , we need to solve an inference problem as specified by [Equation 6.11](#) to determine q for the specific \mathbf{x}_n , denoted as q_n . For example, if we represent $q_n(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_n, \boldsymbol{\sigma}_n)$, this will involve finding a unique $\boldsymbol{\mu}_n$ and $\boldsymbol{\sigma}_n$ for every data point. Consequently, the number of parameters to be estimated are:

$$|\boldsymbol{\theta}| + N(|\boldsymbol{\mu}| + |\boldsymbol{\sigma}|)$$

here, the notation $|\cdot|$ represent the number of parameters. As indicated by the previous formula, the number of parameters to estimate is linear with respect to the size of the dataset N , which in current practical applications can be huge.

6.4.2.2 Amortized Inference.

The main idea behind amortized inference is that if we repeatedly face the same inference problem, could a parameterized function solve it for us? This leads us to consider using a family of neural networks for this task.

Instead of determining a distinct q by solving the optimization problem in [Equation 6.11](#) for every individual \mathbf{x} , we might think of finding a single q_ϕ where the parameters are determined by a Deep Neural Network that

takes \mathbf{x} as input, and that minimizes the average KL divergence across the entire dataset:

$$\min \sum_n \text{KL}(q_\phi(\mathbf{z}_n | \mathbf{x}_n) \| p_\theta(\mathbf{z}_n | \mathbf{x}_n)) = \max \sum_n \mathcal{L}(q_\phi(\mathbf{z}_n | \mathbf{x}_n), p_\theta(\mathbf{x}_n | \mathbf{z}_n)) \quad (6.12)$$

This formulation enables us to use another Deep Neural Network to define the parameters of q_ϕ and solve the problem in an amortized manner. This network takes an input \mathbf{x} and outputs the parameters of a probability distribution over the latent variable \mathbf{z} is commonly referred to as the **encoder** in the literature.

6.4.3 Optimization

As previously seen, to optimize for ϕ and θ , we need to maximize the same training objective, the ELBO, with respect to both θ and ϕ . Now, considering the computation of:

$$\nabla_\phi \mathbb{E}_{\mathbf{z} \sim p_\phi} [p_\theta(\mathbf{x} | \mathbf{z})] \quad (6.13)$$

we face the challenge of computing gradients with respect to ϕ , which involves taking the derivative with respect to the parameters of the distribution over which the expected value is taken. There are two common methods to address this problem: the famous REINFORCE algorithm (Williams, 1992) and the reparametrization trick (Kingma, 2013).

6.4.3.1 The Likelihood Gradient, REINFORCE.

The REINFORCE algorithm can be derived as follows:

(6.14)

$$\begin{aligned}
\nabla_{\phi} \mathbb{E}_{\mathbf{z} \sim q_{\phi}} [p_{\theta}(\mathbf{x}|\mathbf{z})] &= \nabla_{\phi} \int q_{\phi}(\mathbf{z}|\mathbf{x}) p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z} \\
&= \int \nabla_{\phi} q_{\phi}(\mathbf{z}|\mathbf{x}) p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z} \\
&= \int q_{\phi}(\mathbf{z}|\mathbf{x}) \nabla_{\phi} \log q_{\phi}(\mathbf{z}|\mathbf{x}) p_{\theta}(\mathbf{x}|\mathbf{z}) d\mathbf{z} \\
&= \mathbb{E}_{\mathbf{z} \sim p_{\phi}} [\nabla_{\phi} \log q_{\phi}(\mathbf{z}|\mathbf{x}) p_{\theta}(\mathbf{x}|\mathbf{z})]
\end{aligned}$$

This method computes the gradient of an expectation by exploiting the identity $\nabla_x \log f(x) = \frac{\nabla_x f(x)}{f(x)}$. So, the

gradient of an expectation becomes the expectation of a gradient. REINFORCE uses a Monte Carlo approach to estimate the gradient of the expectation in [Equation 6.14](#). One advantage of this method is that it works with latent variables whose value spaces can be either discrete or continuous. However, one of the primary disadvantages with this approach is that it tends to suffer from high variance. For more details about the algorithm, see Williams (1992).

6.4.3.2 Reparameterization Trick.

An alternative approach is the reparameterization trick (Kingma, 2013), which can be utilized for the computation of [Equation 6.13](#).

For expectations involving sampling \mathbf{z} from $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_{\phi}(x), \boldsymbol{\Sigma}_{\phi}(x))$, we can reparametrize \mathbf{z} as:

$$\mathbf{z} = \boldsymbol{\mu}_{\phi}(x) + \boldsymbol{\Sigma}_{\phi}(x)^{1/2} \boldsymbol{\epsilon} \quad (6.15)$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$. With the reparameterization trick, all randomness is absorbed by $\boldsymbol{\epsilon}$, and for computing the

gradient with respect to ϕ , we just need to compute the gradient of the deterministic functions μ_ϕ and Σ_ϕ .

$$\nabla_\phi \mathbb{E}_{\mathbf{z} \sim q_\phi} [p_\theta(\mathbf{x}|\mathbf{z})] = \nabla_\phi \mathbb{E}_{\epsilon \sim \mathcal{N}(0, \mathbf{I})} \left[p_\theta \left(\mathbf{x} \middle| \boldsymbol{\mu}_\phi(x) + \boldsymbol{\Sigma}_\phi(x)^{1/2} \boldsymbol{\epsilon} \right) \right]$$

This gradient can be computed efficiently using automatic differentiation tools such as PyTorch or TensorFlow.

6.4.4 Mind the Gap!

Another way to write [Equation 6.10](#) is:

$$\log p(\mathbf{x}) = L(q, p) + \text{KL}(q(\mathbf{z}) \| p(\mathbf{x}|\mathbf{z})) \quad (6.16)$$

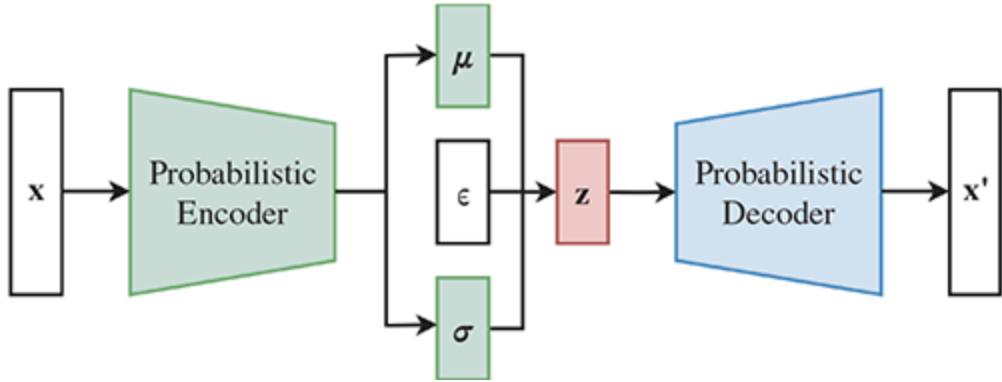
In this formulation, the term $\text{KL}(q(\mathbf{z}) \| p(\mathbf{x}|\mathbf{z}))$ is known as the gap or error that arises when approximating the log likelihood by the ELBO. If we truly match $q(\mathbf{z})$ to $p(\mathbf{x}|\mathbf{z})$, the KL will be zero and the log likelihood is equal to the ELBO. Consequently, the error in our approximation is quantified by the KL divergence between our approximation q_ϕ to $p(\mathbf{z}|\mathbf{x})$.

6.5 Variational Autoencoder (VAE)

The Variational Autoencoder (VAE) is a latent variable model where the marginal distribution over the latent variable is Gaussian, and both the encoder and decoder (to be discussed shortly) are modeled as conditional Gaussians, as described by Kingma (2013). The observations can be images, multivariate time series, etc.

The encoder takes as input \mathbf{x} and outputs the parameters of $q_\phi(\mathbf{z}|\mathbf{x})$, specifically the mean vector and covariance matrix. Similarly, the decoder takes \mathbf{z} as input and outputs the

mean vector and covariance matrix for $p_{\theta}(\mathbf{x}|\mathbf{z})$. The classical VAE is typically trained using the reparametrization trick. A schematic representation of these operations is illustrated in [Figure 6.6](#).



[Figure 6.6](#) VAE.

Source: EugenioTL / https://en.m.wikipedia.org/wiki/File:Reparameterized_Variational_Autoencoder.png, last accessed on 13 January 2025 / CC BY 3.0.

More formally, the classical VAE (Kingma, 2013) is defined by the following representations:

- The prior over \mathbf{z} , $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$
- The encoder, $q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_{\phi}(\mathbf{x}), \boldsymbol{\Sigma}_{\phi}(\mathbf{x}))$
- The decoder $p_{\theta}(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\boldsymbol{\mu}_{\theta}(\mathbf{z}), \boldsymbol{\Sigma}_{\theta}(\mathbf{z}))$

VAEs are typically trained using the reparametrization trick. During training, the ELBO is used as training objective:

$$\mathcal{L}(q, p) = \mathbb{E}_{\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})} [\log p_{\theta}(\mathbf{x}|\mathbf{z})] - \text{KL}(q_{\phi}(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}))$$

with $\mathbf{z} = \boldsymbol{\mu}_{\phi}(\mathbf{x}) + \boldsymbol{\Sigma}_{\phi}(\mathbf{x})^{1/2} \boldsymbol{\epsilon}$. To optimize for ϕ and θ , gradients $\nabla_{\theta} L(q, p)$ and $\nabla_{\phi} L(q, p)$ are computed, typically using Stochastic Gradients. Parameters ϕ and θ can be then updated using gradient ascent.

After training the VAE, new points can be generated by sampling a value from the prior and feeding this value into the decoder to generate a new sample:

$$\begin{aligned}\mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x}|\mathbf{z} &\sim p_{\theta}(\mathbf{x}|\mathbf{z})\end{aligned}$$

In [Figure 6.7](#), we see visualizations from the paper Kingma (2013) of the learned latent space and the corresponding data space, generated by a VAE after training on the Frey Faces dataset.



Figure 6.7 Illustration of the learned data manifold for generative models with a two-dimensional independent Gaussian latent space. Varying one coordinate generates smiling faces, while varying the other alters head poses, both independently. The model was trained on the Frey Faces dataset.

Source: Kingma (2013).

A remarkable aspect is that since the prior of the latent space are independent Gaussian variables (identity covariance), using a two-dimensional latent space illustrates disentanglement in the data space. Varying one coordinate can generate smiling faces, while varying the

other can alter head poses, all independently, as shown in [Figure 6.7](#).

The VAE can also serve as a representation learning tool for downstream machine learning tasks. For example, if \mathbf{x} represents an image, the compressed representation \mathbf{z} could be used as input for an image classifier. In fact, this representation learning approach is utilized in Stable Diffusion, one of the most powerful techniques for image generation currently available.

In the accompanying notebook for this chapter ([\[VAE Notebook\]](#)), we implement and test the traditional VAE model introduced by Kingma (2013). While this example uses image data, it serves as an important step for building a solid understanding of VAEs, which we will extend to capture time-series dynamics.

6.6 VAEs for Sequential Data and Time Series

As we saw in the previous section based on the work of Kingma (2013), and illustrated in [Figure 6.7](#), VAEs are really good at modeling the correlations within an image \mathbf{x}_n in a dataset, where \mathbf{x}_n is a D -dimensional vector of pixel intensities. If we want to apply standard VAEs to time-series data, we might simply reinterpret the vector \mathbf{x}_n as a multivariate time series with D series at time n and use VAEs to sample cross-sectional time series, for example.

However, in the standard latent variable model framework, the data generation process assumes that we sample \mathbf{z}_n from $p(\mathbf{z})$ and then generate \mathbf{x}_n from the conditional distribution $p(\mathbf{x}|\mathbf{z}_n)$. By design, this structure assumes independence between consecutive observations. This independence assumption can significantly limit the

performance of VAEs in scenarios where temporal dependencies between observations are significant.

There are several interesting ways to extend the standard VAE to capture temporal dependencies between observations. This is particularly important for modeling time series at certain time scales, where past prices/returns, market conditions, and other factors significantly influence future prices/returns. These extensions can take inspiration from traditional time-series models, such as incorporating autoregressive structures in the observations or latent variables, similar to classical AutoRegressive models or Dynamic Linear Models (DLMs), see Prado and West (2010). Next, we mention a few alternatives for this.

6.6.1 Extending VAEs for Time Series

6.6.1.1 Sequential Encoders and Decoders.

One straightforward extension is to replace the standard VAE encoder and decoder used in Kingma (2013) with sequential models, such as RNNs, or Transformers. One example of this approach is the Variational Recurrent Autoencoder (VRAE) by Fabius and Van Amersfoort (2014). Following is a brief description of the main idea behind VRAE:

- VRAE uses the final hidden state of the encoder RNN as a representation or summary of the input sequence. This hidden state is then used to determine the parameters of the probability distribution of the latent variable \mathbf{z} . In other words, the distribution over \mathbf{z} is determined from the last state of the RNN.

- The latent variable \mathbf{z} is then sampled using the reparameterization trick.
- Finally, the sampled latent variable \mathbf{z} is passed into the decoder RNN, which reconstructs the input sequence.

6.6.1.2 Superposition of Time-series Components.

Another approach is to model time series as a superposition of components—an idea commonly used in dynamic linear models (DLMs), see Prado and West (2010). In this framework, the observed time series might be composed as a combinations of the following:

- Local level or intercept (e.g., a random walk)
- Long-term trend components
- Seasonal patterns (e.g., weekly, monthly, or yearly periodicities)
- Short-term autocorrelated processes

Here, each time-series component can have its own state vector (latent variables), which is then projected linearly into the data space.

This approach provides both the interpretability and the flexibility to incorporate domain expertise, address data limitations, and adapt to the specific needs of the problem at hand. By using individual time-series components, modelers can design solutions that are tailored to their specific needs. In the next section, we will describe a solution based on this principle.

If you want to learn more about DLM models, check out the introduction to *Bayesian Structural Time-series Models* (Medina Ruiz, 2019), which explores time-series modeling for predicting Criteo's internet traffic load in the context of

server infrastructure capacity planning. This example highlights how injecting domain knowledge into the modeling process can address data limitations and improve the overall performance.

6.6.1.3 TimeVAE: A Flexible VAE for Time-series Generation.

A more recent example of extending VAEs to time series is TimeVAE, introduced by Desai et al. (2021). TimeVAE adapts VAEs specifically for multivariate time-series generation, combining traditional deep learning layers with time-series-specific components.

By using specialized decoders, TimeVAE allows the inclusion of temporal structures—such as trends and seasonal components—directly into the modeling process. This approach combines the flexibility and interpretability of traditional methods like Dynamic Linear Models (DLMs), but with the advantage of capturing nonlinear relationships in high-dimensional data. TimeVAE makes use of both traditional learning layers (e.g., dense and convolutional layers) and custom time-series-specific layers to capture components like levels, multi-polynomial trends, and seasonalities.

The TimeVAE architecture is divided into two main parts: **Base TimeVAE**, which does not incorporate domain knowledge for modeling the time series, and **Interpretable TimeVAE**, which allows domain knowledge to be injected and is tailored to model specific time-series components. Following is a description of the architecture.

6.6.1.3.1 Architecture of TimeVAE

1. Base TimeVAE encoder and decoder

- The encoder processes the input time series through a series of convolutional layers with ReLU activation.
- The output of the encoder is flattened and passed through a fully connected (dense) linear layer to determine the parameters of the distribution over the latent variable \mathbf{z} , which is modeled as a multivariate Gaussian.
- A realization of the latent variable \mathbf{z} is sampled using the reparametrization trick, which is then feed into the decoder.
- The decoder tries to invert the encoding process. It consists of a fully connected linear layer, followed by reshaping and a series of transposed convolutional layers with ReLU activation. Finally, the data passes through a time-distributed fully connected layer to produce the reconstructed time series.

A schematic representation of Base TimeVAE can be found in [Figure 6.8](#).

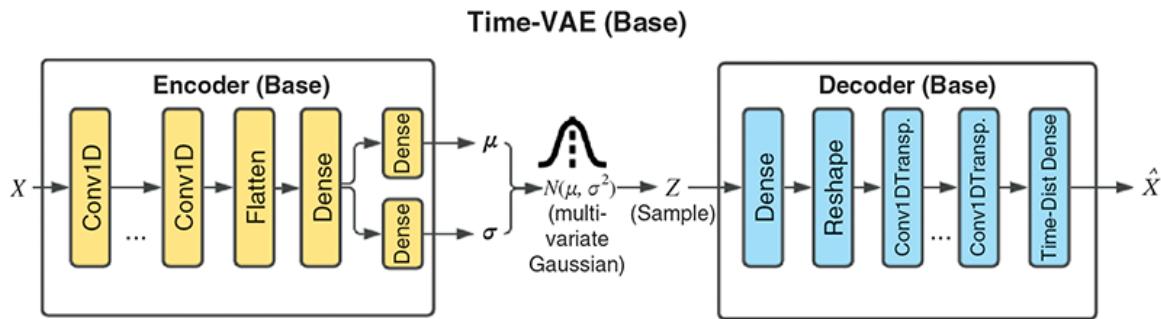


Figure 6.8 Illustration of the encoder-decoder architecture of Base TimeVAE. The input X is processed through a series of operations, including convolutional layers, reshaping, and a dense layer, to determine the parameters of the latent variable distribution Z . The decoder takes a realization of Z and tries to reconstruct the input by inverting the encoding process. Figure 1 from Desai et al. (2021).

2. Interpretable TimeVAE: Time-series-specific components

As mentioned, TimeVAE introduces specialized decoders to model individual components of a time series, such as trends and seasonalities.

Each specialized decoder takes the latent variable z as input (which can be produced by the Base TimeVAE encoder). The outputs of these decoders are combined in parallel by addition to produce the final reconstruction of the time series. A schematic representation of Interpretable TimeVAE can be found in [Figure 6.9](#).

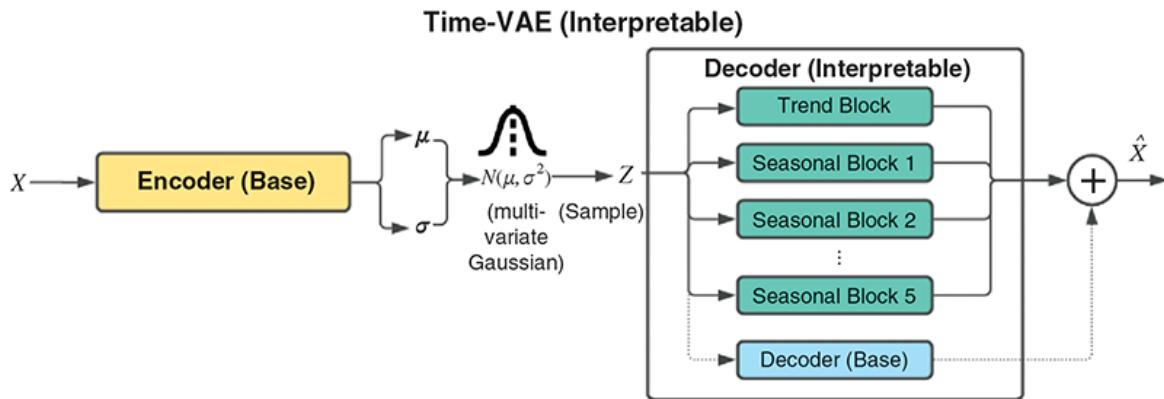


Figure 6.9 Illustration of the main components in Interpretable TimeVAE. Specialized decoders, such as the Trend Block, Seasonal Block 1, and Seasonal Block 2, take the latent variable Z as input. The outputs of these decoders are combined in parallel through addition to produce the final reconstruction of the time series. Seasonal Block 1 and Seasonal Block 2 can represent different seasonal components, capturing multiple seasonalities present in the time series. Figure 2 from Desai et al. (2021).

TimeVAE provides the following advantages:

- Flexibility: You can easily model multiple temporal components, such as polynomial trends of different degrees, and different seasonalities (e.g., weekly, monthly, yearly), making it highly adaptable to capture different time-series characteristics.
- Interpretability: By explicitly modeling components like trends and seasonality, TimeVAE makes it easier to understand the underlying structure of the time series.
- Nonlinear Modeling: unlike traditional models like DLM, TimeVAE can capture complex nonlinear relationships in high dimensional data.

In the accompanying notebook for this chapter ([*TimeVAE Notebook*]), we implement and test TimeVAE on financial data. We believe that the best way to learn is by hands-on practice, and this notebook gives you the opportunity to explore and experiment with the specific decoder architectures used in the paper for modeling trends and seasonalities. You can also extend these architectures to fit your specific needs.

If you are not yet familiar with the standard VAE implementation, we recommend starting with the VAE notebook, where the encoder and decoder architectures are easier to follow. This will provide a solid foundation before diving into TimeVAE.

6.7 Conclusion

In this chapter, we explored how VAEs provide a powerful framework for modeling complex, multimodal, and high-dimensional data.

We saw how VAEs are highly efficient at sampling and flexible for modeling high-dimensional data using a smaller latent space representation. This latent space provides meaningful data representations that can be used for applications such as compression, data generation, and other downstream tasks. However, one trade-off for this flexibility is that the likelihood computation becomes intractable for most practical cases involving deep models. To address this, we rely on likelihood approximations, which enable parameter estimation and tasks like model selection.

We also discussed how VAEs can be extended to handle sequential data, particularly for time-series dynamics. For example, we introduced approaches like the VRAE and

TimeVAE, which combine the VAE framework with sequential architectures like RNNs and time-series-specific components. These models offer a practical starting point for building solutions tailored to your own applications, whether for forecasting financial data, generating financial data, or other sequential problems or data. The latent representation learned by VAEs is highly versatile and can be used to train downstream models to solve a wide variety of applications. While we have seen its use in time-series generation, it can also be used for tasks such as classification, anomaly detection, and forecasting.

In the next chapters, we will continue expanding our toolbox of deep generative models for high-dimensional data—models that were originally successful in NLP and computer vision. More importantly, for our use cases in Trading and Asset Management, we will explore techniques to show how these models can be further extended and adapted for time-series and sequential data applications.

Chapter 7

Flow Models

In the last chapter, we explored how Deep Latent Variable Models (DLVMs) model complex probability distributions as the integral of the product of two simpler distributions. This approach, while powerful, introduces several complexities, such as the intractability of the marginal probability of the observations. To address this, we often resort to approximations like the Evidence Lower Bound (ELBO).

In this chapter, we will examine a simpler yet effective approach for modeling complex distributions: flow models. Flow models are also latent variable models but circumvent many of the challenges associated with DLVMs described in the previous chapter. They allow us to compute marginals (i.e., $p(\mathbf{x})$) exactly, sample efficiently, and use them for representation learning without the need for complex likelihood approximations. Flow models fall into the category of explicit models with a tractable density, as illustrated in [Figure 7.1](#).

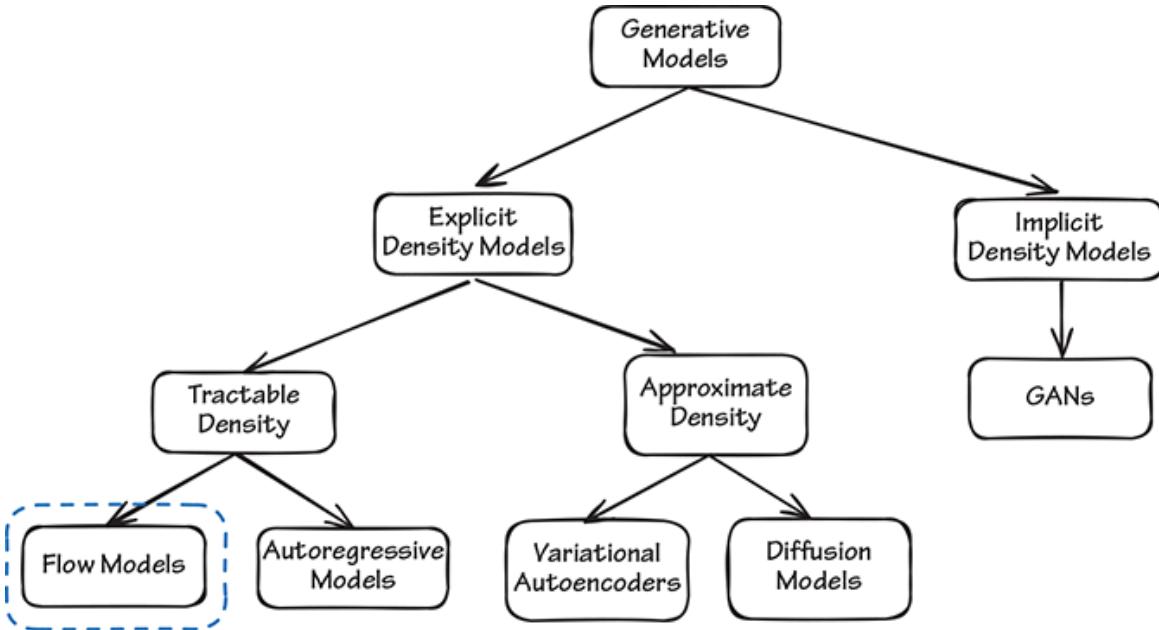


Figure 7.1 Model taxonomy: flow models.

7.1 Introduction

Latent Variable Models (LVMs) are characterized by a probability distribution over the latent variable \mathbf{z} , denoted by $p(\mathbf{z})$, and a conditional probability distribution over the observation \mathbf{x} given the latent variable \mathbf{z} , denoted by $p_{\theta}(\mathbf{x}|\mathbf{z})$. In Deep Latent Variable Models, the parameters of this conditional probability are determined by a Deep Neural Network called the decoder. The decoder takes \mathbf{z} as input and outputs the parameter of the probability distribution over \mathbf{x} . For example, in the case of the Variational Autoencoder (VAE), these parameters are the mean vector and covariance matrix of a Gaussian distribution over \mathbf{x} . In this setting, the value of \mathbf{z} influences the probability density over the possible states of \mathbf{x} , but it does not deterministically specify its state. Thus, the mapping from \mathbf{z} to \mathbf{x} provided by the decoder is probabilistic.

In flow models, we replace the probabilistic decoder $p_{\theta}(\mathbf{x}|\mathbf{z})$, with a deterministic mapping from the latent variable \mathbf{z} to the observation \mathbf{x} , denoted by $\mathbf{x} = f(\mathbf{z})$. In the terminology of flow models, $p_{\mathbf{z}}(\mathbf{z})$ is called the base measure or base distribution, and f is referred as the flow.

In these models, computing the marginal or density of \mathbf{x} involves solving a *derived distribution* problem, where the base measure $p_{\mathbf{z}}(\mathbf{z})$ is given, and we aim to find the probability distribution of \mathbf{x} as a function of the base measure. The distribution of \mathbf{x} is given by the change of variables formula, which allows us to express the probability density $p(\mathbf{x})$ as a function of the known $p(\mathbf{z})$.

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{z}}(\mathbf{z}) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1} \quad (7.1)$$

with $\mathbf{z} = f^{-1}(\mathbf{x})$, and $\mathbf{J} = \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}$ representing the Jacobian matrix.

In flow models, the base measure is typically a simple distribution, such as a normal distribution or a uniform distribution. When the base measure is a normal distribution, the flow model is known as a *normalizing flow*. The goal is to design the flow f to be a highly complex function capable of mapping a simple base measure to a multimodal, complex data distribution.

Compared to deep latent variable models discussed in the previous chapter, normalizing flows allow exact computation of marginals without the need for approximations or lower bounds. Sampling in flow models is also simpler, and can be performed as follows:

$$\begin{aligned}\mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x} &= f(\mathbf{z})\end{aligned}\tag{7.2}$$

Thus, we sample \mathbf{z} from the base measure and apply the flow transformation to obtain the observation \mathbf{x} .

The change of variables formula imposes certain constraints on the family of flows f that we can use. Specifically, for this formula to work, f must be invertible and differentiable. Consequently, the dimensionality of the observations \mathbf{x} must be equal to the dimensionality of the latent variable \mathbf{z} .

To meet these requirements, we need to design special neural network architectures that are invertible and differentiable. This often involves creating innovative and complex deep models that satisfy to these constraints.

7.2 Model Training

We train flow models using maximum likelihood estimation. The log-likelihood is given by:

$$\begin{aligned}\mathcal{L} &= \sum_n \log p_{\mathbf{x}}(\mathbf{x}_n) \\ &= \sum_n \log \left[p_{\mathbf{z}}(\mathbf{z}_n) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1} \right] \\ &= \sum_n \log p_{\mathbf{z}}(\mathbf{z}_n) - \log \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|\end{aligned}\tag{7.3}$$

where $\mathbf{z}_n = f^{-1}(\mathbf{x}_n)$. The primary challenge lies in designing neural network architectures to model a flow f that is invertible and has a Jacobian determinant that can be efficiently computed. This is crucial for ensuring that the

log-likelihood can be computed quickly for potentially millions of data points in our dataset. In the following sections, we will detail and elaborate on some architectures that meet these constraints.

7.3 Linear Flows

Let's revisit the linear factor model from the last chapter, where the prior over \mathbf{z} and the conditional model of $\mathbf{x}|\mathbf{z}$ are given by:

$$\begin{aligned} p_{\mathbf{z}}(\mathbf{z}) &= \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}) \\ p_{\mathbf{x}|\mathbf{z}}(\mathbf{x}|\mathbf{z}) &= \mathcal{N}(\mathbf{x}; \boldsymbol{\mu} + \Lambda \mathbf{z}, \boldsymbol{\Sigma}_0) \end{aligned} \quad (7.4)$$

To convert this linear factor model into a flow model, a simple approach is to replace the probabilistic mapping or probabilistic decoder from \mathbf{z} to \mathbf{x} , defined by the conditional distribution, with a deterministic mapping or flow, represented by f . If we assume the same base measure $p_{\mathbf{z}}(\mathbf{z})$ (prior) and the flow f as $f(\mathbf{z}) = \boldsymbol{\mu} + \Lambda \mathbf{z}$, we obtain what is known as *Linear Normalizing Flows*, see Kobyzev et al. (2021). This can be interpreted as a scale and location transformation of the variable \mathbf{z} . In this setting, Λ must be a $D \times D$ invertible matrix in order for the flow f to be invertible.

Sampling under this model is straightforward. We sample \mathbf{z} from the base measure $\mathbf{z} \sim p_{\mathbf{z}}$, and compute \mathbf{x} using the flow and the realized value of \mathbf{z} : $\mathbf{x} = f(\mathbf{z})$.

As an exercise, let's compute the marginal distribution over \mathbf{x} using the change of variables formula. To apply this formula, we need to know the inverse flow, which is given by $\mathbf{z} = f^{-1}(\mathbf{x}) = \Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu})$, and the Jacobian, which is

equal to Λ . Then, the density over \mathbf{x} can be computed as follows:

$$\begin{aligned}
p_{\mathbf{x}}(\mathbf{x}) &= p_{\mathbf{z}}(\mathbf{z}) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1} \\
&= \mathcal{N}(\mathbf{z}; 0, \mathbf{I}) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1} \\
&= (2\pi)^{-D/2} \exp \left(-\frac{1}{2} \mathbf{z}^T \mathbf{z} \right) |\det \Lambda|^{-1} \\
&= (2\pi)^{-D/2} (\det \Lambda \Lambda^T)^{-1/2} \exp \left(-\frac{1}{2} (\Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu}))^T (\Lambda^{-1}(\mathbf{x} - \boldsymbol{\mu})) \right) \\
&= (2\pi)^{-D/2} (\det \Lambda \Lambda^T)^{-1/2} \exp \left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T (\Lambda \Lambda^T)^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right) \\
&= \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Lambda \Lambda^T)
\end{aligned}$$

where we used the fact that $|\det \Lambda|^{-1} = (\det \Lambda \Lambda^T)^{-1/2}$.

Thus, \mathbf{x} follows a Gaussian distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\Lambda \Lambda^T$. An easier way to compute the marginal distribution over \mathbf{x} is to recognize that a linear transformation of a Gaussian distribution results in another Gaussian distribution. Hence, the probability distribution over \mathbf{x} can be derived by computing its mean vector and covariance matrix:

$$\begin{aligned}
\mathbb{E}[\mathbf{x}] &= \mathbb{E}[\boldsymbol{\mu} + \Lambda \mathbf{z}] = \boldsymbol{\mu} \\
\text{Cov}[\mathbf{x}] &= \Lambda \Sigma_z \Lambda^T = \Lambda \mathbf{I} \Lambda^T = \Lambda \Lambda^T
\end{aligned}$$

Therefore,

$$p_{\mathbf{x}}(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Lambda \Lambda^T)$$

However, linear flows are limited in their expressiveness, as they only modify the mean and covariance matrix of the base distribution. Even if we stack multiple linear transformations, they remain closed under composition. This means that the composition of linear transformations is equivalent to a single linear transformation, which limits the expressiveness of this model. This is insufficient for modeling very complex data distributions. In general, for base measures that are members of the exponential family, a linear transformation remains within the exponential family, potentially eliminating the need for the change of variables formula for computing the probability density over the observations. However, in the case of nonlinear flows, this formula is essential.

Additionally, the determinant of the Jacobian $\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}$

equal to $\det \Lambda$. If Λ is a full matrix, computing this determinant requires $\mathcal{O}(D^3)$ operations, which can be prohibitively expensive for high-dimensional datasets.

One way to reduce the computational complexity of the Jacobian's determinant and its inverse is to impose some particular structure on the linear transformation, in our case on the matrix Λ , that reduce the computational cost of inverse and determinant calculations. For example, [Table 7.1](#) shows useful structures that can be imposed on the matrix used for the linear transformation, along with the computational complexity for computing its inverse and determinant. More useful structures are discussed in Kobyzev et al. (2021) and Brubaker and Köthe (2021).

TABLE 7.1

Computational complexity of the inverse and determinant of the Jacobian based on matrix structure. Here, D denotes the dimension of the latent variable, and c represents the block size for Block Diagonal matrices or the number of channels for 1×1 convolutions.

Adapted from Brubaker and Köthe (2021).

Matrix Type	Inverse Complexity	Determinant Complexity
Full	$\mathcal{O}(D^3)$	$\mathcal{O}(D^3)$
Diagonal	$\mathcal{O}(D)$	$\mathcal{O}(D)$
Triangular	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$
Block Diagonal	$\mathcal{O}(c^3 D)$	$\mathcal{O}(c^3 D)$
LU Factorized	$\mathcal{O}(D^2)$	$\mathcal{O}(D)$
Special Convolution	$\mathcal{O}(D \log D)$	$\mathcal{O}(D)$
1×1 Convolution	$\mathcal{O}(c^3 + c^2 D)$	$\mathcal{O}(c^3)$

7.4 Designing Nonlinear Flows

As we saw in the previous example with a linear flow, its expressiveness is very limited since a composition of linear transformations is equivalent to a single linear transformation. Additionally, we highlighted some of the main requirements that a flow must satisfy for the change of variables formula to work, as well as some key

properties for model expressiveness and computational efficiency. We want flows that are:

- Expressive enough to model complex data distributions
- Invertible
- Differentiable
- Computationally efficient to invert (f^{-1})
- Computationally efficient to compute the Jacobian determinant

Flows are also called flow layers or bijections, see Kobyzev et al. (2021).

From deep neural networks, we know that an easy way to improve model expressiveness is through function composition. An important aspect of flow models is that the composition of invertible flows or layers remains invertible. So, if we compose K different flows, such as $f = f_K \circ f_{K-1} \circ \dots \circ f_2 \circ f_1$, and each f_k is invertible and differentiable, then f will also be invertible and differentiable.

Additionally, the determinant of the Jacobian for the composed function has a very convenient form:

$$\begin{aligned}
\det \mathbf{J} &= \det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} & (7.5) \\
&= \det \prod_{k=1}^K \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k} \\
&= \prod_{k=1}^K \det \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k} \\
&= \prod_{k=1}^K \det \mathbf{J}_k
\end{aligned}$$

where $\mathbf{J}_k = \frac{\partial f_k(\mathbf{z}_k)}{\partial \mathbf{z}_k}$. Thus, under the composition of flows,

the determinant of the Jacobian \mathbf{J} is equal to the product of the determinants of the Jacobians of the individual flows, which provides significant computational advantages. This form simplifies the computation of log-likelihoods to the sum of the log-determinants of each function involved in the composition.

In the following sections, we will describe one main way of designing nonlinear flows that are used in practice—coupling flows. Coupling flows are based on the principle of composition of invertible layers, a principle widely applied to the design of deep invertible neural network flows. Within this class, we will describe two specific cases: Nonlinear Independent Components Estimation (NICE) and Non-volume Preserving Transformation (Real-NVP).

7.5 Coupling Flows

Coupling flows are a widely used approach for constructing nonlinear flows (Dinh et al., 2015). The main idea is to partition the vector \mathbf{z} into two parts:

$$\mathbf{z} = \begin{bmatrix} \mathbf{z}_{1:D'} \\ \mathbf{z}_{D'+1:D} \end{bmatrix} \quad (7.6)$$

To transform \mathbf{z} into \mathbf{x} , we apply the flow f , such that $\mathbf{x} = f(\mathbf{z})$. In the same manner, we can partition \mathbf{x} :

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{1:D'} \\ \mathbf{x}_{D'+1:D} \end{bmatrix} \quad (7.7)$$

A flow f is considered a coupling flow if:

$$f(\mathbf{z}) = \begin{bmatrix} \mathbf{x}_{1:D'} \\ \mathbf{x}_{D'+1:D} \end{bmatrix} = \begin{bmatrix} \mathbf{z}_{1:D'} \\ f_1(\mathbf{z}_{D'+1:D}; \Theta(\mathbf{z}_{1:D'})) \end{bmatrix} \quad (7.8)$$

where the first partition of $\mathbf{z}(\mathbf{z}_{1:D'})$ remains unaffected by the flow f , and f_1 is another invertible transformation (another flow) with parameters θ , applied to the second partition of $\mathbf{z}(\mathbf{z}_{D'+1:D})$.

The parameters of the function f_1 , denoted as θ , are defined by an arbitrary function Θ . In practice, Θ is modelled by a neural network, whose input depend only on the first partition of $\mathbf{z}(\mathbf{z}_{1:D'})$ and not in the second partition $\mathbf{z}(\mathbf{z}_{D'+1:D})$. The function Θ is referred to as a **conditioner**, which does not need to be invertible. The function f_1 is known as a **coupling function** or **coupling transform**, and must be a monotonic function of its argument, therefore invertible. The function f is called a **coupling flow** or **coupling layer**. As we will see in the following examples, different well-known coupling flows differ in how they define the conditioner and the coupling function. Specifically, we will examine two cases: one utilizing an additive coupling function and another employing an affine coupling function.

This structure has several advantages for building complex nonlinear mappings, which are invertible. By stacking multiple “coupling layers,” we can create highly expressive flows while keeping the computation of the Jacobian determinant efficient, as we will see in the next section.

As an illustration of the process, [Figure 7.2](#) demonstrates how the vector \mathbf{z} is partitioned and then transformed using the flow f as described.

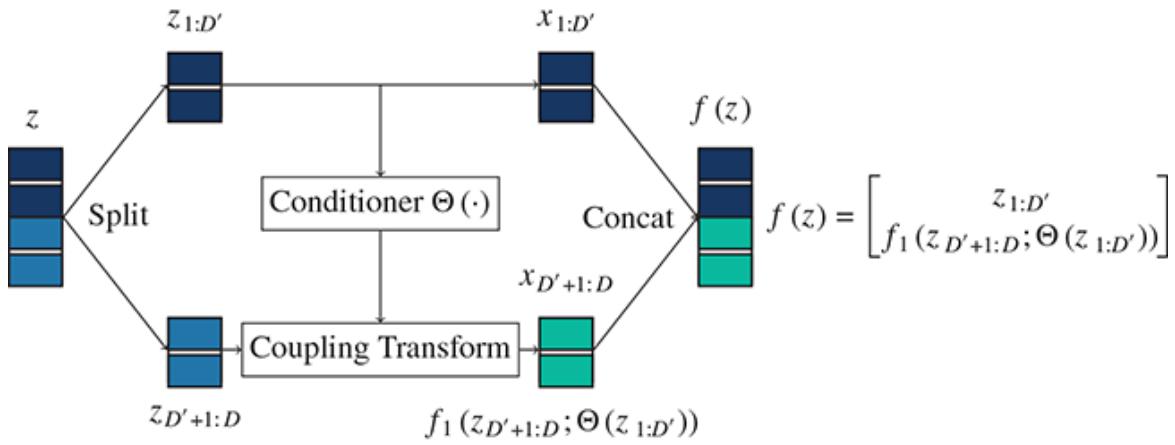


Figure 7.2 Illustration of the operations involved in coupling flows, including partitioning and transformation using a coupling layer.

Adapted from Brubaker and Köthe (2021).

In the following section, inspired by the exposition style of Levine (2021), we will explore two specific implementations of coupling layers: **Nonlinear Independent Components Estimation (NICE)** and **Non-volume Preserving Transformation (Real-NVP)**.

7.5.1 NICE: Nonlinear Independent Components Estimation

In this section, we explain the approach to designing invertible layers based on the work by Dinh et al. (2015),

known as NICE: Nonlinear Independent Components Estimation.

As mentioned before, the approach involves partitioning the vector \mathbf{z} into two parts, described as $\mathbf{z} = [\mathbf{z}_{1:D'}, \mathbf{z}_{D'+1:D}]^T$. Due to the constraints of the invertible mapping, the dimension of \mathbf{x} must also be equal to D . We decompose the entries of \mathbf{x} also into two components, namely $\mathbf{x} = [\mathbf{x}_{1:D'}, \mathbf{x}_{D'+1:D}]^T$. In NICE, the invertible layer is defined as follows:

$$\begin{aligned}\mathbf{x}_{1:D'} &= \mathbf{z}_{1:D'} \\ \mathbf{x}_{D'+1:D} &= \mathbf{z}_{D'+1:D} + g_{\theta}(\mathbf{z}_{1:D'})\end{aligned}\tag{7.9}$$

NICE uses an additive coupling function, which is an invertible transformation with parameters defined by $g_{\theta}(\mathbf{z}_{1:D'})$. In practice, g_{θ} is implemented using a neural network, such as an MLP, a CNN, or other architectures tailored to the specific problem. Given the value \mathbf{x} , we can recover the vector \mathbf{z} by first recovering its first D' entries: $\mathbf{z}_{1:D'} = \mathbf{x}_{1:D'}$, and then using this result to recover the second part of the entries $\mathbf{z}_{D'+1:D} = \mathbf{x}_{D'+1:D} - g_{\theta}(\mathbf{z}_{1:D'})$.

The Jacobian of this mapping is given by:

$$\begin{aligned}\mathbf{J} = \frac{\partial f(\mathbf{z})}{\partial z} &= \begin{bmatrix} \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{D'+1:D}} \\ \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{(D'+1):D}} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I}_{D'} & \mathbf{0} \\ g'_{\theta}(\mathbf{z}_{1:D'}) & \mathbf{I}_{D-D'} \end{bmatrix}\end{aligned}\tag{7.10}$$

where $\mathbf{I}_{D'}$ is an identity matrix with D' diagonal elements.

Since we are interested in the determinant of the Jacobian, we do not need to compute the $(2, 1)$ entry of \mathbf{J} , as the $(1, 2)$ of \mathbf{J} is zero, and $g'_{\theta}(\mathbf{z}_{1:D'})$ will not contribute to the result. Thus, the determinant of the Jacobian is $\det \mathbf{J} = 1$.

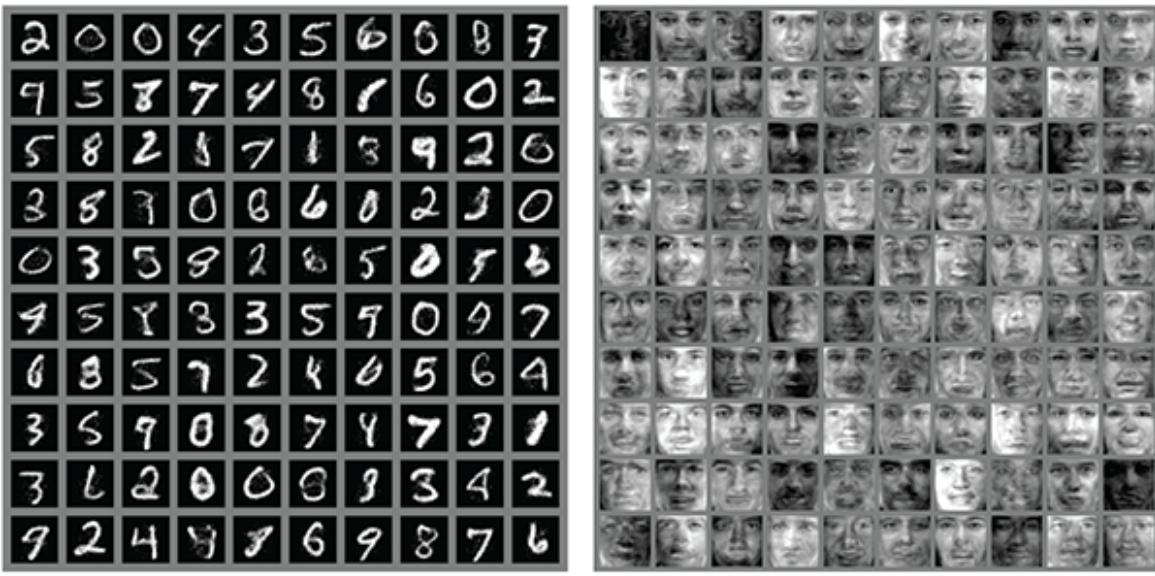
The log-likelihood for this model is given by:

$$\begin{aligned}\mathcal{L} &= \sum_n \log p(\mathbf{z}_n) - \log \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial z} \right) \right| \quad (7.11) \\ &= \sum_n \log p(\mathbf{z}_n) - \log 1 \\ &= \sum_n \log p(\mathbf{z}_n)\end{aligned}$$

which is computationally convenient. However, the fact that the determinant of the Jacobian is constant and equal to 1 means that this flow only transforms the base measure but does not rescale it. This limits the expressiveness of the model and makes it difficult to fit very complex data distributions (see Levine [2021]). Therefore, in practical settings, a rescaling layer is often inserted into the model (see Dinh et al. [2015]).

In practice, we stack several of these invertible layers to train generative flow models effectively. As an illustrative example of how these models perform on simple image data, in [Figure 7.3](#), subfigures (a) and (b) show samples from a NICE model that has been trained to model the distribution of the simple MNIST digits dataset and basic images from the TFD dataset. The architecture used in both experiments consists of a stack of four NICE layers/functions. Each NICE layer is implemented as a deep rectified network with linear output units. The authors use the same network architecture for all NICE functions: five hidden layers with 1,000 units each for the MNIST dataset and four hidden layers with 5,000 units each for the TFD

dataset. As observed, the generated digit images (subfigure a) look like real digits from the MNIST dataset, and the generated simple face images (subfigure b) resemble faces to some extent. This model, developed in 2015, was the first normalizing flow model capable of generating images of reasonable quality for its time. We will discuss how to extend these models to capture temporal dependencies in [Section 7.8](#), as well as an example in financial applications in [Section 7.8.3](#).



(a) Model trained on MNIST

(b) Model trained on TFD

Figure 7.3 Illustration of unbiased samples generated by the NICE model when trained on the MNIST dataset (subfigure a) and the TFD dataset (subfigure b).

Source: Dinh, Krueger, and Bengio (2015).

However, when the model is trained on more complex datasets, as shown in [Figure 7.4](#), its performance deteriorates. In subfigure (c), we see samples generated from a model trained on the SVHN dataset, which consists of color images of digits taken from house numbers. The model begins to struggle to generate these kinds of color digits. When trained on CIFAR-10, which is a more complex

dataset containing a diverse set of realistic object images, the model generates very low-quality images, showing its limitations in handling more complex data distributions.

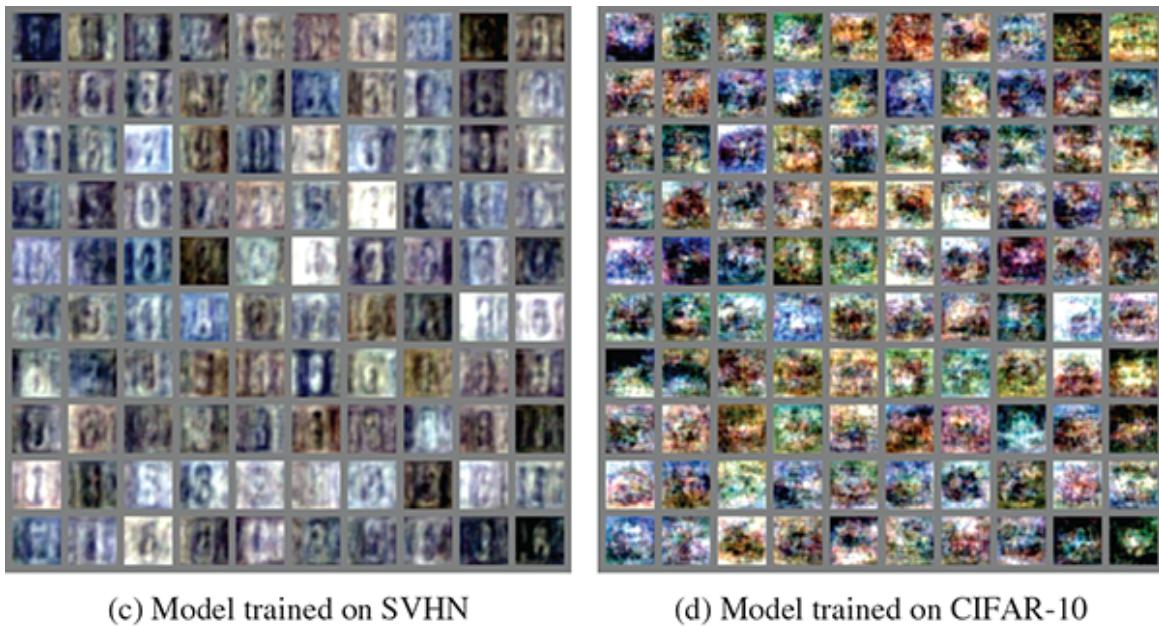


Figure 7.4 Illustration of unbiased samples generated by the NICE model when trained on the SVHN dataset (subfigure c) and on the CIFAR-10 dataset (subfigure d).

Source: Dinh, Krueger, and Bengio (2015).

In the accompanying notebook for this chapter ([*NICE Notebook*]), we implement and test the NICE coupling flow with custom architectures for generating equity financial data, which can be easily customized for your own applications. Additionally, we include experimental results to demonstrate the performance and practical applications of this approach.

7.5.2 Real-NVP: Non-volume Preserving Transformation

The Non-volume Preserving Transformation (Real-NVP) introduced by Dinh et al. (2017) proposes a new kind of invertible layer that enhances the model's expressiveness. In particular, this model's Jacobian determinant is not constant, allowing for greater flexibility. Similar to the NICE model, in Real-NVP, they partition the vectors \mathbf{z} and \mathbf{x} into two parts, namely $\mathbf{z} = [\mathbf{z}_{1:D'}, \mathbf{z}_{D'+1:D}]^T$ and $\mathbf{x} = [\mathbf{x}_{1:D'}, \mathbf{x}_{D'+1:D}]^T$. The proposed invertible layer (coupling function) is defined as follows:

$$\begin{aligned}\mathbf{x}_{1:D'} &= \mathbf{z}_{1:D'} \\ \mathbf{x}_{D'+1:D} &= \mathbf{z}_{D'+1:D} \odot \exp(h_{\theta}(\mathbf{z}_{1:D'})) + g_{\theta}(\mathbf{z}_{1:D'})\end{aligned}\tag{7.12}$$

where \odot stands for the element-wise product, and h_{θ} and g_{θ} are neural network layers that can be implemented using architectures such as MLPs, CNNs, or other designs tailored to your specific problem. As defined in [Equation 7.12](#), Real-NVP uses an affine coupling function, which is by definition an invertible transformation with parameters given by $\{h_{\theta}(\mathbf{z}_{1:D'}), g_{\theta}(\mathbf{z}_{1:D'})\}$. This can be interpreted as a scale and location transformation of the second partition of $\mathbf{z}(\mathbf{z}_{D'+1:D})$.

To recover the vector \mathbf{z} from \mathbf{x} , we apply a transformation similar to that used in the NICE model:

$$\begin{aligned}\mathbf{z}_{1:D'} &= \mathbf{x}_{1:D'} \\ \mathbf{z}_{D'+1:D} &= (\mathbf{x}_{D'+1:D} - g_{\theta}(\mathbf{z}_{1:D'})) / \exp(h_{\theta}(\mathbf{z}_{1:D'}))\end{aligned}\tag{7.13}$$

Given this mapping, we can derive the Jacobian, which is given by:

$$\begin{aligned}
\mathbf{J} = \frac{\partial f(\mathbf{z})}{\partial z} &= \begin{bmatrix} \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{1:D}} & \frac{\partial(\mathbf{x}_{1:D'})}{\partial \mathbf{z}_{D'+1:D}} \\ \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{1:D'}} & \frac{\partial(\mathbf{x}_{D'+1:D})}{\partial \mathbf{z}_{(D'+1):D}} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{I} & 0 \\ g'_{\theta}(\mathbf{z}_{1:D'}) & \text{diag}(\exp(h_{\theta}(\mathbf{z}_{1:D'}))) \end{bmatrix}
\end{aligned} \tag{7.14}$$

The Jacobian is triangular, so its determinant is the product of diagonal entries:

$$\begin{aligned}
\det \mathbf{J} &= \prod_{d=D'+1}^D \exp(h_{\theta}(\mathbf{z}_{1:D'})_d) \\
&= \exp\left(\sum_{d=D'+1}^D h_{\theta}(\mathbf{z}_{1:D'})_d\right)
\end{aligned} \tag{7.15}$$

Here we can see that the determinant of the Jacobian is not constant, allowing for the rescaling of the base measure and enabling greater expressiveness for the model to fit very complex data distributions. This makes the Real-NVP model more expressive compared to the NICE model.

Once again, as an illustrative example of how this performs on simple image data, [Figure 7.5](#) shows samples generated by the RealNVP model, see Dinh et al. (2017). The authors of the paper present results of a RealNVP model trained on various datasets: CIFAR-10, ImageNet (32×32), ImageNet (64×64), CelebA, and LSUN (bedroom). In each pair of columns, the left column shows examples from the dataset, while the right column displays samples generated by the model trained on that dataset.



Figure 7.5 Samples generated by the Real-NVP model across four datasets: CIFAR-10, ImageNet (32 × 32), ImageNet (64 × 64), CelebA, and LSUN (bedroom). For each dataset (subfigures), the left column shows real examples, while the right column displays samples generated by the model trained on the respective dataset.

Source: Dinh, Sohl-Dickstein, and Bengio (2017).

The authors of this work employed a multi-scale architecture, as described in [Section 3.6](#) of Dinh et al. (2017). This architecture incorporates a sequence of coupling-squeezing-coupling layers, utilizing deep convolutional residual networks within the coupling transformations. For a more detailed explanation of the

architecture used in the original experiments, we encourage readers to refer to the work by Dinh et al. (2017).

As observed, this model can generate much more realistic samples, including images of various objects, bedrooms, natural scenes, and human faces.

Another interesting aspect demonstrated in the RealNVP paper is the model's ability for representation learning. Recall from the beginning of the chapter that one limitation of these models being invertible is that the dimensions of the latent variable \mathbf{z} and the observed variable \mathbf{x} must be the same, with \mathbf{x} being high dimensional. Normally, we prefer the latent dimension to be lower than the data dimension, as in the VAEs studied in [Chapter 6](#), to provide a compressed representation of the input data.

In [Figure 7.6](#), the authors showcase Real-NVP's ability to capture a useful representation of the data in the latent space. This figure shows new images generated by interpolations in the latent space between four examples for different datasets. The datasets used, in clockwise order from the top left, are CelebA, ImageNet (64×64), LSUN (tower), and LSUN (bedroom).

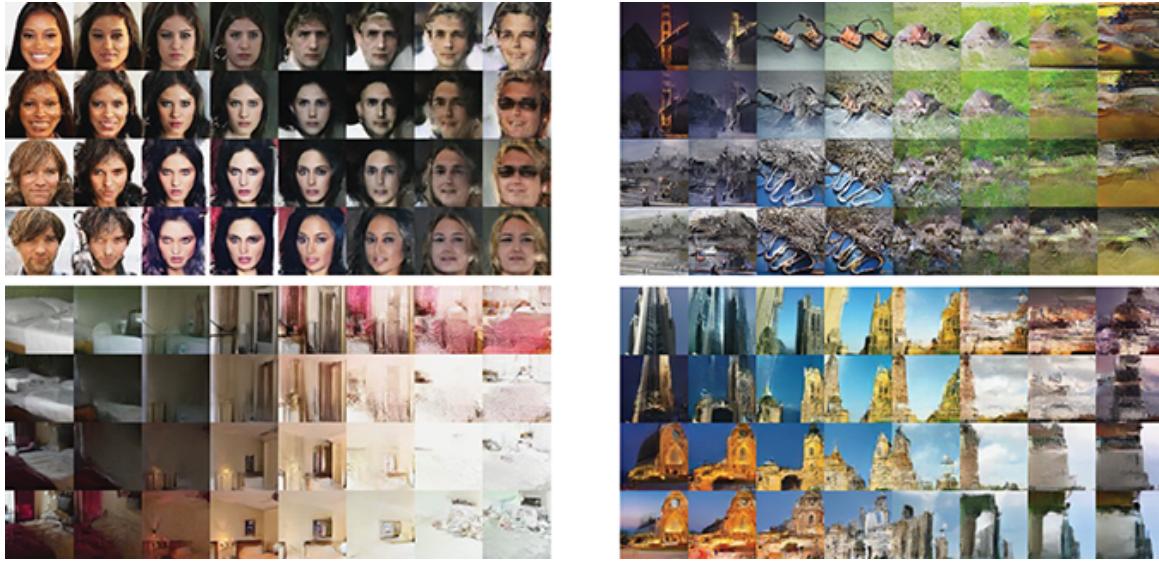


Figure 7.6 Illustration of new images generated through interpolations between four examples from the dataset. The datasets used are, in clockwise order from the top left: CelebA, ImageNet (64×64), LSUN (tower), and LSUN (bedroom).

Source: Dinh, Sohl-Dickstein, and Bengio (2017).

For each dataset, four examples are placed at the corners of each subimage, and the interpolations between them are generated. The results suggest that even if the latent space is high dimensional, the model still learns an interesting structure, where similar images in the data space are also close in the latent space. This indicates that the model is capturing a meaningful and semantic representation that goes beyond simple pixel-space interpolation.

To further test whether the latent space has a consistent semantic interpretation, the authors trained a class-conditional model on the CelebA dataset. They found that the learned representation had a consistent semantic meaning across class labels, as described in Dinh et al. (2017).

In the accompanying notebook for this chapter ([*RealNVP Notebook*]), we implement and test the Real-NVP coupling

flow with custom architectures for generating equity financial data, which can be easily customized for your own applications. We will discuss how to extend these models to capture temporal dependencies in [Section 7.8](#), along with an application to financial data in [Section 7.8.3](#).

In the following sections, we will briefly introduce two additional approaches for constructing flows:

Autoregressive Flows and **Continuous Normalizing Flows**.

7.6 Autoregressive Flows

Another type of flow model is known as **Autoregressive Flows**, which are, in fact, one of the first classes of flows developed (see Papamakarios et al. [2021]). These flows are motivated by the idea that the joint distribution over \mathbf{x} can be factorized into a product of one-dimensional conditional probability distributions using the chain rule, as introduced in [Chapter 5](#) on autoregressive models. By parameterizing each conditional distribution, these factors form the basis of a normalizing flow known as the **Masked Autoregressive Flow (MAF)** (see Papamakarios et al. [2017]; Bishop and Bishop [2023]).

Some of our experiments in [Section 7.8](#) will incorporate MAF flows as building blocks in a model's architecture. Regarding the modeling flexibility of Autoregressive flows, an interesting result, presented in Papamakarios et al. (2021), shows that all autoregressive models for continuous variables can be interpreted as autoregressive flows with a single autoregressive layer. If you want to know more about this result, refer to Papamakarios et al. (2021) for further details.

7.7 Continuous Normalizing Flows

Another type of flow model is known as **Continuous Normalizing Flows**, which take a different approach to define the mapping from \mathbf{z} to \mathbf{x} . In the previous sections, we discussed constructing flows by parameterizing a function f and applying a finite sequence of transformations (e.g., K coupling layers in Real-NVP) to transform the base measure $p(\mathbf{z})$ into a more complex distribution $p(\mathbf{x})$.

In contrast, Continuous Normalizing Flows view the mapping from \mathbf{z} to \mathbf{x} as a **continuous process**. Instead of applying a finite number of discrete transformations, the transformation can be thought of, loosely speaking, as applying an “infinite number of infinitesimal transformations.” This process is mathematically described by a **Neural Ordinary Differential Equation (Neural ODE)**. In this framework, the flow is defined by an ODE, and during the learning process, the parameters of the ODE are learned from the data, rather than learning the parameters of individual discrete flows, as in the previous cases we studied. Continuous Normalizing Flows can be used to generate the complex continuous stochastic process of financial data (see Deng et al. [2020]). For more details on Continuous Normalizing Flows, we encourage the reader to take a look at Chen et al. (2018) and Papamakarios et al. (2021).

7.8 Modeling Financial Time Series with Flow Models

7.8.1 Transitioning from Image Data to Time-series Dynamics

In earlier sections, we introduced two flow models, NICE and Real-NVP. These models create flows that are differentiable, invertible, and allow for quick computation of log-likelihoods. The efficiency in log-likelihood computation is mainly due to the structure imposed in these flows, which simplifies the computation of the determinant of the Jacobian. While so far we had used image data as examples to demonstrate the capacity of these models to approximate complex data distributions (and to highlight the field where these models were originally developed), flows are not restricted to this domain and can be used whenever we need to model complex probability distributions. They can be extended to other areas, such as time-series modeling, unlocking new modeling opportunities.

To transition into time series, we can start by simply giving another interpretation to the data \mathbf{x} . Instead of thinking as \mathbf{x}_n as a vector of pixel intensities for the n -th image in our dataset, we can think of \mathbf{x}_n as a vector representing multivariate time series at time n . Each component or “dimension” might correspond to a variable, such as the returns of a given stock at time n , with the vector \mathbf{x}_n providing a snapshot of the cross-sectional returns at that time. For simplicity, we will use t to index time instead of n , as it is more intuitive in the context of time series.

In this setup, flows like NICE or Real-NVP can model the cross-sectional relationships (i.e., correlations) among the

time series at time t . However, by design, these architectures assume independence between consecutive observations. Recall how sampling works in standard flow models: to generate the observations \mathbf{x}_t , we sample \mathbf{z}_t from the base measure, apply the flow to it to produce \mathbf{x}_t , and repeat this process for every t . This assumption of independence can limit the performance of these models at certain time scales, where dependencies between observations may be significant.

7.8.2 Adapting Flows for Time Series

Conventional flows like NICE and Real-NVP are good at modeling cross-sectional dynamics but impose independence between observations by design. This might not be enough for modeling financial time series at certain time granularities, where past prices/returns, market conditions, and other factors significantly impact future prices/returns.

So, how can we adapt flows to capture the temporal dynamics (dependencies over time) present in the data we want to model? Here are a few ideas:

1. Latent state dynamics: Introduce temporal dependencies into the latent variables, such that \mathbf{z}_t depends on past latent variables $\mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \dots$, either deterministically or stochastically.

$$\begin{aligned}\mathbf{z}_t &\sim p_{\mathbf{z}_t | \mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \dots} \\ \mathbf{x}_t &= f(\mathbf{z}_t)\end{aligned}$$

This mirrors the idea behind models like the Kalman filter, or RNN, where latent states depends on the previous latent space.

2. AR-style dependencies: Extend the generative process to depend on past observations, making \mathbf{x}_t a function of both the latent variable \mathbf{z}_t and previous observations $\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots,$

$$\mathbf{x}_t = f(\mathbf{z}_t, \phi(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_{t-p}))$$

where ϕ is a function that can encode the effect of past p observations on \mathbf{x}_t . This approach mirrors with autoregressive models.

3. Hybrid approaches: Combine the two methods, allowing \mathbf{x}_t to depend both of past observations and past latent variables

$$\mathbf{x}_t = f(\mathbf{z}_t, \phi(\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \dots, \mathbf{x}_{t-p}), \psi(\mathbf{z}_{t-1}, \mathbf{z}_{t-2}, \dots, \mathbf{z}_{t-q}))$$

where ϕ encodes the effect of past p observations and ψ the effect of past q latent variables.

4. Time-varying flows: allow the flow to vary with time f_t .

7.8.3 Case Study: A Practical Example—Conditioned Normalizing Flows

To demonstrate these ideas, let's explore practical implementations of flows adapted for modeling time-series dynamic. One notable example is presented in the paper *Multivariate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows* (Rasul et al., 2021). In this work, the authors use Real-NVP flows but adapt them to handle temporal dependencies. The key idea is to condition the flow transformation f on a hidden state \mathbf{h}_t , which encodes information about the past:

$$\mathbf{x}_t = f(\mathbf{z}_t, \mathbf{h}_t)$$

So, how do we get \mathbf{h}_t ? The authors explore two approaches.

1. RNN-based hidden state: the first option is to use an RNN (like an LSTM or GRU) to summarize past observations and covariates into \mathbf{h}_t :

$$\mathbf{h}_t = \text{RNN}(\text{concat}(\mathbf{x}_{t-1}, \mathbf{c}_{t-1}), \mathbf{h}_{t-1})$$

Here \mathbf{c}_{t-1} includes any additional covariates (i.e., in financial time series think like additional market indicators or macroeconomic variables, or categorical variables).

2. Transformer-based hidden state: The second option is to use an encoder-decoder transformer architecture. Here, the encoder embeds the past observations and covariates, while the decoder generates the hidden states for the flow.

As we know, both RNNs and Transformers are effective at modeling sequential data, with Transformers being particularly better than RNN for capturing long-range dependencies.

At this point, you might notice that we are only limited by our creativity when designing architectures to capture temporal dynamics with flows. This is another example of how the building blocks we've covered in previous chapters come together—combining flows with powerful sequential models like RNNs or Transformers to achieve the same goal.

Exploring examples like this one can bring to life many ideas for customizing your own solutions. You might consider training the model differently, adding or removing components, or adapting the architecture to better fit your specific needs.

7.8.3.1 Importance of Domain Knowledge in Financial Time Series.

In financial time-series modeling, incorporating domain knowledge during the modeling process is critical. This field is well known for its low signal-to-noise ratio and non-stationarity is all over the place, which violates the core assumptions of independence and identical distribution that underlie most machine learning solutions. These challenges make it difficult to achieve good results with generic models.

This means that most of the time we can't simply throw a generic model at the data and expect great outcomes in this domain. Who knows—perhaps in the near future, maybe even a few months after this book is published, this might change. For now, let's focus on customizing our solutions to the specific problem at hand so we can extract valuable insights and, hopefully, some profit.

In the accompanying notebook for this chapter, we explore these ideas using the Exchange Rate dataset, which contains daily exchange rates for the currencies of eight countries (Australia, the United Kingdom, Canada, Switzerland, China, Japan, New Zealand, and Singapore) from 1990 to 2016. We will walk through the main components of the implementation and discuss variations in the architecture and its performance on other datasets.

7.9 Conclusion

In this chapter, we have seen how flow models are a powerful addition to our toolbox for handling high-dimensional data. They combine the fast sampling capabilities of traditional latent variable models with the key advantage of exact likelihood computation. By

designing flows with the right structure, we can make computations like the determinant of the log Jacobian very efficient, which is a key element for likelihood computation.

We also explored how to extend flows to capture temporal dependencies. This opens up opportunities to combine flow models with architectures that we know excels at this task, such as autoregressive models based on RNN and Transformers, both of which work great at modeling sequential data.

By mastering flow models, you add a flexible and efficient tool to your modeling toolbox, giving you another great option for modeling high-dimensional multimodal data.

Chapter 8

Generative Adversarial Networks

In previous chapters, we explored modeling high-dimensional complex data distributions using Autoregressive Models, Variational AutoEncoders (VAEs), and Flow Models. While each of these models has its advantages and disadvantages in terms of computing densities—whether exactly or through approximations, sampling efficiency, and representation learning—they all share a common characteristic: they are likelihood-based models. This means they use Maximum Likelihood Estimation (MLE) as the primary approach for learning their parameters. In the generative learning literature, these kinds of models are known as explicit models.

In this chapter, we will delve into a different approach for learning generative models, referred to as implicit models in the literature, as shown in [Figure 8.1](#). These models do not require the ability to compute the probability density. This chapter will focus on Generative Adversarial Networks (GANs) as a prominent example of implicit models, with a training process that focuses on generating high-quality realistic high-dimensional samples rather than performing density estimation.

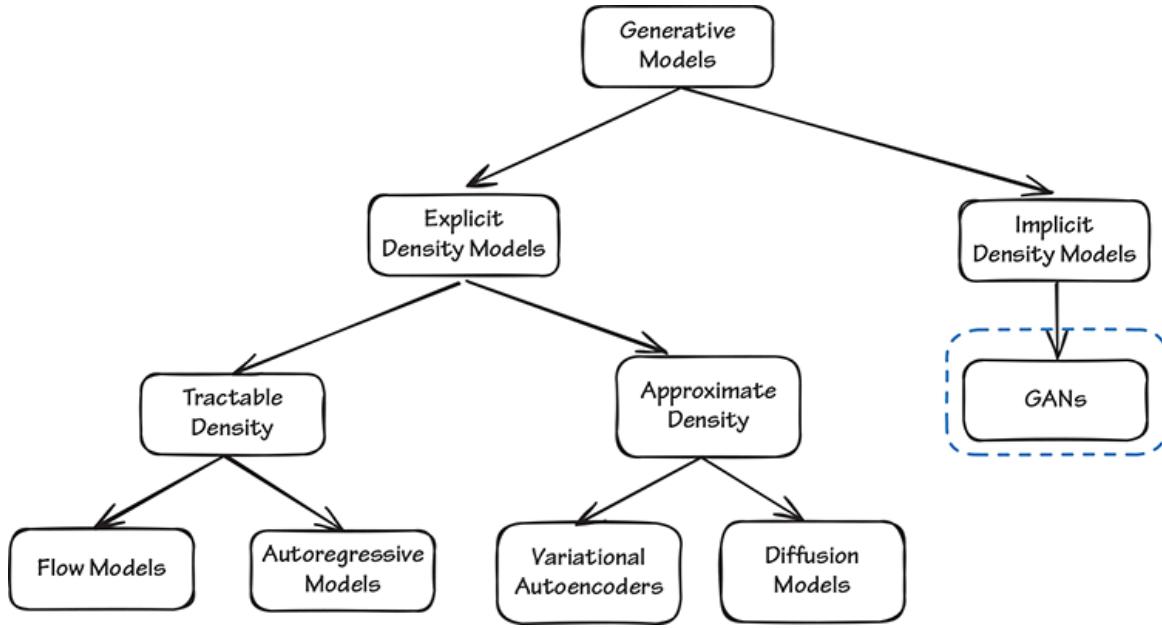


Figure 8.1 Model taxonomy: GANs.

8.1 Introduction

In previous chapters, we discussed the primary goals of generative models: to build models for high-dimensional data that can accomplish the following:

- Compute probability densities
- Perform representation learning (identifying hidden structures)
- Generate new samples

Choosing a single metric that measures the quality all these objectives can be challenging. A specific metric or training objective might be good for representation learning but not for generating high-quality samples, for example. However, intuitively, if a model can closely approximate the data distribution, it should, in theory, perform well in the other tasks. Model families like Autoregressive Models, VAEs, and Flow Models use this principle and during the learning

process, try to align the model distribution as closely as possible to the data distribution. This alignment ensures that the resulting model is not only a good density estimator but also provides useful hidden structures and high-quality samples. All these model families propose a tractable or approximate density model class such that the learning can be based on Maximum Likelihood Estimation, which we saw is equivalent to minimizing the Kullback-Leibler divergence between the data distribution and the model distribution.

Generative Adversarial Networks (GANs), on the other hand, focus their training process on producing a model that can generate high-quality samples without the need to compute density estimation explicitly. The inventor of GANs, Ian Goodfellow, described his solution in his paper (Goodfellow et al., 2014), detailing how this can be achieved. His very clever idea is to use a different neural network, called the discriminator, whose sole task is to score whether an observation sample has been generated by the data distribution or by the model distribution and to use this signal provided by the discriminator to improve the model distribution. The model distribution is called the generator in GAN terminology. Ideally, a good generator will produce samples such that one cannot distinguish between the distribution generated by the model and the actual data distribution.

Yann LeCun who received the 2018 Turing Award, together with Yoshua Bengio and Geoffrey Hinton, for their work on deep learning, declared GANs to be “the most interesting idea in the last 10 years in machine learning.” As a historical note, Ian Goodfellow conceived generative adversarial networks while brainstorming programming techniques with friends at a bar. For more details, read the interview at <https://www.deeplearning.ai/the-batch/ian-goodfellow-a-man-a-plan-a-gan/>.

Similar to VAEs or Flow Models, in GANs, the sampling process for the generator consists of first sampling a latent variable \mathbf{z} from a prior distribution $p_{\mathbf{z}}$, which is chosen to be easy to sample from, and then mapping the latent variable space to the observation or data space using the generator, denoted by G_{θ} and parameterized by θ . This mapping can be either probabilistic or deterministic. For a deterministic mapping G_{θ} , sampling can be computed very efficiently, as in Flow Models. The sampling process in GANs is given by:

$$\begin{aligned}\mathbf{z} &\sim p(\mathbf{z}) \\ \mathbf{x} &= G_{\theta}(\mathbf{z})\end{aligned}\tag{8.1}$$

A key difference from Flow Models is that, because we are not using the density for model learning, the generator can be any neural network. There is no requirement that the mapping be invertible, and the dimensions of the latent variable \mathbf{z} and the observations \mathbf{x} can be different, as in VAEs, which can provide interesting learned representations.

Unlike Flow Models, where we train only the encoder or flow, GANs require learning the parameters of two different models simultaneously: the generator neural network G_{θ} and the discriminator neural network, denoted by D_{ϕ} and parameterized by ϕ .

The big-picture idea of what happens during training is the following. We train the discriminator, which is a binary classifier that classifies real data (generated by the data distribution), versus fake data (generated by the generator), optimizing its parameters to get better at discriminating these classes. To train the discriminator, we build the “real dataset,” denoted by \mathcal{D}_T , by collecting real data related to our application (e.g., images of faces) and

assigning the target label true or positive (or 1) to all of them. We build the “fake dataset,” denoted by \mathcal{D}_F , by randomly initializing the generator G_θ , sampling a number of images, and assigning the target label false or negative (or 0) to all of them. With the combined dataset

$\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$, we train the discriminator D_ϕ . Once the discriminator is trained, we use the signal provided by the discriminator as a loss function to train G_θ , and iterate.

Thus, the GAN training process involves a game where the discriminator tries to get better at classifying real versus fake data, and the generator tries to get better at fooling the discriminator into classifying fake data as real. This is the adversarial part of the game, which can eventually reach an equilibrium called the Nash Equilibrium.

GANs are known for generating very high-quality high-dimensional images. In [Figure 8.2](#), we see a progression of samples generated by GANs over a span of 4.5 years, from the original work by Goodfellow et al. (2014) in 2014 to 2018. However, GANs are known to be difficult to train. In the following sections, we will delve into more details about the training process and what GANs are actually learning. The following exposition was inspired by the style presented in Levine (2021).



Figure 8.2 Evolution of GAN-generated images over time, illustrating the progressive improvement in image generation. This figure is adapted from a tutorial by Ian Goodfellow titled “Ian Goodfellow: Adversarial Machine Learning” at ICLR 2019. The image is sourced from a screenshot taken at the 6:47 mark of the presentation.

Source: Ian Goodfellow: Adversarial Machine Learning (ICLR 2019 invited talk) / <https://www.youtube.com/watch?v=sucqskXRkss> / last accessed on January 17, 2025.

8.2 Training

Training GANs involves solving a two-player game between the discriminator and the generator, as described in the original formulation by Goodfellow et al. (2014). The goal is to solve the following min-max optimization problem:

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} V(\boldsymbol{\theta}, \boldsymbol{\phi}) \quad (8.2)$$

where the objective function $V(\boldsymbol{\theta}, \boldsymbol{\phi})$ is given by:

$$V(\boldsymbol{\theta}, \boldsymbol{\phi}) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_{\boldsymbol{\phi}}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\mathbf{z})))] \quad (8.3)$$

In this formulation, $V(\boldsymbol{\theta}, \boldsymbol{\phi})$ represents the loss function of the discriminator $D_{\boldsymbol{\phi}}$, which takes an input and outputs the probability that it belongs to the true dataset (real observations). This loss function has the form of a typical cross-entropy loss (negative log likelihood) used in binary classification problems (see Murphy [2022]; López de Prado, 2018), where the goal in this particular case is to distinguish real from fake data. The discriminator aims to maximize this objective, while the generator aims to minimize it, creating an adversarial training dynamic. The discriminator seeks to maximize the log-likelihood of real data (positive labels) and the log-likelihood of fake data (negative labels). Thus, in the previous equation, the first expectation is taken with respect to the distribution of real data, p_{data} , and the second expectation is taken with respect to the latent variable distribution, $p_{\mathbf{z}}$, which is used to generate the fake dataset.

These expectations are approximated using sample-based estimators. The objective function can be approximated by:

$$\begin{aligned} V(\boldsymbol{\theta}, \boldsymbol{\phi}) &= \sum_{n=1}^N \log D_{\boldsymbol{\phi}}(\mathbf{x}_n) + \sum_{m=1}^M \log(1 - D_{\boldsymbol{\phi}}(G_{\boldsymbol{\theta}}(\mathbf{z}_m))) \\ &= \sum_{n=1}^N \log D_{\boldsymbol{\phi}}(\mathbf{x}_n) + \sum_{m=1}^M \log(1 - D_{\boldsymbol{\phi}}(\mathbf{x}_m)) \end{aligned} \quad (8.4)$$

where $\mathbf{x}_m = G_{\boldsymbol{\theta}}(\mathbf{z}_m)$. In most practical applications, the number of negative samples M is set equal to the number

of positive samples, N , making the discriminator's binary classification problem balanced.

To solve the min-max problem, we need to find the gradients with respect to θ and ϕ of the objective function. For the discriminator, we need the gradient with respect to ϕ :

$$\nabla_{\phi} V(\theta, \phi) = \nabla_{\phi} \left(\sum_{n=1}^N \log D_{\phi}(\mathbf{x}_n) + \sum_{m=1}^M \log(1 - D_{\phi}(\mathbf{x}_m)) \right) \quad (8.5)$$

For the generator, the gradient with respect to θ is:

$$\begin{aligned} \nabla_{\theta} V(\theta, \phi) &= \nabla_{\theta} \left(\sum_{n=1}^N \log D_{\phi}(\mathbf{x}_n) + \sum_{m=1}^M \log(1 - D_{\phi}(\mathbf{x}_m)) \right) \\ &= \nabla_{\theta} \left(\sum_{m=1}^M \log(1 - D_{\phi}(\mathbf{x}_m)) \right) \end{aligned} \quad (8.6)$$

Once the gradients are computed, we update the parameters using stochastic gradient ascent in the discriminator, and stochastic gradient descent on the generator:

$$\begin{aligned} \phi &\leftarrow \phi + \alpha_1 \nabla_{\phi} V(\theta, \phi) \\ \theta &\leftarrow \theta - \alpha_2 \nabla_{\theta} V(\theta, \phi) \end{aligned} \quad (8.7)$$

where α_1 and α_2 are the learning rates for the discriminator and the generator, respectively.

An interesting aspect is the computation of the gradients for the generator. While the discriminator's gradient computation is straightforward, the generator's gradient

requires backpropagating through the discriminator. If we denote $\ell = \log(1 - D_\phi(G_\theta(\mathbf{z})))$, then:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} V(\boldsymbol{\theta}, \boldsymbol{\phi}) &= \nabla_{\boldsymbol{\theta}} \left(\sum_{m=1}^M \log(1 - D_\phi(\mathbf{x}_m)) \right) \\
&= \nabla_{\boldsymbol{\theta}} \left(\sum_{m=1}^M \log(1 - D_\phi(G_\theta(\mathbf{z}_m))) \right) \\
&= \left(\sum_{m=1}^M \nabla_{\boldsymbol{\theta}} \log(1 - D_\phi(G_\theta(\mathbf{z}_m))) \right) \\
&= \sum_{m=1}^M \nabla_{\boldsymbol{\theta}} \ell_m \\
&= \sum_{m=1}^M \frac{\partial \mathbf{x}_m}{\partial \boldsymbol{\theta}} \frac{\partial \ell_m}{\partial \mathbf{x}_m}
\end{aligned} \tag{8.8}$$

This indicates that we are backpropagating the gradients through the discriminator to the generator. In practice, we concatenate the generator and the discriminator as a compose function, $D_\phi(G_\theta(\mathbf{z}))$, and our favorite automatic differentiation tool will compute the entire gradient efficiently.

The typical training procedure for GANs involves the following steps:

1. Obtain a true dataset $\mathcal{D}_T = \{x_n\}$ and label each observation as real (positive label).
2. Initialize the Generator G_θ randomly.
3. Generate a fake dataset, $\mathcal{D}_F = \{x_m\}$, by sampling from the generator, $\mathbf{z}_m \sim p_{\mathbf{z}}$, $x_m = G_\theta(\mathbf{z}_m)$, and label each observation as fake (negative label).

4. Combine the true and fake datasets to create a combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$.
5. Optimize the discriminator $D_\phi(\mathbf{x}) = p(\mathbf{y}|\mathbf{x})$ using the combined dataset \mathcal{D} with one step of stochastic gradient ascent.
6. Optimize the generator D_θ using the feedback signal from the discriminator with one step of stochastic gradient descent.
7. Repeat steps 3 to 6 until convergence.

By alternative updating the discriminator and the generator, we train the GAN to improve the quality of the generated data, making it increasingly difficult for the discriminator to distinguish between real and fake data more difficult.

In [Figure 8.3](#), we show samples generated as illustrated in the original work by Goodfellow et al. (2014). At that time, the sample quality was at least competitive with the best generative models available, as noted by the authors. In [Figure 8.3](#), the rightmost column shows the nearest training example to the neighboring sample, demonstrating that the model has not memorized the training set. This is illustrated for the following datasets: (a) MNIST, (b) TFD, (c) CIFAR-10 (fully connected model), and (d) CIFAR-10 (convolutional discriminator and “deconvolutional” generator). For more details, see Goodfellow et al. (2014).

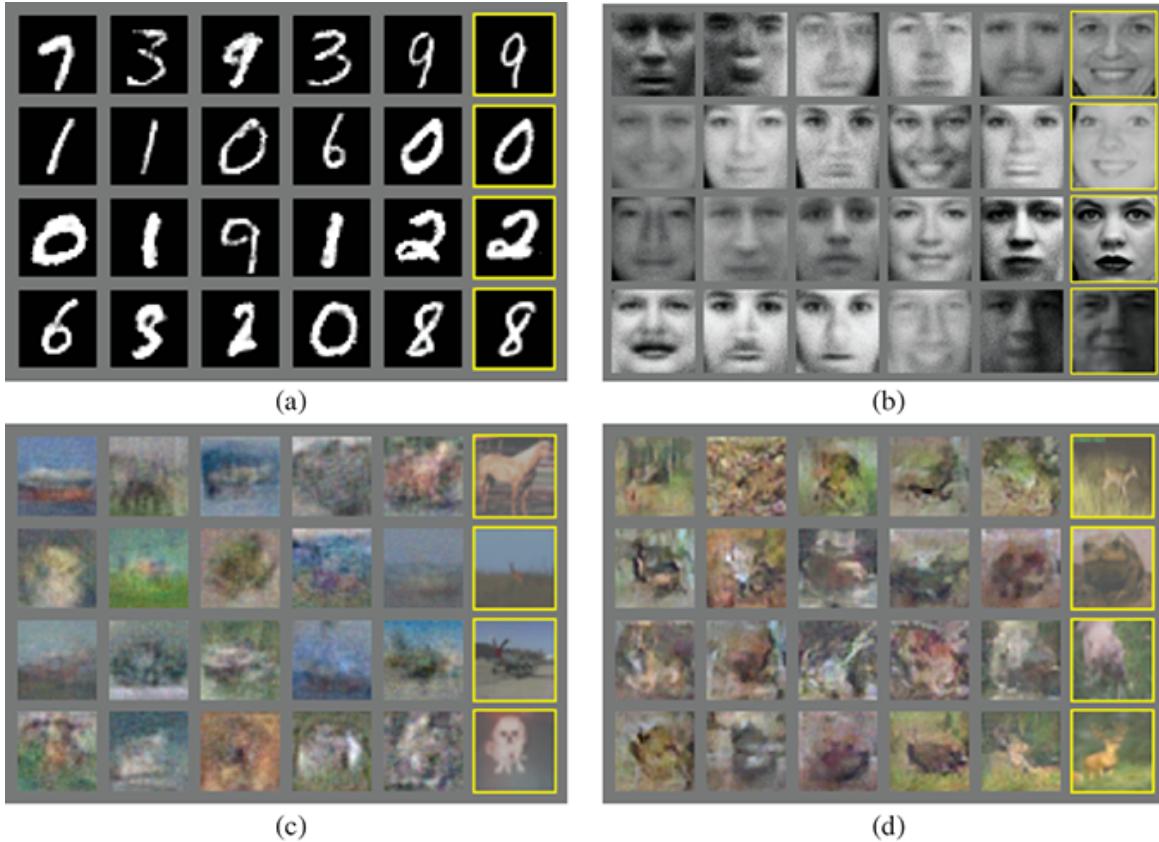


Figure 8.3 The rightmost column displays the nearest training example to each corresponding generated sample, demonstrating that the model has not memorized the training set. This is illustrated for the following datasets: (a) MNIST, (b) TFD, (c) CIFAR-10 (fully connected model), and (d) CIFAR-10 (convolutional discriminator and “deconvolutional” generator).

Source: I. J. Goodfellow et al. (2014).

8.2.1 Evaluation

The evaluation of GANs is still an open problem. Since we cannot compute the density explicitly, and the discriminator probability score is used to drive the generator to create data that are similar to the data distribution, this score for a given observation does not

provide much insight into the quality of the generated samples. This is especially important when we want to compare models. One initial solution was to use Parzen window-based log-likelihood estimates, also known as Kernel Density Estimation (KDE), as a replacement for likelihoods. More details about KDE can be found in Bishop (2006).

The [Figure 8.4](#) shows the Parzen window-based log-likelihood on the original work on GANs, allowing for a quantifiable model comparison rather than simply eyeballing the generated images. They reported numbers on MNIST and TFD datasets for different generated models.

Model	MNIST	TFD
DBN [3]	138 ± 2	1909 ± 66
Stacked CAE [3]	121 ± 1.6	2110 ± 50
Deep GSN [6]	214 ± 1.1	1890 ± 29
Adversarial nets	225 ± 2	2057 ± 26

Figure 8.4 Table 1 from Goodfellow et al. (2014): Parzen window-based log-likelihood estimates. The reported values for MNIST represent the mean log-likelihood of test set samples, with the standard error of the mean computed across individual examples.

More evaluation metrics have been proposed in the literature to better account for the coverage of the generated samples. One such metric is the Inception Score, see Salimans et al. (2016), which utilizes the Inception neural network to classify images and evaluate the quality of conditional generation with GANs. Improvements over the Inception Score include the Frechet Inception Distance (FID), which aims to improve the measurement of diversity by incorporating statistics of the images in a different

embedding space. More details about the FID distance can be found in Heusel et al. (2018).

8.3 Some Theoretical Insight in GANs

We have already seen that the training of GANs requires solving a min-max two-player game between the generator and the discriminator. From the objective function, the task for each model is clear: the discriminator maximizes the likelihood of the data, while the generator tries to fool the discriminator by minimizing the same objective function. However, since we are interested in the quality of the generated samples, we can ask ourselves the following question: Are GANs trained with the aforementioned procedure trying to optimize some kind of divergence between the data distribution p_{data} and the generator distribution p_G ?

To answer this, let's perform a theoretical exercise to solve for the optimal discriminator and generator and see if we can gain some insights. Although we do not train GANs using the exact procedure to be described here, this exercise provides valuable insights that have been used in practice to improve the training and performance of GANs.

Let's solve for the optimal classifier. Given our objective:

$$\begin{aligned} \min_{\theta} \max_{\phi} \quad & V(\theta, \phi) \\ \min_{\theta} \max_{\phi} \quad & \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_{\phi}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\log(1 - D_{\phi}(G_{\theta}(\mathbf{z})))] \end{aligned} \tag{8.9}$$

what can we say about G at convergence? Let's see if we can find a closed-form solution for D . We can rewrite [Equation 8.9](#) as follows:

$$\begin{aligned} \min_G \max_D & \mathbb{E}_{\mathbf{x} \sim p} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [\log(1 - D(G(\mathbf{z})))] \\ \min_G \max_D & \mathbb{E}_{\mathbf{x} \sim p} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim q} [\log(1 - D(\mathbf{x}))] \end{aligned} \quad (8.10)$$

Here, we simplify our notation and optimize over functions D and G instead of optimizing over parameters θ and ϕ . We also replace p_{data} by p , the distribution of the true dataset, and let q represent the distribution used to generate our fake dataset, with $D(\mathbf{x}) = D(G(\mathbf{z}))$.

Let's solve for the optimal discriminator D :

$$D^* = \arg \max_G \mathbb{E}_{\mathbf{x} \sim p} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim q} [\log(1 - D(\mathbf{x}))] \quad (8.11)$$

To find the optimal value of D , we compute the gradient with respect to D and set it equal to zero:

$$\begin{aligned} \nabla_D &= \mathbb{E}_{\mathbf{x} \sim p} [\nabla_D \log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim q} [\nabla_D \log(1 - D(\mathbf{x}))] \quad (8.12) \\ &= \mathbb{E}_{\mathbf{x} \sim p} \left[\frac{1}{D(\mathbf{x})} \right] - \mathbb{E}_{\mathbf{x} \sim q} \left[\frac{1}{1 - D(\mathbf{x})} \right] \end{aligned}$$

Setting $\nabla_D = 0$ and solving for D , we find that optimal value is:

$$\begin{aligned} D^* &= \frac{p(x)}{p(x) + q(x)} \quad (8.13) \\ D^* &= \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} \end{aligned}$$

where q is replaced by p_G , to indicate that the negative labels are generated from the generator distribution, p_G .

Now, if we plug the optimal value of D into the objective function, we can determine what the generator G is trying to optimize:

(8.14)

$$\begin{aligned}
V(G, D^*) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_G} [\log(1 - D(\mathbf{x}))] \\
&= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} \right] + \mathbb{E}_{\mathbf{z} \sim p_G} \left[\log \frac{p_G(x)}{p_{\text{data}}(x) + p_G(x)} \right] \\
&= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log p_{\text{data}}(x) - \log(p_{\text{data}}(x) + p_G(x))] + \\
&\quad \mathbb{E}_{\mathbf{x} \sim p_G} [\log p_G(x) - \log(p_{\text{data}}(x) + p_G(x))]
\end{aligned}$$

Let $q(x) = (p_{\text{data}}(x) + p_G(x))/2$. Then

(8.15)

$$\begin{aligned}
V(G, D^*) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log p_{\text{data}}(x) - \log q(x)] + \mathbb{E}_{\mathbf{x} \sim p_G} [\log p_G(x) - \log q(x)] - \log 4 \\
&= D_{\text{KL}}(p_{\text{data}} \| p_G) + D_{\text{KL}}(p_G \| p_{\text{data}}) - \log 4 \\
&= 2D_{\text{JS}}(p_{\text{data}} \| p_G) - \log 4
\end{aligned}$$

where D_{JS} is the Jensen-Shannon Divergence.

Thus, at convergence, the generator minimizes a notion of distance between the data distribution p_{data} and the generator distribution p_G . This insight has led to modern approaches that address both the drawbacks and advantages of minimizing this divergence, proposing better alternatives for training GANs.

8.4 Why Is GAN Training Hard? Improving GAN Training Techniques

Training GANs in practice is challenging. Implementing GANs as described in the original work by Goodfellow et al. (2014) requires an enormous amount of hyperparameter tuning to achieve convergence and generate high-quality samples.

For example, the paper by Salimans et al. (2016) illustrates the extensive hyperparameter tuning required and some of

the techniques used to make GANs work effectively. These techniques include minibatch discrimination, feature matching, historical averaging, and others, which are applicable to either the generator, the discriminator, or both. For more details, see Salimans et al. (2016).

One reason why training GANs is difficult can be understood by examining a simple example, illustrated in [Figure 8.5](#). For the sake of simplicity, we generate samples from both the data distribution and the generator, overlaying the corresponding density functions and assuming this is the initial data for training. At initialization, with a random generator, the densities and samples generated from p_{data} and p_G are far apart. If we train the discriminator with the combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$ using the true data \mathcal{D}_T and the fake data \mathcal{D}_F from these samples, the discriminator can easily achieve perfect classification. This results in a very sharp decision boundary, saturated at 1 for samples from p_{data} and 0 for samples generated by p_G .

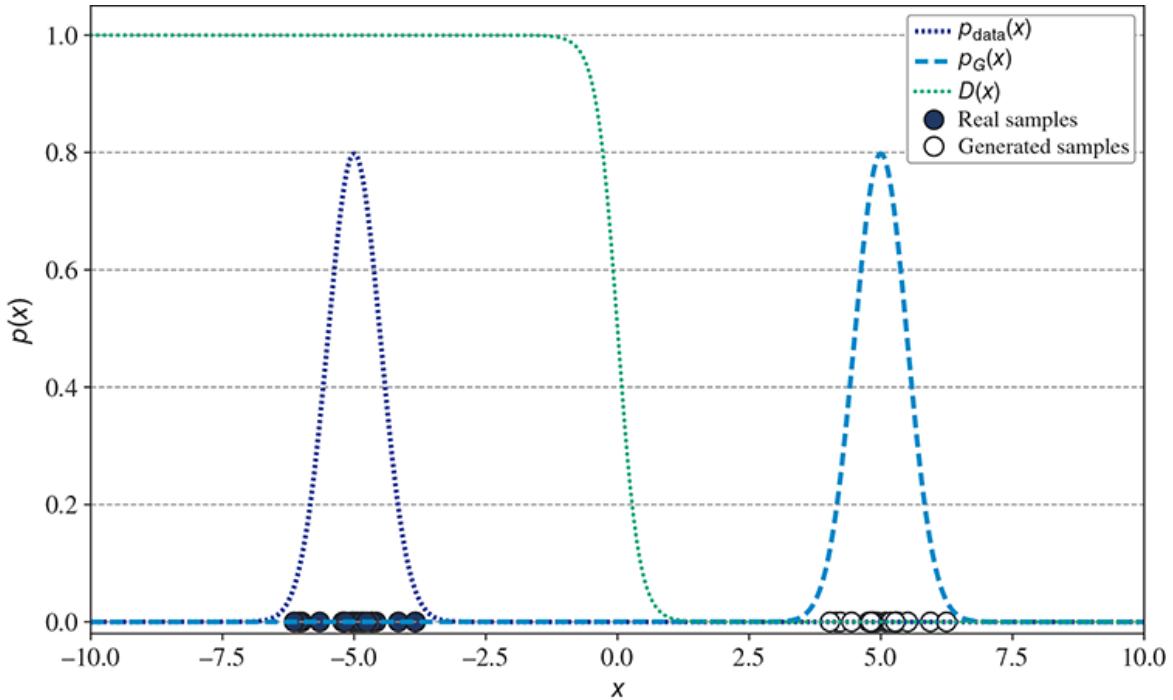


Figure 8.5 Illustration of one reason why training GANs can be difficult. In this conceptual illustration, at initialization, the data distribution and the generator distribution are far apart, allowing the discriminator to achieve perfect classification. This results in a sharp decision boundary, where the discriminator outputs 1 for real samples and 0 for generated samples, making it challenging for the generator to receive meaningful gradient updates.

As the generator learns by backpropagating the complete gradients of the discriminator, in situations where the discriminator is saturated, it provides very poor gradient signals to the generator for almost all the points that are generated from p_G . This makes it almost impossible for the generator to improve the quality of the generated samples. In short, one of the main reasons why GAN training is hard is that the discriminator might provide a very weak gradient signal, if the data distribution and the generator distribution are very far apart.

To improve this situation where p_{data} and p_G are very far apart, we aim to have a smoother decision boundary for the discriminator. This would provide meaningful signals to drive the training of the generator and improve the quality of the generated samples. Solutions around this point include explicitly modifying the discriminator to have a smoother decision boundary or making the samples from p_{data} and p_G overlap, making it more difficult for the discriminator to classify them. Instance noise, as described by Mescheder et al. (2018), is a method where noise is added to both real and fake samples to make their distributions overlap, making the classification task harder for the discriminator and resulting in a smoother decision boundary.

Techniques for explicitly modifying the decision boundary include Least Squares Generative Adversarial Networks (LSGANs), as described by Mao et al. (2017), where the discriminator outputs an unbounded real number instead of a probability, a real number between 0 and 1. Other solutions involve using different objective functions to force the discriminator to have a smoother decision boundary. The main solution proposed in Wasserstein GANs (WGANs), as described in Arjovsky et al. (2017), constrains the discriminator to be Lipschitz-continuous. Further solutions for making the decision boundary even smoother include WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017) and Spectral Normalization GAN (SNGAN) by Miyato et al. (2018).

For detailed analysis of convergence and different training techniques for GANs, refer to the paper by Mescheder et al. (2018).

8.5 Wasserstein GAN (WGAN)

From [Section 8.3](#), we got the intuition that the generator is trying to minimize the Jensen-Shannon (JS) divergence between the data distribution and the generator distribution. However, the JS divergence is not effective at capturing the distance between distributions when they are very far apart or have little to no overlap.

To illustrate this, consider [Figure 8.6](#). In the top figure, the distributions p_{data} and p_G are very far apart with zero overlap. In the bottom figure, p_{data} and p_G are closer but still do not overlap. Ideally, we want to drive the generator distribution p_G closer to p_{data} , and between these two options, we would prefer the generator from the bottom figure. However, the JS divergence assigns the same score to both situations since there is no overlap between the two distributions, which poses a problem for training the generator effectively.

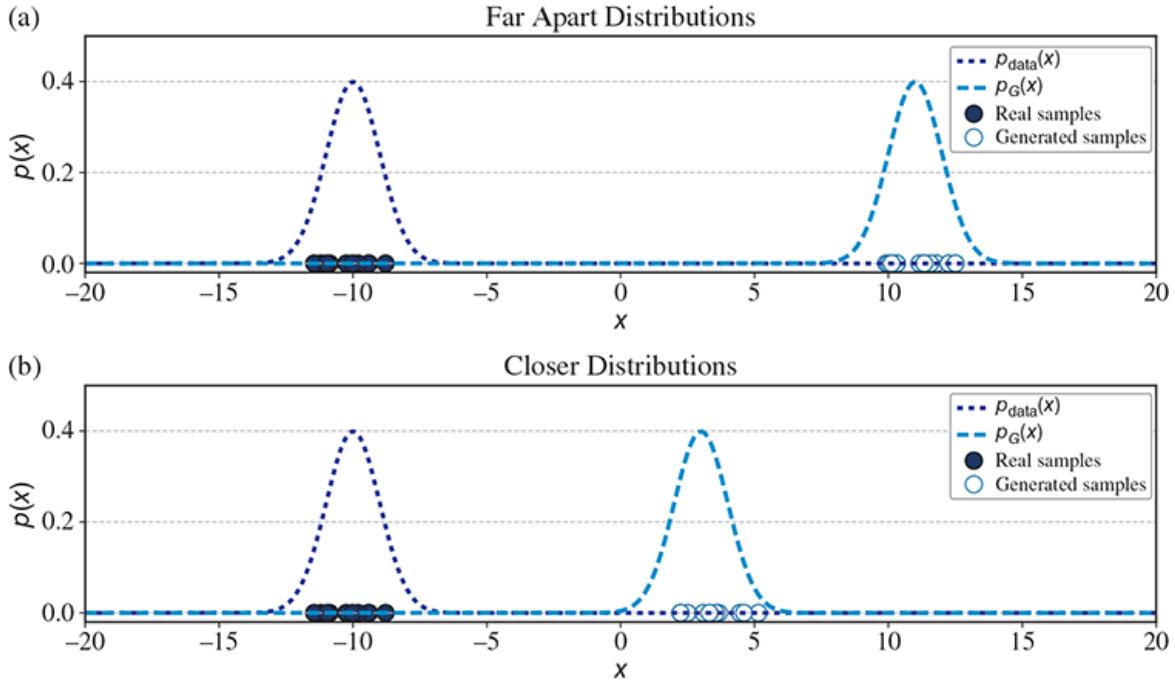


Figure 8.6 Illustration of two different cases for the data distribution and the generator distribution. In Figure (a), the distributions are far apart, whereas in Figure (b), they are closer. Despite this difference, the JS divergence assigns the same score to both cases.

An improvement to this situation could be to use a different divergence measure that actually quantifies the distance between two distributions. One candidate is the Wasserstein distance, which is used to optimize GANs as proposed by Arjovsky et al. (2017).

The Wasserstein distance is defined as:

$$W(p_{\text{data}}, p_G) = \inf_{\gamma} \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim \gamma} \|\mathbf{x} - \mathbf{y}\| \quad (8.16)$$

where γ is the distribution over \mathbf{x} and \mathbf{y} that satisfies the marginals $\gamma_{\mathbf{x}} = p_{\text{data}}$ and $\gamma_{\mathbf{y}} = p_G$.

Directly optimizing this formulation of $W(p_{\text{data}}, p_G)$ is complicated because finding the optimal γ is not an easy

task. However, there is a dual formulation known as the Kantorovich-Rubinstein duality, which makes the problem more tractable:

$$W(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} f(\mathbf{x}) - \mathbb{E}_{\mathbf{x} \sim p_G} f(\mathbf{x}) \quad (8.17)$$

This is equal to the supremum over the set of all possible 1-Lipschitz functions f , of the expected value of f under p_{data} minus the expected value of f under p_G . The set of all possible 1-Lipschitz scalar functions f satisfies:

$$|f(x) - f(y)| \leq |x - y| \quad (8.18)$$

which means roughly speaking that f should have a bounded gradient or slope.

Returning to our example, if f_ϕ is our discriminator neural network parameterized by ϕ , how can we constrain the network to be 1-Lipschitz, or equivalently, to have a bounded gradient? The original solution proposed by Arjovsky et al. (2017) involves clipping the weights of the network to ensure the gradient is bounded. However, the paper's authors clearly mentioned that weight clipping is probably not the best way to enforce this constraint and illustrated many problems with it. In fact, at the time of writing, gradient clipping for achieving this constraint is mostly obsolete. Other works build on this idea to enforce a Lipschitz-continuous neural network, such as WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017) and Spectral Normalization GAN (SNGAN) by Miyato et al. (2018).

Once we constrain the search space of functions to the set of 1-Lipschitz functions, we can use the sample-based objective to find the optimal discriminator and generator:

$$\begin{aligned}
W(p_{\text{data}}, p_G) &= \sup_{\|f_\phi\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [f_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{x} \sim p_G} [f_\phi(\mathbf{x})] \\
&= \sup_{\|f_\phi\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [f_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}} [f_\phi(G_\theta(\mathbf{z}))]
\end{aligned} \tag{8.19}$$

We approximate $W(p_{\text{data}}, p_G)$ using sample-based estimators:

$$\hat{W}(p_{\text{data}}, p_G) = \sup_{\|f_\phi\|_L \leq 1} \sum_{n=1}^N f_\phi(\mathbf{x}_n) - \sum_{m=1}^M f_\phi(G_\theta(\mathbf{z}_m)) \tag{8.20}$$

We can then optimize by taking gradients with respect to ϕ and θ .

The pseudo code for training WGANs is as follows (adapted from Levine (2021)):

1. Obtain a true dataset $\mathcal{D}_T = \{\mathbf{x}_n\}$ and label each observation as real (positive label).
2. Initialize the Generator G_θ randomly.
3. Generate a fake dataset, $\mathcal{D}_F = \{\mathbf{x}_m\}$, by *sampling from the generator*, $\mathbf{z}_m \sim p_{\mathbf{z}}$, $x_m = G_\theta(\mathbf{z}_m)$, and label each observation as fake (negative label).
4. Combine the true and fake datasets to create a combined dataset $\mathcal{D} = \mathcal{D}_T \cup \mathcal{D}_F$.
5. Update the discriminator f_ϕ to maximize $\hat{W}(p_{\text{data}}, p_G)$. Compute $\nabla_\phi \hat{W}(p_{\text{data}}, p_G)$ and use stochastic gradient ascent.
6. Clip all weights matrices in ϕ such that each weight matrix layer, denoted by ϕ_l , is within $[-c, c]$ for some constant c .

7. Update the generator G_{θ} to minimize $\widehat{W}(p_{\text{data}}, p_G)$.

Compute the gradients

$\nabla_{\theta} \widehat{W}(p_{\text{data}}, p_G) = \nabla_{\theta} \sum_{m=1}^M f_{\phi}(G_{\theta}(\mathbf{z}_m))$ and use stochastic gradient descent.

8. Repeat steps 3 to 7 until convergence.

In [Figure 8.7](#), we see the results from Figure 2 in the paper by Arjovsky et al. (2017). This figure illustrates the effect of the Wasserstein distance on optimizing GANs, particularly highlighting the decision boundary obtained by the discriminator. In [Figure 8.7](#), the decision boundary of the discriminator for typical vanilla GANs is labeled “GAN Discriminator” in red, optimized using the two-player min-max game. The decision boundary for the WGAN, labeled “WGAN Critic” in cyan, shows a much smoother boundary, providing better gradient signals for the generator and improving the training process.

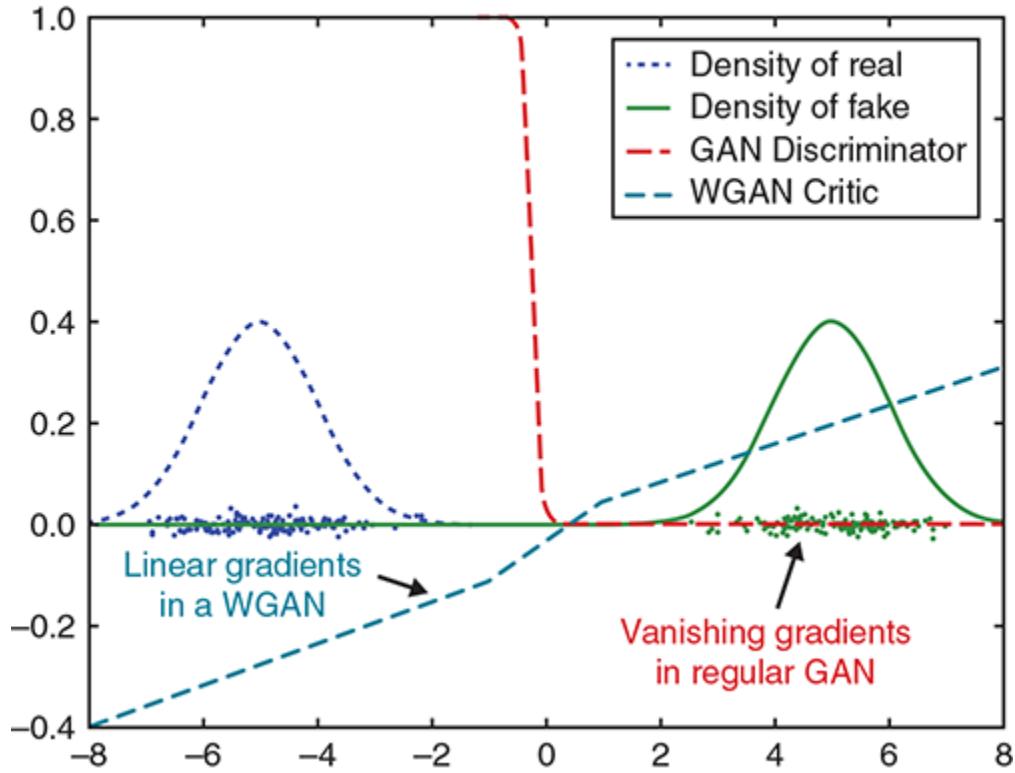


Figure 8.7 Illustration of the optimal discriminator and critic when distinguishing two Gaussians. The minimax GAN discriminator (red) saturates, leading to vanishing gradients, while the WGAN critic (cyan) provides smoother gradients, improving generator training. Figure 2 from Arjovsky, Chintala, and Bottou (2017).

8.5.1 Gradient Penalty GAN (WGAN-GP)

As mentioned, another way to enforce Lipschitz constraints in a neural network is to directly constrain the gradient, which is the approach proposed in WGAN with Gradient Penalty (WGAN-GP) by Gulrajani et al. (2017).

In their work, they modified the original WGAN objective by introducing a gradient penalty for Lipschitz continuity, given by:

$$\begin{aligned}
W(p_{\text{data}}, p_G) &= \sup_{\|f_\phi\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [f_\phi(\mathbf{x})] - \mathbb{E}_{\mathbf{x} \sim p_G} [f_\phi(\mathbf{x})] \\
&\quad - \lambda \mathbb{E}_{\hat{\mathbf{x}} \sim p_{\hat{\mathbf{x}}}} \left[(\|\nabla_{\hat{\mathbf{x}}} f_\phi(\hat{\mathbf{x}})\|_2 - 1)^2 \right]
\end{aligned} \tag{8.21}$$

where the gradient penalty term is the last term of the previous equation. Here, the objective function directly penalizes the discriminator for having gradients with norms different from one, using the regularization parameter λ . The expectation is computed under the distribution $p_{\hat{\mathbf{x}}}$, for which samples are generated by sampling uniformly along straight lines between pairs of points sampled from the data distribution p_{data} and the generator distribution p_G . For more details, see Gulrajani et al. (2017). In practice, WGAN-GP improves the stability of WGAN training and is very easy to implement.

A detailed analysis of the convergence of different GAN training algorithms can be found in Mescheder et al. (2018).

8.6 Extending GANs for Time Series

Similarly, as we have seen in previous chapters with models like VAEs, and Flows, these models, by their structural design, assume independence between consecutive observations. Then, the natural questions arise, how can we adapt them to capture this time-series dynamics that might be important at certain time scales?

One potential answer is similar to the way we have done it previous chapters. We could try to combine GANs with models that excel at handling the kind of dynamics we want to capture, such as autoregressive models like RNN or Transformers.

Let's consider two interesting cases for time-series generation.

- Recurrent Conditional GAN (RCGAN), see Esteban et al. (2017): Uses a combination of RNN with GAN. It uses an RNN for the generator and discriminators. Due to this design choice, data are generated sequentially. More details about model architecture and experimental results are in [*RCGAN Notebook*].
- Time-series Generative Adversarial Networks (TimeGAN), see Yoon et al. (2019): TimeGAN employs a sequence generator and discriminator to model time-series dynamics. However, it does that by modeling the temporal dependencies in the latent space. They do that by combining supervised learning objectives as well as generative adversarial objectives in a jointly optimization problem. The supervised learning objective drives the network to capture the dynamics in the latent space. The adversarial objective, like in standard GANs, drives the network to generate realistic samples. More details about model architecture and experimental results are in [*TimeGAN Notebook*].

8.7 Conclusion

In this chapter, we explored another powerful tool for modeling high-dimensional data—GANs. Unlike the models studied in previous chapters, GANs belong to the class of implicit generative models, meaning they do not require explicit density estimation. Instead, they rely on an adversarial learning process in which two models—the generator and the discriminator—are trained simultaneously in an adversarial framework. Unlike flow models, the GAN generator can be any neural network,

with no constraints on invertibility or matching dimensionality between the latent variable and observations. This flexibility allows GANs to learn complex representations and generate high-quality samples. We also discussed key challenges in training GANs and explored strategies to improve their stability and performance.

Additionally, we examined how GANs can be extended to model sequential data, particularly time series. By integrating GANs with architectures specialized for capturing temporal dependencies—such as RNNs—we can generate sequential data with time-dependent dynamics. In the notebooks associated with this chapter, you will find practical examples of both traditional GANs and time-series-adapted GANs applied to financial data generation.

With this chapter, you have added yet another powerful tool to your generative modeling toolbox. GANs provide a flexible and effective approach for generating high-dimensional, multimodal data.

Chapter 9

Leveraging LLMs for Sentiment Analysis in Trading

In this chapter, we pivot from our previous discussions, where we focused on studying different generative model families in isolation, to explore and build a practical financial application that combines these models. The goal is to create a system that takes speech data as input and outputs trading signals. This system leverages powerful models, such as transformer models for sequence-to-sequence tasks to convert audio signals to text, and transformer models in Large Language Models (LLMs) to assign sentiment scores to text. We will apply these techniques to analyze the sentiment of Federal Reserve (Fed) press conference speeches and generate potentially profitable trading signals.

9.1 Sentiment Analysis in Fed Press Conference Speeches Using Large Language Models

Is it possible to use sentiment analysis on Federal Reserve press conference speeches, computed using LLMs, to generate predictive trading signals?

Let's imagine a scenario where we are watching US Federal Reserve Chair Jerome Powell hold a press conference on November 2, 2022, where he is discussing the latest Fed rate hike and communicating the committee's current economic outlook, as shown in [Figure 9.1](#) (WGN News, 2022).



Figure 9.1 Fed Chair Powell discusses latest Fed rate hike.

Source: WGN News (2022).

As traders, we might wonder how to use the information provided in such a speech to make informed trading decisions. If you were someone like George Soros, known for his exceptional trading skills and remarkable track record in macro trading, you might be able to filter the relevant information from the speech, converting it into insights that you can combine with other data, such as market data, to predict the market's next moves easily.

However, for those of us without Soros' insights, a more quantitative approach could be helpful. This approach requires converting segments of the video into numerical values from which we can generate buy or sell signals. Our approach has been inspired by reports on JPMorgan AI's analysis of Fed speeches for trading signals from *Bloomberg*

News (2023) and *Fortune* (2023), as well as an OpenAI Whisper tutorial on Fed Speech Transcription from Part Time Larry (2022).

There are several ways to design a system with the goal of translating video content into actionable trading signals. Here, we attempt to provide a solution based on a cascade approach that involves a series of subsystems for processing different datatypes. The whole system takes a segment of the Fed press conference video as input and outputs a numerical value representing the sentiment score of that segment. The design of these subsystems will involve the following:

1. Extracting speech from the video of the Fed press conference
2. Converting speech segments to text
3. Converting text to numerical values (sentiment scores) that can generate relevant signals

The system's block diagram illustrating this process is in [Figure 9.2.](#)

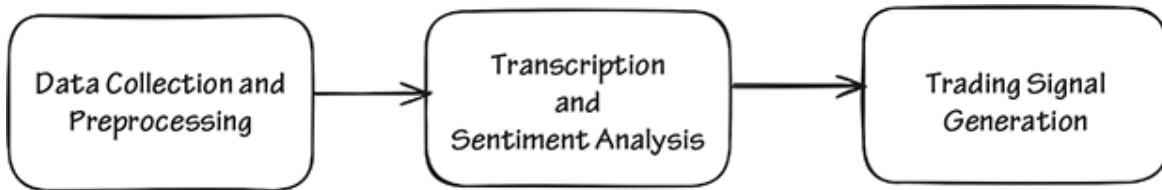


Figure 9.2 System block diagram.

To explore this, we will take a step back and simulate making decisions based on the press conference and related market data available to us.

Let's say we want to design a trading strategy to trade the SPY in 30-second intervals. In [Figure 9.3](#), we illustrate the SPY price series during the November 2, 2022, Fed press conference, from 14:30 to 15:30 in 30-second intervals. Ideally, we would like to find some kind of correlation

between what Jerome Powell is saying at or up to that particular moment and the subsequent SPY price moves.

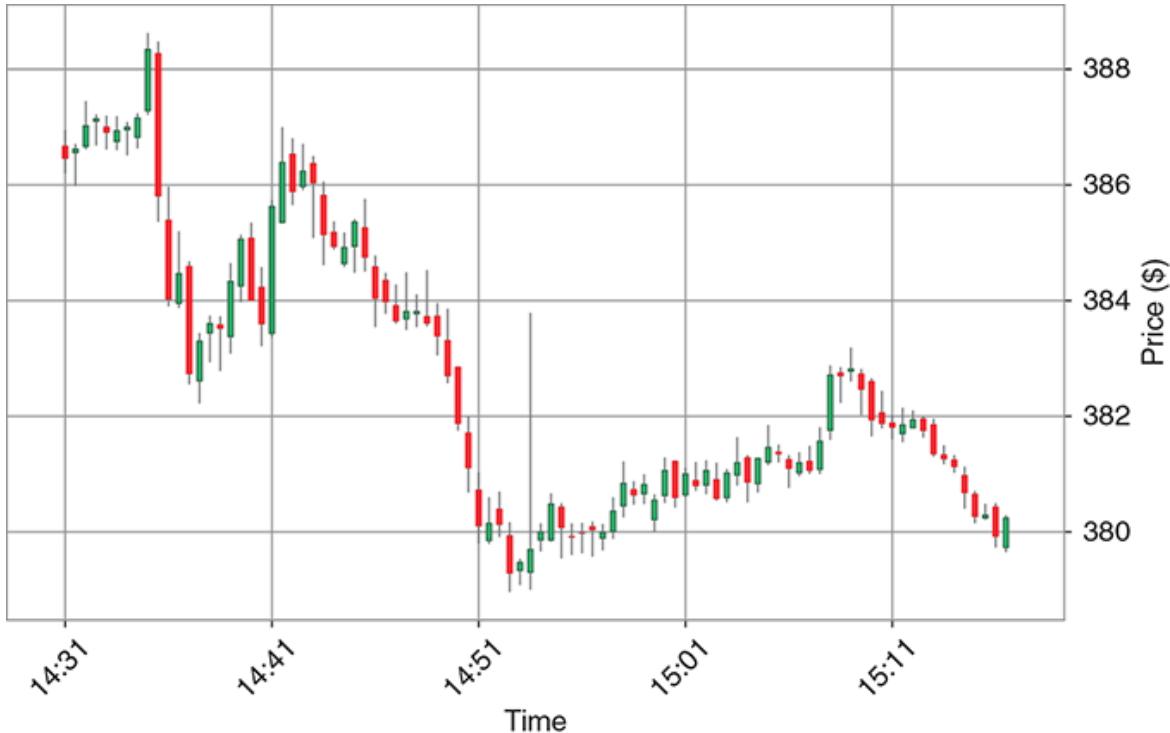


Figure 9.3 SPY Price series during Fed press conference.

For every 30-second trading bar of SPY data, we would do the following:

1. Extract audio from the video up to that particular bar.
2. Perform speech-to-text conversion.
3. Perform Sentiment Analysis based on text.
4. Generate signals to make buy and sell decisions.

This process will enrich the SPY open-high-low-close (OHLC) price series with the sentiment scores of Powell's speech during each 30-second interval, making it easier to quantify the predictive power of the signal. The desired tabular data format is illustrated in [Figure 9.4](#).

timestamp		text	video_id	sentiment_score	cst_timestamp	open	high	low	close	return
2022-06-15 14:31:00-04:00		Good afternoon. I will begin with one overarching message. We at the Fed unders...	1	0.116885	2022-06-15 13:31:00-05:00	374.40	374.57	373.65	373.94	0.000053
2022-06-15 14:31:30-04:00		Good afternoon. I will begin with one overarching message. We at the Fed unders...	1	0.360220	2022-06-15 13:31:30-05:00	373.80	374.15	373.47	373.96	-0.001685
2022-06-15 14:32:00-04:00		one overarching message. We at the Fed unders...	1	0.313680	2022-06-15 13:32:00-05:00	374.05	374.07	373.00	373.33	-0.001848
2022-06-15 14:32:30-04:00		if we were to have a sustained period of stro...	1	0.115033	2022-06-15 13:32:30-05:00	373.18	373.30	372.55	372.64	-0.000510
2022-06-15 14:33:00-04:00		rate will be appropriate. In addition, we are...	1	0.034392	2022-06-15 13:33:00-05:00	372.49	372.87	372.12	372.45	0.002121
2022-06-15 14:33:30-04:00		with consumption spending remaining strong. I...	1	-0.311607	2022-06-15 13:33:30-05:00	372.41	373.50	372.30	373.24	0.000831
2022-06-15 14:34:00-04:00		have marked down their projections for econom...	1	-0.186132	2022-06-15 13:34:00-05:00	373.06	373.61	372.90	373.55	0.000428
2022-06-15 14:34:30-04:00		robust. Improvements in labor market conditio...	1	0.522128	2022-06-15 13:34:30-05:00	373.59	374.35	373.45	373.71	-0.000722
2022-06-15 14:35:00-04:00		on wages and prices. The median projection in...	1	0.380036	2022-06-15 13:35:00-05:00	373.69	374.01	373.25	373.44	0.001446
2022-06-15 14:35:30-04:00		or prices rose 4.9%. In May, the 12-month cha...	1	-0.127001	2022-06-15 13:35:30-05:00	373.60	374.04	373.34	373.98	0.001230

Figure 9.4 Enriched price series.

With this enriched data, we can investigate whether there is a statistical relationship between the computed sentiment scores and future returns. In [Figure 9.5](#), we illustrate a scatter plot for a few Fed press conferences videos that took place in 2022 and 2023, where the x-axis represents the sentiment scores and the y-axis represents the SPY forward returns over the next 30 seconds.

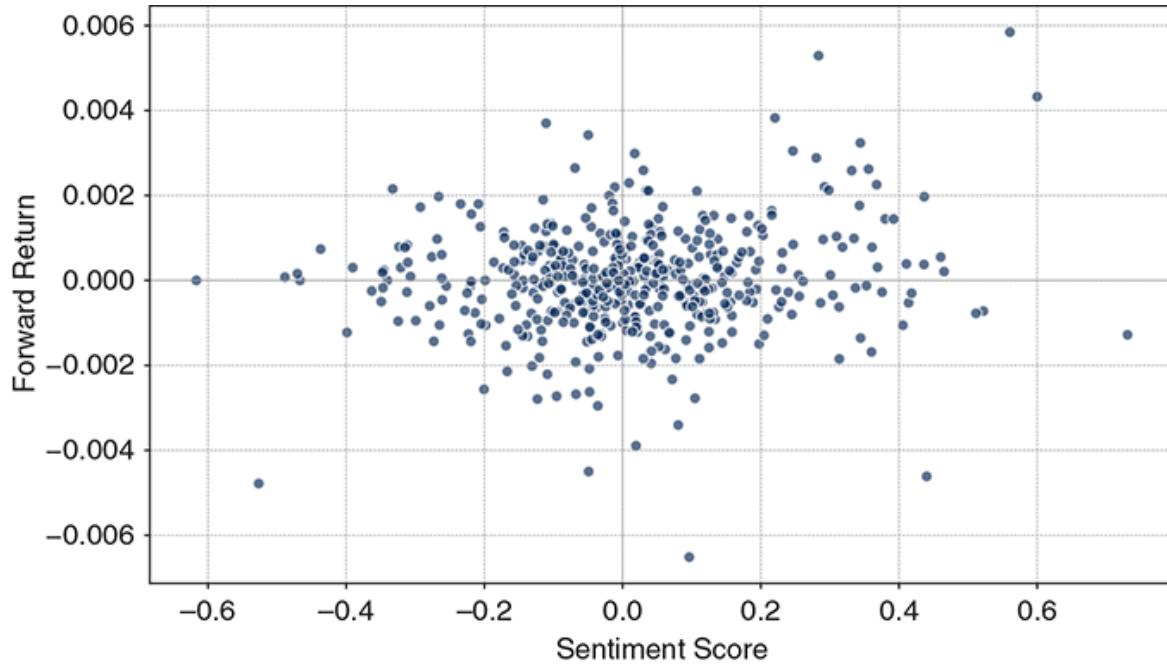


Figure 9.5 Scatter plot of sentiment signal vs forward returns.

In the rest of the section, we will describe in more detail the main components of the system, how they work, and how to put them together. In our experience, using a limited sample size of videos, we found a significant Pearson correlation of 14.14% between the sentiment signal and forward SPY return, with a p -value of 0.199%.

9.2 Data: Video + Market Prices

In this use case, we utilize video data, extract the corresponding audio, process it into text, converting the text into numerical values, and finally joining these numerical values with market price data for model construction and evaluation. This approach leverages the unique insights offered by alternative data sources to enhance trading strategies and market predictions.

As noted by López de Prado (2018):

Alternative data offers the opportunity to work with truly unique, hard-to-process datasets. Remember, data that is hard to store, manipulate, and operate is always the most promising. You will recognize that a dataset may be useful if it annoys your data infrastructure team. Perhaps your competitors did not try to use it for logistic reasons, gave up midway, or processed it incorrectly.

9.2.1 Collecting Audio Data

For collecting audio data from YouTube, we use the [pytube](#) tool, which is a lightweight, Pythonic, and command-line utility for downloading YouTube Videos.

Downloading YouTube videos or audios can be easily achieved using pytube. Once the url of the desired video is defined, you can use the library as shown below:

```
from pytube import YouTube
video_url = 'http://youtube.com/watch?v=9bZkp7q19f0'
yt = YouTube(video_url)
yt.streams
    .filter(progressive=True, file_extension='mp4')
    .order_by('resolution')
    .desc()
    .first()
    .download()
```

where in the code snippet specified:

- We first import the pytube library.
- We specify the video url `video_url`.
- We create a YouTube object.
- We then filter mp4 file streams, order them by resolution, and download the highest resolution file.

9.3 Speech-to-text Conversion

9.3.1 Whisper Model

“We’ve trained and are open-sourcing a neural net called Whisper that approaches human level robustness and accuracy on English speech recognition.” —Introducing Whisper (<https://openai.com/index/whisper/>)

Whisper, Radford et al. (2022), is an automatic speech recognition (ASR) system trained on 680,000 hours of multilingual and multitask supervised data collected from the web. This large-scale dataset enables Whisper to transcribe in multiple languages and translate from multiple languages to English. OpenAI has open-sourced Whisper under the MIT License. According to the paper results, Whisper is comparable to standard benchmark and often competitive with models that have been finetuned for specific datasets.

In their paper, OpenAI describes various methods for evaluating and collecting data. The data collected for Whisper is weakly supervised, meaning it is not annotated by professional transcribers but instead has labels collected from the web. To build a dataset for model training consisting of audio paired with transcripts, the authors performed a great deal of data engineering work, including multiple preprocessing and heuristic filtering steps for tasks like input language detection and transcript alignment, which are detailed in the paper. Due to the vast amount of data collected, the dataset is highly diverse, encompassing a wide range of speakers, languages, and environments. More details about the Whisper architecture can be found in Radford et al. (2022).

Whisper processes the input audio by splitting it into 30-second chunks and converting it into a log-magnitude Mel spectrogram representation, which serves as the input to the

encoder. The decoder is trained to predict the corresponding text caption, intermixed with special tokens that direct the model to perform tasks such as language identification, phrase-level timestamps, multilingual speech transcription, and translation to English (see “Introducing Whisper” or the paper for more details).

According to the Whisper model card available in the official OpenAI GitHub repository for Whisper

(<https://github.com/openai/whisper>), it comes in five model sizes, offering different speed and accuracy trade-offs. Four of these models are English-only versions. [Figure 9.6](#) lists the names of the available models and their approximate memory requirements and inference speed relative to the large model.

Size	Parameters	English-only model	Multilingual model	Required VRAM	Relative speed
tiny	39 M	tiny.en	tiny	~1 GB	~32x
base	74 M	base.en	base	~1 GB	~16x
small	244 M	small.en	small	~2 GB	~6x
medium	769 M	medium.en	medium	~5 GB	~2x
large	1550 M	N/A	large	~10 GB	1x

Figure 9.6 Available models and languages.

According to OpenAI experiments, the .en models for English-only applications tend to perform better, especially for the tiny.en and base.en models. See their [GitHub](#) cited earlier for more details.

9.3.1.1 Python Usage.

We perform transcription using the forementioned Python Whisper library. Following is an example of how to transcribe audio using Whisper:

```
import whisper
audio_input = "audio.mp3"
model_name = "base"
model = whisper.load_model(model_name)
result = model.transcribe(audio_input)
print(result["text"])
```

In this example:

- We first import the Whisper library.
- We specify the audio input file `audio.mp3` and the model we want to use `base`.
- We load the specified Whisper model using `whisper.load_model(model_name)`.
- We transcribe the audio file using `model.transcribe(audio_input)`.
- Finally, we print the transcribed text using `print(result["text"])`.

Internally, the `transcribe` method reads the entire file and processes the audio with a sliding 30-second window, performing autoregressive sequence-to-sequence predictions on each window.

9.3.2 Whisper on FED Speech Audio Data

As we have seen, whisper models are trained on 30-second input chunks and cannot process longer audio input at once. This limitation can pose challenges for real-world applications. The authors of the Whisper paper propose a solution for transcribing longer audio sequences by consecutively transcribing 30-second audio windows and shifting this window according to the timestamps predicted by the model. They also emphasize the importance of making

correct decisions about decoding heuristics to reliably transcribe long audio, as detailed in [section 4.5](#) of their paper, Radford et al. (2022).

This has several implications for our use case. Fed speech conferences typically last between 45 and 60 minutes. Using such long audio inputs means that the timestamps predicted by Whisper might not be accurate, which could potentially introduce data alignment and look-ahead bias issues when merging data by timestamp between audio and price data—and important issue when backtesting trading strategies. The unreliability of the predicted timestamps for long audio files is something we have also observed during our experimentation.

To ensure the input audio signals are correctly aligned with the corresponding market time-series data, we avoid relying on the transcription timestamps produced by Whisper, whose accuracy decreases as the audio length increases. Instead, we perform audio segmentation (described in [Section 9.3.3](#)) beforehand, creating 60-second input chunks sampled every 30 seconds. This process generates 60-second “audio bars” every 30 seconds. This method allows us to determine the start and end timestamps of the corresponding audio, and therefore its transcript, without relying on Whisper for this purpose. Additionally, we introduced an embargo period in the audio series. This means joining price data timestamps with audio/transcribe timestamps 15 seconds earlier, thereby reducing the risk of look-ahead bias even further. While all these design choices limit the amount of information used to compute the sentiment score in subsequent subsystems, they significantly reduce the risk of introducing look-ahead bias.

9.3.3 Audio Segmentation

For the forementioned audio segmentation, we use FFmpeg (<https://ffmpeg.org>), which is cross-platform solution to

record, convert and stream audio and video.

You can use ffmpeg-python: Python bindings for FFmpeg (<https://github.com/kkroening/ffmpeg-python>), which are Python wrappers for FFmpeg. For example, in the code snippet below we can see how to convert between different audio formats:

```
import ffmpeg

# Define input and output files
input_file = 'input_video.mp4'
output_file = 'output_audio.wav'

# Extract audio from video
ffmpeg.input(input_file).output(output_file).run()
```

For audio segmentation, you can specify the start time and duration. This allows you to create specific “audio bars” of defined lengths, containing pure alternative data. These can then be combined with market data to create enriched datasets for model training and evaluation.

```
# Define start time and duration in seconds
start_time = 30
duration = 60

# Extract a specific segment
ffmpeg.input(input_file, ss=start_time, t=duration).output(output_file).run()
```

Results on actual FED data are shown in [Figure 9.7](#).

```
# load the whisper model
model = whisper.load_model(WHISPER_MODEL_NAME)

100% |██████████| 139M/139M [00:01<00:00, 74.6MiB/s]

If you are running locally with a recent model GPU, the following cell may give the message that the installed version of PyTorch does not support the CUDA level of your GPU. You can fix the problem by installing the appropriate version of PyTorch using these instructions: https://pytorch.org/get-started/locally/

transcript = transcribe(model, cut_details, seed=MAIN_SEED)
print(transcript.head())

started at 2023-09-29 14:57:46.030204
  0% |          0/105 [00:00<?, ?it/s]
ended at 2023-09-29 15:03:34.321123
time elapsed: 0:05:48.290919

text timestamp
0 Good afternoon. I will begin with 2022-06-15 13:31:00
1 Good afternoon. I will begin with one overarc... 2022-06-15 13:31:30
2 one overarching message. We at the Fed unders... 2022-06-15 13:32:00
3 if we were to have a sustained period of stro... 2022-06-15 13:32:30
4 rate will be appropriate. In addition, we are... 2022-06-15 13:33:00

transcript.head()

text timestamp
0 Good afternoon. I will begin with 2022-06-15 13:31:00
1 Good afternoon. I will begin with one overarc... 2022-06-15 13:31:30
2 one overarching message. We at the Fed unders... 2022-06-15 13:32:00
3 if we were to have a sustained period of stro... 2022-06-15 13:32:30
4 rate will be appropriate. In addition, we are... 2022-06-15 13:33:00
```

Figure 9.7 Whisper output on FED data.

9.4 Sentiment Analysis

After performing the speech-to-text conversion using whisper, the next step is to compute the sentiment of the speech based on the transcribed text. For this task, we use the FinBERT model, Araci (2019), a variant of BERT Devlin et al. (2019) fine-tuned specifically financial data. You might wonder—why use FinBERT for this task instead of the more powerful ChatGPT? One simple reason is cost—both economic and computational. Using a BERT-based model for this demo allows you to work with a widely used industry model that can run on your own computer with minimal resources or with free cloud resources available at the time from Google Cloud. Beyond cost considerations, in the next chapter, we will explore how to customize this model. Specifically, we'll cover how to fine-tune it and optimize it for inference, making it more memory-efficient and faster at runtime—something that would be far more challenging with ChatGPT. With a large proprietary model like ChatGPT, you

would have to rely entirely on OpenAI's infrastructure, limiting customization and hands-on learning.

9.4.1 BERT

The BERT model, which stand for Deep Bidirectional Representation from Transformers, was introduced in 2019 by Devlin et al. (2019).

BERT has been highly influential in both academic research and industrial applications. In industry, Google Search results were powered by a BERT model, eBay's recommender system utilizes BERT, and variants of BERT have been successfully applied in Hamlet's own work at Criteo.

Academically, extensive work has been done to improve BERT's performance, scalability, and efficiency. As of now, the BERT paper has been cited more than 119,678 times according to Google Scholar.

One reason BERT has been so influential is its transformation of the typical workflow in NLP, both in industry and academia. The paper demonstrated the power of pre-training BERT plus fine-tuning for a variety of downstream NLP tasks, consistently outperforming state-of-the-art models at the time and proving invaluable for scenarios with limited training data. Unlike traditional NLP approaches at the time of BERT's introduction, which often required training separate models for each task, the pre-training and fine-tuning approach enabled a more generalized and efficient workflow.

The concept of pre-training and fine-tuning, as highlighted in the paper by demonstrating state-of-the-art experimental results across the board at the time, has driven the industry and academia to build better models by increasing the amount of training data and model size. This approach has led to the development of foundational models, which can later be fine-tuned for more specific tasks. Fine-tuning techniques will be covered in the next chapter.

For this chapter, we will describe the main aspects of the BERT model, as FinBERT, which we use for sentiment analysis, is a fine-tuned version of BERT specifically for financial data.

9.4.1.1 BERT Overview.

BERT is based on a transformer encoder-only architecture, which we covered in [Chapter 5](#) in the Transformers section. Unlike earlier models like GPT-1 (Radford et al., 2018) or ELMo (Peters et al., 2018), which used left-to-right context or a concatenation of left-to-right and right-to-left contexts computed independently, BERT introduced **bidirectional context**. This means that BERT considers both the left and right context of a word simultaneously. Empirical results have shown that this bidirectional context provides embeddings that are better adapted to the surrounding context. The differences in how these models handle contextual information are illustrated in [Figure 9.8](#).

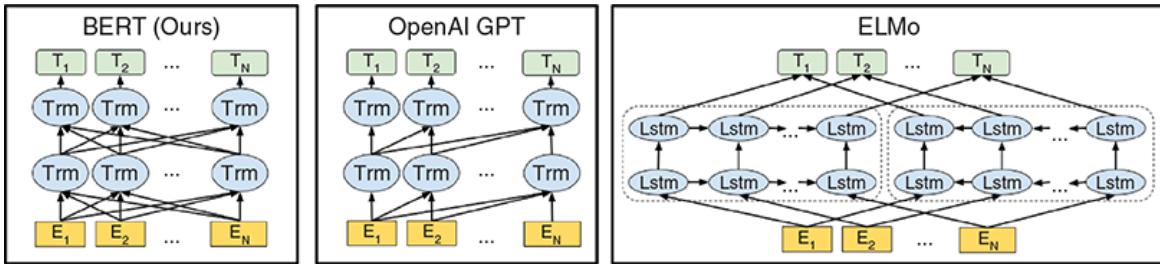


Figure 9.8 Figure 3 from Devlin et al. (2019): Illustration of differences in pre-training model architectures, comparing BERT, OpenAI-GPT, and ELMo. OpenAI GPT uses a left-to-right Transformer. ELMo uses the concatenation of independently trained left-to-right and right-to-left LSTMs to generate features for downstream tasks. Among the three, only BERT representations are jointly conditioned on both left and right context in all layers. In addition to the architecture differences, BERT and OpenAI GPT are fine-tuning approaches, while ELMo is a feature-based approach.

9.4.1.2 Input/Output Representations.

To enable BERT to handle a variety of downstream tasks, Devlin et al. (2019) designed a special input representation. This representation allows BERT to unambiguously represent both single sentences and pairs of sentences with a single token sequence, making it suitable for various downstream tasks. Let's explore how inputs are represented in BERT.

9.4.1.2.1 Input Representations.

As discussed in [Section 5.3.5](#), the first step in the NLP pipeline involves a **tokenization** process, which takes a sequence of raw words as input and outputs a sequence of **token IDs**. Once we have this sequence of token IDs, BERT

augments the input representation. Let's illustrate this process with two sentences, Sentence A and Sentence B, and see how BERT computes their initial representation and processes them. Using the notation from Devlin et al. (2019):

1. Tokenization

- Sentence A is tokenized into a sequence of tokens: $\text{Tok}_1, \text{Tok}_2, \dots, \text{Tok}_N$, where N is the number of tokens in Sentence A.
- Sentence B is tokenized into a sequence of tokens: $\text{Tok}_1, \text{Tok}_2, \dots, \text{Tok}_M$, where M is the number of tokens in Sentence B.

2. Use of special tokens [SEP] and [CLS]

- The tokenized sequences of Sentence A and Sentence B are concatenated and separated by a special token called [SEP]. This token helps BERT distinguish between the two sentences when they are combined into a single token sequence.
- A special classification token [CLS] is added to the beginning of the sequence. This token plays a critical role in downstream tasks, as its final representation is treated as the “aggregate sequence representation” of the input; see Devlin et al. (2019).

3. Initial Embeddings: each token is mapped to an initial embedding (embeddings are discussed in more detail in [Section 5.3.5](#))

- [CLS] token is mapped to $E_{[\text{CLS}]}$.
- [SEP] token is mapped to $E_{[\text{SEP}]}$.
- Sentence A's tokens are associated with embeddings E_1, \dots, E_N .
- Sentence B's tokens are associated with embeddings E'_1, \dots, E'_M .

These steps are part of BERT's input representation process, which is illustrated in [Figure 9.9](#). As an example to understand the dimensions associated with each embedding, the embeddings for each token typically reside in a 768-dimensional hidden space for the BERT-Base configuration or a 1024-dimensional hidden space for the BERT-Large configuration (more details about this are provided in the next section).

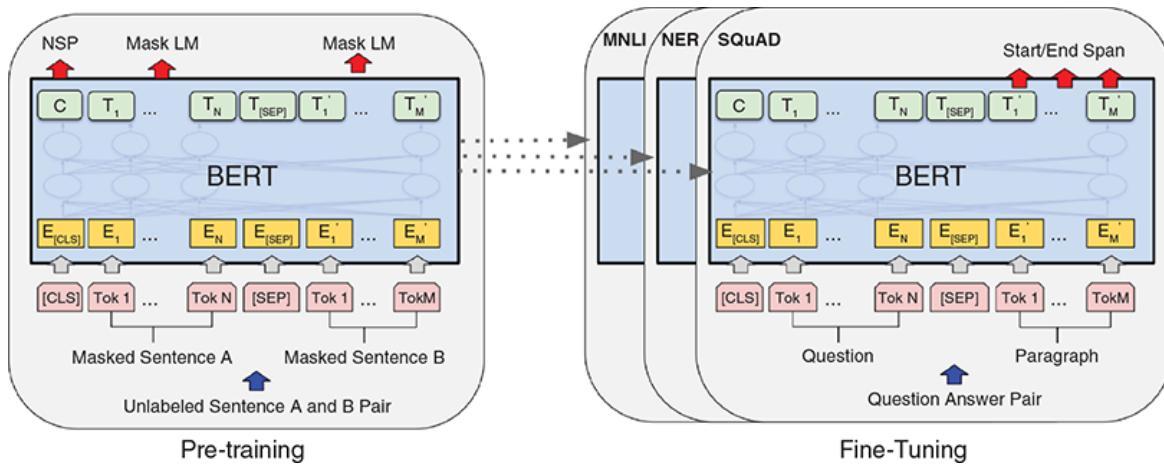


Figure 9.9 Figure 1 from Devlin et al. (2019): Illustration of the input/output BERT representation. Apart from the output layers, the same architecture is used for both pre-training and fine-tuning. The same pre-trained model can be used as an initialization and adapted for multiple tasks, as shown in the image on the right. In the paper, during fine-tuning, all parameters were updated. [CLS] is a special symbol added at the beginning of every input example, and [SEP] is a special token used to separate pairs of sentences.

9.4.1.2.2 Output Representations.

After the Transformer layers process the input, the initial token embeddings are updated to produce the final hidden

states or representations for each token. At the output:

- The final hidden state for the [CLS] token is denoted as C
- The final hidden state for the [SEP] token is denoted as $T_{[\text{SEP}]}$.
- The final hidden states for tokens in Sequence A and Sequence B are T_1, \dots, T_N and T'_1, \dots, T'_M , respectively.

This is also visualized in [Figure 9.9](#).

The role of the special token [CLS], short for classification, is of special importance, it is used as the **aggregate representation** of the entire input sequence. Devlin et al. (2019) used C in experiments related to classification tasks showing its efficacy. In fact, FinBERT also uses C to fine-tune the model to perform sentiment analysis. In such task, C is passed to a classifier, often referred to as the “classification head,” to predict the sentiment of the input sequence. (Remember that a classification head is a neural network added on top of the encoder that maps C , the encoded representation, to the desired number of classes.)

Another key feature of BERT is the introduction of a third type of embedding for encoding the input, known as **segment embeddings**. If you recall from our introduction to Transformers in [Section 5.3.5](#), the input embeddings to transformers are typically computed as a combination of **token embeddings** and **positional embeddings**, which are usually combined by addition. In BERT, **segment embeddings** are learned during the training process and used to indicate whether a token belongs to Sentence A or Sentence B. In summary, inputs embeddings are the sum of three components: token embeddings, positional embeddings, and segment embeddings. This is illustrated in [Figure 9.10](#).

Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[\text{SEP}]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\text{ing}}$	$E_{[\text{SEP}]}$
Segment Embeddings	+ E_A	+ E_A	+ E_A	+ E_A	+ E_A	+ E_A	+ E_B	+ E_B	+ E_B	+ E_B	+ E_B
Position Embeddings	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}

Figure 9.10 Figure 2 from Devlin et al. (2019): Illustration of the BERT input representation. The input embeddings to the Transformer model are the sum of the token embeddings, segment embeddings, and positional embeddings.

This input representation allows BERT to unambiguously represent both a single sentence and pairs of sentences in one token sequence, which enables BERT to be used for downstream tasks like Question and Answering, where we typically provide the dataset as a collection of (Question, Answer) pairs. Moreover, this representation makes the Next Sentence Prediction (NSP) pre-training objective possible, which we will discuss next.

9.4.1.2.3 Pre-training Objectives.

The pre-training objectives of BERT include two tasks:

1. Masked Language Model (MLM) task: In this task, words in a sentence are randomly masked, and the model is trained to predict the missing words. In the BERT paper, 15% of the words are masked. Of these, 80% of the time, the [MASK] token is used, 10% of the time the correct word is used, and the remaining 10% of the time a random word is used.
2. Next Sentence Prediction (NSP): This task involves predicting whether a given sentence B logically follows

sentence A. The model is trained on pairs of sentences from a corpus.

The bidirectional context and the segment embeddings introduced by BERT, along with its two pre-training objectives, are considered the main innovations of the BERT work (Devlin et al., 2019).

The data used for pre-training BERT included the BookCorpus (900M words) (Zhu et al., 2015) and English Wikipedia (2,500M words), with some additional pre-processing described in the paper.

BERT comes in two configurations:

- BERT-Base: 12-layer (transformer blocks), 768-hidden dimension, 12 attention-heads, with a total number of parameters equal to 110M
- BERT-Large: 24-layer (transformer blocks), 1024-hidden dimension, 16 attention-heads, with a total number of parameters equal to 340M

The process of pre-training plus fine-tuning BERT for specific downstream NLP applications outperformed state-of-the-art models at the time, including GPT-1 from OpenAI, in all considered NLP tasks.

9.4.1.3 Fine-tuning BERT for Enhanced Financial Sentiment Analysis: Producing FinBERT.

Fine-tuning BERT on financial data is crucial because general-purpose sentiment models may not accurately capture the nuances of financial vocabulary and market-specific sentiments. For example, the sentence “Mark left Facebook” might be considered neutral by most sentiment models, but in the financial context, it could significantly impact the market. By fine-tuning BERT on financial texts, we

enable the model to learn new concepts, names, and entities specific to finance, resulting in more accurate sentiment analysis. Fine-tuning BERT for multiple tasks is straightforward, as demonstrated by Devlin et al. (2019) in multiple experiments.

As discussed in the previous section, fine-tuning BERT for a classification task can be done by using the final hidden representation of the classification token [CLS], which serves as the “aggregate representation” of the entire sequence. This representation is then fed to a classification head, which outputs the probabilities associated with each sentiment class.

To train the entire model, we simply need a collection of (input, output) pairs. During training, all the model’s parameters are updated, including those of BERT and the classification head. In the literature, BERT is often referred to as the “Body” of the network, to distinguish it from the additional layer appended to perform a specific task, such as classification, which is called the “classification head.”

In Hamlet’s experience, there are variations in what parameters to train or finetune in practice. For certain applications, fine-tuning only the classification head can yield good results. However, this depends on the specific use case and requires empirical validation. As we will see in the next chapter, fine-tuning all the parameters might also be very time-consuming and costly, but it often improves performance. To address this, more efficient fine-tuning methods can be utilized, which will be discussed in [Section 10.5.1](#).

As discussed, fine-tuning for tasks like text classification (e.g., sentiment analysis), the input text is fed into BERT, and the final [CLS] token representation is passed into an output layer for classification. This is the approach followed by the authors of Araci (2019) to produce the FinBERT model for text classification. The authors also experimented with

producing a FinBERT for regression, where the main difference from the classification case is the use of a continuous target, an output layer for regression, and a regression loss function, specifically mean squared error. This score can serve as a representation summarizing the processing of relatively complex alternative data in a downstream multi-factor trading strategy, for example. If you want to learn more about applying Transformer-based models for time series, check out [Section 5.3.6](#), where we have already covered this use case.

The datasets used to fine-tune BERT into FinBERT were as follows:

1. TRC2-financial, which is a subset of Reuters' TRC2. It consists of 1.8M news articles that were published by Reuters between 2008 and 2010. These data are used to pre-train BERT. The corpus can be obtained for research purposes by applying here:
<https://trec.nist.gov/data/reuters/reuters.html>.
2. Financial PhraseBank, see Malo et al. (2014), which is the main sentiment analysis dataset used in this paper.
3. FiQA Sentiment, see *WWW '18: Companion Proceedings of the Web Conference 2018* (2018), a dataset created for the WWW '18 conference financial opinion mining and question-answering challenge. It includes 1,174 financial news headlines and tweets with their corresponding sentiment score.

For more details about the datasets, see Araci (2019).

So, let's explore how we can leverage the FinBERT model for our practical application.

9.4.1.4 Using FinBERT.

In this experiment, we use FinBERT, which has been trained by Araci (2019) and made available at ProsusAI/finbert (<https://huggingface.co/ProsusAI/finbert>). FinBERT is based on fine-tuning BERT-Base, so it has a model size of 110M parameters. We have also written some additional code on top of the ProsusAI/finbert to make sentiment evaluation quite easy.

Here, we illustrate how we can make use of this model for computing the sentiment of sentences. Let's start with a simple example.

To compute the sentiment of the sentence "Mark has left Facebook," we can use the following code:

```
# negative
finbert_sa.predict_overall_sentiment('Mark has left fac
ebook',
    finbert_model)
>>> -0.45569813
```

The sentiment score of -0.45569813 indicates a negative sentiment. Similarly, we can compute the sentiment of other sentences. For example, consider the sentences "Jeff bezos acquires Netflix":

```
# seems neutral
finbert_sa.predict_overall_sentiment('Jeff bezos acquir
es Netflix',
    finbert_model)
>>> -0.0069163926
```

This sentence has a sentiment score of -0.0069163926 , which is quite neutral. Let's look at another example, "Business magnate Elon Musk initiated an acquisition of American social media company Twitter":

```
# seems neutral
finbert_sa.predict_overall_sentiment(
    '''Business magnate Elon Musk
initiated an acquisition of American
social media company Twitter''',
    finbert_model)
>>> 0.090285115
```

The sentence has a slightly positive sentiment score of 0.090285115. Finally, let's evaluate the sentiment of a more complex sentence related to the Federal Reserve:

```
# negative
finbert_sa.predict_overall_sentiment(
    '''The Federal Reserve held interest rates steady,
while also indicating it still expects one more hike
before the end of the year and fewer cuts than
previously indicated next year.'''',
    finbert_model)
>>> -0.13109717
```

This sentence has a negative sentiment score of -0.13109717 . We can also process multiple sentences at one. For example:

```

sentences = [
    'Mark has left facebook',
    'Jeff bezos adquires Netflix',
    '''Business magnate Elon Musk initiated an
        acquisition of American social media company Twitter
    ''',
    '''The Federal Reserve held interest rates steady,
        while also indicating it still expects one more
        hike before the end of the year and fewer cuts
        than previously indicated next year.'''
]
sentiments = finbert_sa.process_sentences(finbert_model
, sentences)

sentiments
>>> array([-4.5569813e-01, -1.7964840e-04,
       9.0285115e-02, -1.3109717e-01], dtype=float32)

```

This array of sentiment scores makes it easy to analyze the sentiment of multiple sentences efficiently.

9.5 Experiment Results

In this section, we describe the results of the model pipeline used to convert FED speech audio data into trading signals. As a reminder, the model pipeline consists of the following steps:

1. Collecting audio from YouTube using pytube.
2. Audio segmentation using ffmpeg.
3. Transcribing the audio bars into text using the Whisper model.

4. Computing sentiment scores for the transcribed text using FinBERT.
5. Generating trading signals based on the sentiment scores, with buy or sell decisions made according to predefined thresholds from the sentiment score distribution.

Price SPY data are sampled at 30-second intervals (30-second bars), and based on sentiment, we attempt to make a trading decision every 30 seconds.

In our test, we collected a total of 5 hours of speech data from the Fed using pytube, corresponding to five different Fed conferences. The audio was downloaded with a specified bit rate (ABR) of 160kbps to ensure high quality.

As previously explained, after downloading the data, the audio was segmented to create 60-second “audio bars,” sampled at 30-second intervals, with timestamps delayed by 15 seconds (embargo) to reduce the risk of introducing lookahead bias. For transcription, we used the Whisper base model for its trade-off between speed and accuracy. In our experiments related to the speech processing part, we noticed that the results were sensitive to using larger versions of Whisper for producing the transcriptions, but this did not significantly affect the results. Instead, the quality of the input data had a more important impact. Therefore, we downloaded the highest quality available that was common across all videos.

Once the audio bars were created along with the corresponding transcriptions, we used FinBERT to compute the sentiment score of each audio bar. Trading signals were generated based on the computed sentiment scores. The sentiment score distribution was analyzed, and thresholds were set based on its quantiles. If the sentiment score was above the upper threshold, we generated a buy signal for the SPY; if it was below the lower threshold, a short signal was

generated. Otherwise, no action was taken unless a position was already open, in which case we closed it.

A summary of the main steps performed by each subsystem is described here:

1. Data collection and preprocessing

- Collected 5 hours of FED speech data
- Downloaded audio with bit rate (ABR) of 160 kbps
- Segmented the audio into 60-second bars with a 30-second sampling interval and a 15-second embargo

2. Transcription and sentiment analysis

- Used the Whisper base model for transcription
- Applied FinBERT to compute sentiment scores for each audio bar

3. Trading signal generation

- Analyzed the sentiment score distribution to set thresholds
- Generated buy or sell signals based on sentiment scores

[Figure 9.11](#) shows the SPY forward return and sentiment score time series produced by the system:

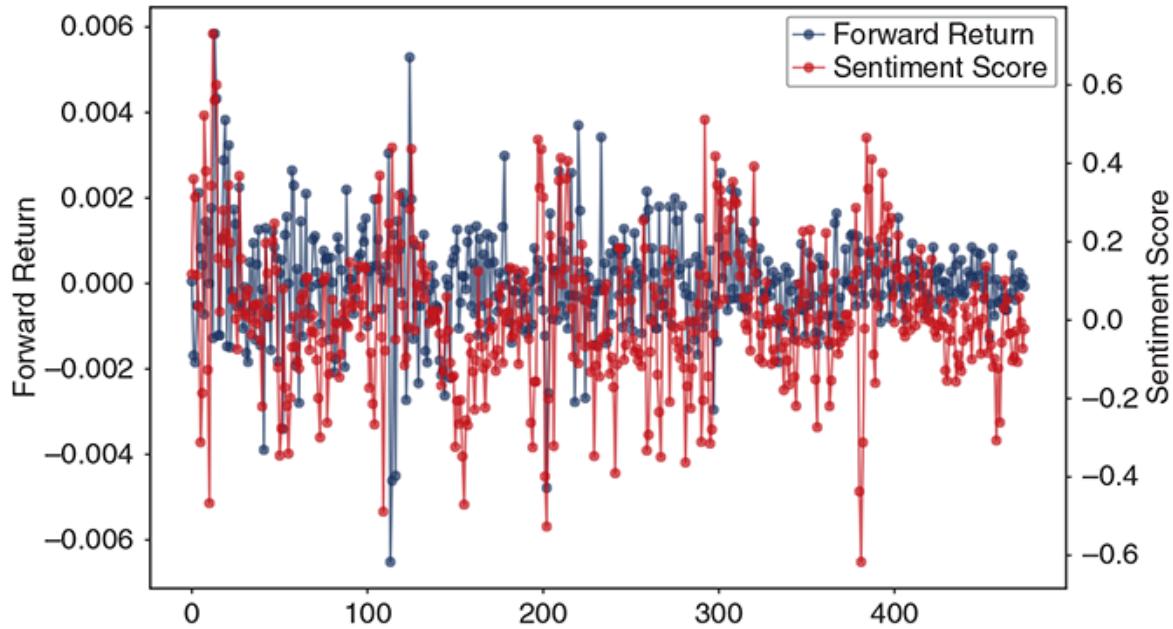


Figure 9.11 Time-series sentiment signal and forward returns.

In [Figure 9.12](#) we present the scatter plot between the sentiment score and the SPY forward returns, which reveals a positive correlation between them.

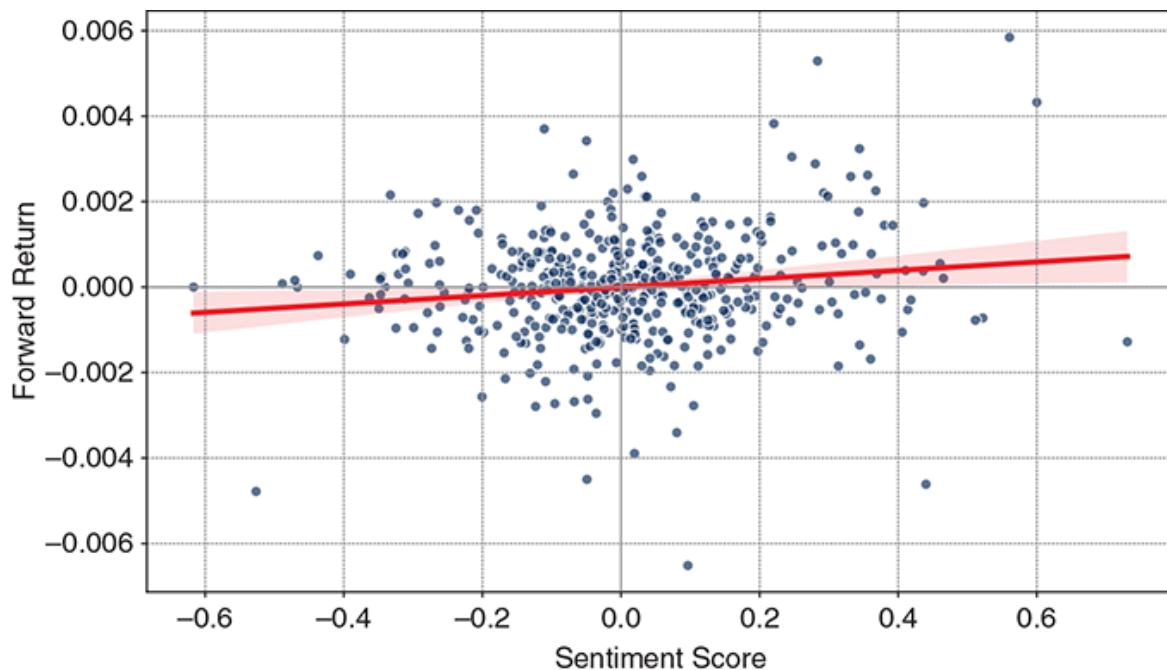


Figure 9.12 Scatter plot of sentiment signal vs forward returns.

Finally, [Table 9.1](#) summarizes the key performance results of the strategy. These results include a simple constant model for transaction costs, assuming 5 basis points (bps) per transaction. Overall, across all videos, the trading signal predicts the sign of the SPY forward returns with an accuracy of 53.15%. The correlation between the sentiment score and the SPY forward returns is 14.14% with a *p*-value of 0.199%.

TABLE 9.1			
Performance table.			
Unnormalized Sharpe ratio	Pearson Correlation of Sentiment Signal Accuracy vs SPY	<i>p</i>-value	
11.3844%	53.1532% 14.1458%	0.1999%	

9.6 Conclusion

The experimental results demonstrate how we can effectively process alternative data, in this case, audio, to generate actionable trading signals. By collecting high-quality audio from the internet, using appropriate audio segmentation, and employing complex models for transcription and sentiment score computation, we were able to generate profitable high-frequency trading signals to trade the next 30 seconds of the SPY. This application highlights the power of deep autoregressive transformer models for modeling audio and text, as covered in [Chapter 5](#). Our experiments demonstrated the critical role of FinBERT, a fine-tuned BERT model for financial data. Unlike general sentiment analysis models, which often miss financial nuances, FinBERT outperformed other tested models not trained on financial data, significantly improving our performance metrics. The next chapter explores methods for customizing LLMs for specialized tasks. This approach highlights the potential for composing very complex deep learning models to address complex financial applications.

Chapter 10

Efficient Inference

10.1 Introduction

In the past, the typical machine learning workflow in industry was “relatively” straightforward: collect data for the problem at hand, train a model on that data, and then deploy the trained model directly into production to respond to new inputs.

However, with the advent of increasingly large models, this traditional approach has become less feasible. The large size of modern models often makes it impractical to deploy them in production as they are, at least for most companies or institutions. This can be due to constraints such as latency requirements for the specific applications these models are used for, or due to the prohibitive infrastructure costs associated with running these models at scale.

As a result, the typical machine learning workflow in the industry has evolved. Instead of deploying the model directly after training, a compression step is now commonly introduced. This compression step takes the model produced after training and typically produces a smaller version—smaller in terms of the number of parameters or the memory required to store them—while striving to maintain or even improve the performance of the original model. Nowadays, model compression is essential for meeting system requirements for certain applications, such as reducing latency or simply to reduce infrastructure costs.

One example from industry is the eBay case study (see Xue et al. [2023]). They improved their recommender system by fine-tuning BERT on eBay item titles, which often include

product-specific words, alongside the Wikipedia corpus, that is more suited to the needs of eBay language understanding tasks. In offline evaluations, this eBERT model outperforms out-of-the-box BERT models on a collection of eBay-specific tagging tasks. However, the eBERT model was too large to achieve low latency specifications at inference time, making it challenging to deliver recommendations in real time. To address this issue, eBay used model compression techniques to produce a smaller version of BERT that met their inference requirements. They also optimized the model for CPU inference to reduce costs. For more details on their work, see Xue et al. (2023).

In Hamlet’s professional experience at Criteo, he has seen that compression techniques are crucial not only for cost reduction, which is a significant concern, but also for meeting system design constraints. While massive parallelization techniques can be employed to deploy models and potentially meet latency requirements, in some cases, the high costs associated with these solutions can diminish the potential benefits of the system.

In this chapter, we introduce the most common techniques used in the industry for efficient inference, such as knowledge distillation, and model quantization. To illustrate these techniques, we use FinBERT—the model applied in our practical application in the last chapter—as a case study, demonstrating how to make it faster at inference. We conclude the chapter by showcasing techniques for customizing LLMs, allowing you to adapt models to your specific needs.

10.2 Scaling Large Language Models: High Performance, High Computational Cost, and Emergent Abilities

As discussed in the previous chapter, where we explored one of the first, most widely used, and successful large language models, BERT, the modern approach to solving a variety of NLP tasks involves a two-step training process: pre-training and fine-tuning. In the first step, a large model is pre-trained on vast amounts of data, then fine-tuned to solve specific NLP tasks.

Nowadays, there is strong evidence that improving the model's performance during the pre-training phase leads to improvements across multiple NLP tasks. For example, in the paper by Devlin et al. (2019), they empirically demonstrated that the larger version of BERT, named BERT-Large, outperformed the smaller version, BERT-Base, during pre-training and consistently outperformed it across all the downstream NLP tasks considered.

Given this evidence, a natural question arises: if strong performance during pre-training is crucial, how should we approach this phase effectively? Among the many variables and strategies available, which are the most critical?

In the paper by Kaplan et al. (2020), the authors found that one path to improving performance lies in scaling up the models—specifically, increasing the number of model parameters, the amount of training data, and the duration of training time. When these factors are increased, they lead to significant gains in performance. The authors empirically found that performance follows a power-law relationship with respect to these factors, as shown in [Figure 10.1](#).

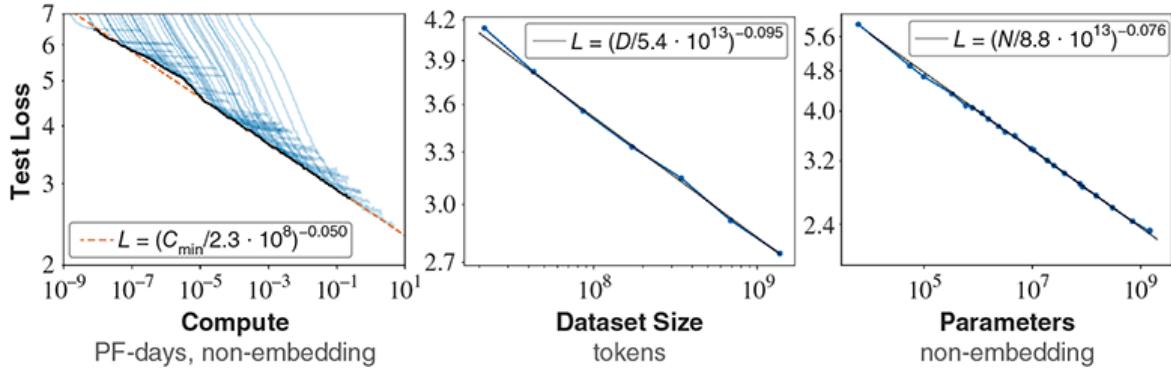


Figure 10.1 Figure 1 from Kaplan et al. (2020): Illustration of how language modeling performance improves with increases in model size, dataset size, and the amount of compute used for training. For optimal performance, all three factors must be scaled up in tandem. Empirical performance exhibits a power-law relationship with each individual factor when the other two are not limiting.

For more details about how other parameters affect performance, refer to the paper by Kaplan et al. (2020).

These empirical laws, which suggest that increasing any of the mentioned factors can lead to almost predictable improvements in model performance, have further motivated the industry to scale up models. Scaling up models is also crucial for unlocking *emergent abilities* of large models, which we briefly discuss on the next section.

While this path of scaling up is highly effective for improving performance, it also comes with some drawbacks, such as increased inference costs and the infrastructure required to serve these large models.

10.2.1 Emergent Abilities

While the scaling laws provide a powerful path for improving model performance, recent evidence suggests that they are also key for unlocking the *emergent abilities* of large

language models. Emergent abilities is defined by Wei et al. (2022) as, “An ability is emergent if it is not present in smaller models but is present in larger models.”

There are two main aspects of emergent abilities that are important to know:

- They cannot be directly predicted by extrapolating from scaling laws (like the one discussed earlier).
- These abilities begin to emerge only after the model reaches a certain scale.

In the following section, we will use the main work by Wei et al. (2023) on emergent abilities of large models. Particularly, we will illustrate how model size and training time contribute to the emergence of these abilities.

10.2.2 Impact of Model Size

To illustrate the impact of model size on emergent abilities, let's examine how Large Language Models (LLMs) perform in solving middle school math word problems. In the work by Wei et al. (2023), the GSM8K dataset is used for this evaluation, where the LLMs are asked to solve math problems using both standard prompting and chain-of-thought prompting. According to Wikipedia, “Chain-of-thought (CoT) prompting is a technique that allows large language models (LLMs) to solve a problem as a series of intermediate steps before giving a final answer” (see [https://en.wikipedia.org/wiki/Prompt_engineering]).

In [Figure 10.2](#), you can see the performance measured as solve rate (%) versus model scale, represented by the number of model parameters in billions.

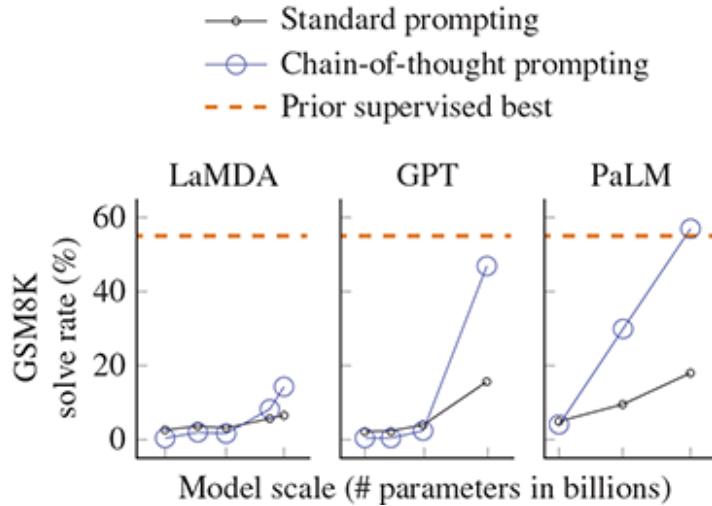


Figure 10.2 Figure 4 from Wei et al. (2023): An illustration of how performance (solve rate) improves as model scale, measured in billions of parameters, increases for standard prompting versus chain-of-thought prompting. Chain-of-thought prompting enables LLMs to solve challenging math problems, with chain-of-thought reasoning notably emerging as an ability with increasing model scale.

The plot shows the performance of three different models: LaMDA, GPT, and PaLM. A clear pattern emerges for each of these models: as the number of parameters increases, so does the performance. For example, focusing on the results using chain-of-thought prompting, which tends to yield better results, LaMDA, with 137 billion parameters, achieves a performance of less than 20%. In contrast, GPT, with 175 billion parameters, achieves a performance exceeding 40%. PaLM, with 540 billion parameters, reaches a performance of around 50%.

10.2.3 Effect of Training Time

Evidence suggests that emergent abilities only appear after a large number of training iterations. It's not enough to simply

have a large model; it must also be trained extensively to unlock these capabilities.

In [Figure 10.3](#), taken from the paper by Wei et al. (2022), we see the performance of various models—LaMDA, GPT-3, Gopher, Chinchilla, PaLM, and a Random Model—on eight different common NLP tasks, like Modular Arithmetic, Multi-task Natural Language Understanding (NLU), Word-in-Context, among others.

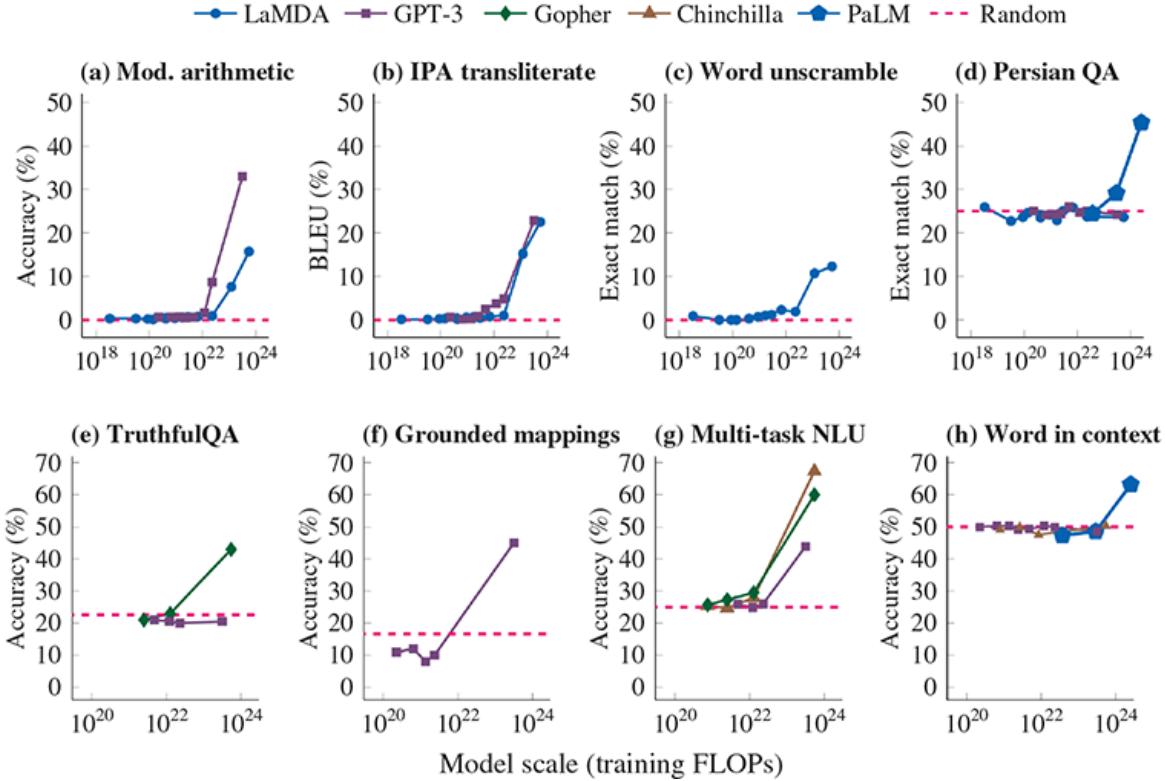


Figure 10.3 Figure 2 from Wei et al. (2022): Illustration of performance, measured by accuracy, versus model scale (measured in training FLOPs) for eight different tasks. The figure shows how performance transitions from random levels to significantly above random after reaching a certain scale threshold. These tasks demonstrate examples of emergent abilities in the few-shot prompting setting, which only appear after sufficient training.

Each graph shows model performance (measured in accuracy) as a function of the model scale (training FLOPS, or training floating point operations per second). For example, looking at the Modular Arithmetic task, large models like GPT-3 and PaLM show a significant performance boost only when their training FLOPS exceed 10^{22} . Similarly, for the Multi-task NLU task, models such as Chinchilla, Gopher, and GPT-3 only start improving once their training FLOPS pass the same threshold. We observe the same behavior for the Word-in-Context task as well.

The key takeaway is that not only do we need large models, but we also need to train them for many iterations to improve their performance and unlock emergent abilities.

However, it is important to note that even after extensive training, large models can be slow during inference due to their huge number of parameters. This creates the need for methods to speed up inference, as the ones described in this chapter.

Lastly, although not explicitly shown in the figures, the amount of training data is also a critical factor in the model's performance. Large models benefit from vast amounts of data to learn from, further enhancing their capabilities.

10.2.4 Efficient Inference for Deep Models

To develop efficient inference for deep models, recent work has focused on model compression and acceleration techniques, such as the following:

1. Knowledge distillation
2. Quantization
3. Parameter pruning
4. Neural Architecture Search

In this chapter, we will focus only on techniques 1 and 2.

10.3 Making FinBERT Faster

10.3.1 Knowledge Distillation

Knowledge distillation is a technique used to create a faster model (the *student*)—meaning one with lower inference time—that approximates the performance of a slower but more

powerful model (the *teacher*). In this process, the student model is by design smaller than the teacher model, with fewer parameters, making it more efficient at inference time. Knowledge distillation involves a learning process for the student model where it not only learns from data, as in typical training, but also has knowledge transferred from the teacher model.

In our case study, the teacher model we want to approximate is FinBERT; see Araci (2019). As a reminder, FinBERT is a fine-tuned version of BERT that has been specifically trained for sentiment analysis prediction in the financial domain.

If our only concern were inference speed, we could simply train the student model using the given data (which, for example, could be a combination of pre-training on the BERT corpus followed by fine-tuning with the financial data used by FinBERT). However, because the student model is smaller than the teacher, it would likely underperform compared to the teacher. Knowledge distillation offers a different approach to this situation: it involves not only learning from the data but also aligning the student model with certain aspects of the teacher model. In its simplest form, this alignment could involve matching the probability distribution that the teacher model assigns to a given input, as described by Hinton et al. (2015) in their paper on knowledge distillation. (Yes, the same Hinton who shared the 2024 Nobel Prize in Physics with John Hopfield and is considered one of the Godfathers of Deep Learning. See https://en.wikipedia.org/wiki/Geoffrey_Hinton.)

An analogy can help clarify this process. Consider how we learn a subject in school: imagine we have the option to study directly from a textbook or to attend lectures given by a teacher who is an expert in the subject. Most of the time, a combination of both—attending lectures and reading the textbook—leads to a better understanding than either approach alone. In this analogy, the textbook represents the

data used in standard training, while the teacher represents the model that also guides the learning process. Similarly, in knowledge distillation, the student model benefits from learning both from the data and from the guidance provided by the teacher model.

To enable the student model to learn both from the data and from the teacher model, the cost function used during training must account not only for how well the student model fits the data but also for how well it matches the specific aspects of the teacher model that we want to capture.

We will explore how to apply knowledge distillation to FinBERT, demonstrating how this technique can be used to create a faster model without sacrificing too much performance.

10.3.1.1 Which Aspect of the Teacher Model to Match.

In knowledge distillation, there are various aspects of the teacher model that the student model can try to align with. These different aspects are known in the knowledge distillation literature as different types of knowledge. For example, in the most basic form of knowledge distillation, often called vanilla knowledge distillation, the focus is on aligning the probability distribution that the teacher model assigns to different classes given the input, as in the case of a classification task. This approach was popularized by Hinton et al. (2015).

However, there are other forms of knowledge that can be transferred from the teacher to the student model. For instance, the intermediate representations computed by the network are a potential source of knowledge that can be transferred. Some methods use the activation features of intermediate layers in the teacher model to guide the learning of the student model, as described by Romero et al.

(2015) in their work on FitNets. Also, the parameters or weights of the teacher model can also be a different source of knowledge.

These different types of knowledge to transfer fall into three categories: response-based knowledge, feature-based knowledge, and relation-based knowledge. A very good overview of these classification can be found in the excellent survey on knowledge distillation by Gou et al. (2021).

For the purpose of this book and in our case study, we will focus on the response-based knowledge, which is the simplest yet highly effective method for model compression.

10.3.1.2 Response-based Knowledge.

Response-based knowledge refers to a scenario where the student model attempts to mimic or match the output of the teacher model's last layer, meaning that we try to match its final prediction.

In typical classification tasks, the most popular form of response-based knowledge is known as *soft targets*, as described in Hinton et al. (2015). In this context, soft targets refer to the probability distribution allocated by the teacher model over the different classes for a given input. This contrast with the “ground truth” or *hard targets*, where the probability mass is fully allocated to the “correct class.”

In neural networks for classification tasks, the class probabilities are usually generated by applying a “softmax” function to the logits, which are the outputs of the last fully connected layer of the network. Following the notation introduced in Hinton et al. (2015), we denote the outputs logits as the vector \mathbf{z} , whose dimension depends on the number of classes specified by the classification problem. (Logits are the outputs of a neural network before the softmax activation function is applied. They are scores of the

input belonging to a certain class. Recall the discussion on Logistic Regression in [Chapter 3](#)). The probability distribution is then computed by applying the softmax function to \mathbf{z} :

$$p(z_i, T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

where z_i is the output corresponding to the i -th class, and T is a parameter known as the *temperature*. Typically, in classification problems, T is set to 1, but in knowledge distillation, using larger values of T tends to produce a “softer” probability distribution over the classes.

For example, in [Figure 10.4](#), we show how the output of the softmax function applied to the vector of logits

$\mathbf{z} = [30, 20, 10]^T$ changes as we increase the temperature. At a temperature of 1, the output is close to a hard target, with almost all of the probability mass allocated to a single class. As the temperature increases, the probability distribution becomes progressively softer. For very large temperatures—such as in our case temperature = 1,000—the distribution approaches a uniform distribution across all classes.

(Physicists will recognize this as the Boltzman distribution that gives the probability that a system will be in a certain state as a function of that state’s energy and the temperature of the system.)

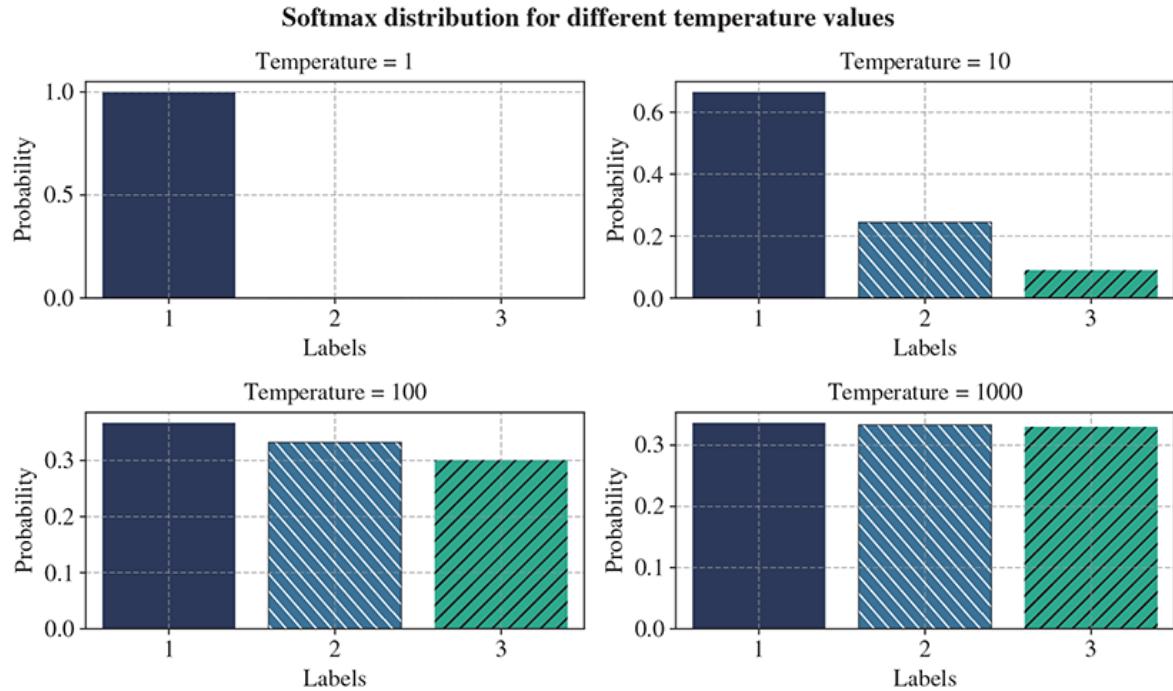


Figure 10.4 Softmax distribution.

As we have discussed, the goal in knowledge distillation is for the student model to learn both from the data and from the teacher model, and specifically in soft targets response-based knowledge, from the soft targets produced by the teacher. To achieve this, we need an objective function for the student that accounts for how well it fits the empirical data distribution (typically measured by the cross-entropy loss in classification problems) and how well it matches the teacher's soft targets. One approach is to use a weighted average of both objectives:

$$\mathcal{L}_{\text{student}} = \alpha \mathcal{L}_{\text{CE}}(y, p(\mathbf{z}_s, T=1)) + (1 - \alpha) \mathcal{L}_{\text{KD}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T))$$

where $\mathcal{L}_{\text{student}}$ represents the total loss for the student, \mathcal{L}_{CE} represents the cross-entropy loss between the student model and the ground truth label y , α is the weight hyper-parameter between 0 and 1, and \mathcal{L}_{KD} is the distillation loss that measures how well the student's distribution $p(\mathbf{z}_s, T)$ aligns with the teacher's distribution $p(\mathbf{z}_t, T)$.

The knowledge distillation loss is typically proportional to the Kullback-Leibler (KL) divergence,

$$\mathcal{L}_{\text{KD}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T)) = T^2 D_{\text{KL}}(p(\mathbf{z}_t, T), p(\mathbf{z}_s, T))$$

where T is the temperature and D_{KL} is the KL divergence, which measures the difference between the teacher and student distributions. The factor T^2 ensures that the relative contributions of the hard and soft targets remain balanced. For more detailed explanations about the proportionally factor, we encourage you to refer to Hinton et al. (2015)

10.3.1.3 Implementation Details.

We will use the Hugging Face (<https://huggingface.co>) transformers python library for performing knowledge distillation due to its powerful API and utilities, which greatly simplifies the training and inference with transformer models. In the repository accompanying this book, you will find a short tutorial covering the basics of this library, making sure you have the knowledge needed to understand the relevant components discussed in the book.

In the code snippets provided later, we focus on the most important steps for implementing knowledge distillation. Many steps, such as dataset creation, tokenization, train-test splitting, among others, are common across machine learning and NLP workflows and not specific to this project, so they are not shown here. However, you can find the complete step-by-step process for building this use case in the accompanying notebook for this chapter.

Our goal is to find a balance between theory and specific implementation, allowing you to understand the core concepts of the topic and apply them using different libraries, such as PyTorch Lightning

(<https://lightning.ai/docs/pytorch/stable/>), or even implementing everything in plain Pytorch or TensorFlow. We hope this approach give you a solid understanding of the theory while providing detailed implementation examples in the notebook, which can be updated as new powerful tools become available.

Now that we understand the student loss described earlier—a linear combination of cross-entropy loss (commonly used for classification tasks) and knowledge distillation loss (proportional to the KL divergence between the teacher and student distributions)—we can proceed to implement it.

For standard training procedures, we typically use the Trainer class of the transformers library, which provides an API for training models in PyTorch, offering many features out of the box, such as distributed training on multiple GPUs/TPUs, etc. More information can be found in the library documentation

https://huggingface.co/docs/transformers/v4.15.0/main_classes/trainer.

Since our objective is to train the model using the distillation loss $\mathcal{L}_{\text{student}}$, and the Trainer class does not support this loss out of the box, we need to extend its capabilities. Fortunately, this is straightforward.

We extend the standard Trainer class to create a DistillationTrainer, as suggested in Tunstall et al. (2022), which supports the student loss function necessary for knowledge distillation. The DistillationTrainer can be instantiated as shown in the following code:

```
distilbert_trainer = DistillationTrainer(  
    model_init=student_init,  
    teacher_model=teacher_model,  
    args=student_training_args  
    train_dataset=YOUR_TRAINING_SET,  
    eval_dataset=YOUR_EVALUATION_SET,  
    compute_metrics=compute_metrics,  
    tokenizer=student_tokenizer)
```

The `DistillationTrainer` object takes the following inputs:

- `student_init`: a function used to provide the initialization of the student model
- `teacher_model`: the teacher model used for distillation
- `student_training_args`: an instance of the `TrainingArguments` class
- `YOUR_TRAINING_SET`: a `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, or `datasets.Dataset`, the dataset used to train the model
- `YOUR_EVALUATION_SET`: a `torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, or `datasets.Dataset`, the dataset used to evaluate the model
- `compute_metrics`: a function used to report metrics on the evaluation set, such as precision, recall, accuracy, etc.
- `student_tokenizer`: The tokenizer used to preprocess the data

A key difference, from the input arguments point of view, between the `DistillationTrainer` and the standard `Trainer` class is the inclusion of the `teacher_model` argument, which is essential for knowledge distillation.

From a functional perspective, `DistillationTrainer` must compute the correct loss function. To do this, it needs access

to the α and temperature T hyperparameters. The implementation of the `compute_loss` method is shown here:

```
import torch.nn as nn
import torch.nn.functional as F
from transformers import Trainer

class DistillationTrainer(Trainer):
    def __init__(self, *args, teacher_model=None, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher_model = teacher_model

    def compute_loss(self, model, inputs, return_outputs=False):
        device = torch.device("cuda" if
torch.cuda.is_available() else "cpu")
        inputs = inputs.to(device)
        outputs_stu = model(**inputs)
        # Extract cross-
entropy loss and logits from student
        loss_ce = outputs_stu.loss
        logits_stu = outputs_stu.logits
        # Extract logits from teacher
        with torch.no_grad():
            outputs_tea = self.teacher_model(**inputs)
            logits_tea =
outputs_tea.logits

        # Soften probabilities and compute distillation
loss
        loss_fct = nn.KLDivLoss(reduction="batchmean")
        loss_kd = self.args.temperature ** 2 *
loss_fct(
            F.log_softmax(logits_stu /
self.args.temperature, dim=-1),
            F.softmax(logits_tea /
self.args.temperature, dim=-1))
        # Return weighted student loss
        loss = self.args.alpha * loss_ce + (1. -
self.args.alpha) * loss_kd
```

```
        return (loss, outputs_stu) if  
return_outputs else loss
```

In this customized Trainer class, we:

- Save the teacher model in the `__init__` method.
- Compute the student loss in the `compute_loss` method, where:
 - `loss_ce` corresponds to the cross-entropy loss.
 - `loss_fct` is an instance of the `nn.KLDivLoss` object, which computes the KL divergence loss.
 - `loss_kd` corresponds to the knowledge distillation loss between the teacher and student distributions.
 - `loss` is the linear combination of both losses.

As shown in the code, the class accesses α and T as `self.args.alpha` and `self.args.temperature` through the `student_training_args` parameters object. To make these available during training, we extend the `TrainingArguments` class as shown here:

```
from transformers import TrainingArguments  
  
class DistillationTrainingArguments(TrainingArguments):  
    def  
        __init__(self, *args, alpha, temperature, **kwargs):  
            super().__init__(*args, **kwargs)  
            self.alpha = alpha  
            self.temperature = temperature
```

After creating our custom trainer and custom training arguments, we can train our student model using the convenient features provided by the `transformers` library by simply calling `distilbert_trainer.train()`. As previously mentioned, the detailed step-by-step process for distilling

FinBERT is provided in the accompanying notebook, while this section focuses on the key concepts and implementation details.

10.3.2 Case study results. Distilling FinBERT

In the original paper by Araci (2019) on FinBERT, the authors used two different datasets to fine-tune the model. One of these datasets is TRC2-financial, which was employed to improve the relevance of corpus to financial keywords. You can obtain these dataset for research purposes by applying here: <https://trec.nist.gov/data/reuters/reuters.html>.

The second dataset, Financial PhraseBank created by Malo et al. (2014), is the main dataset used the FinBERT paper for the specific task of sentiment analysis. It can be obtained for free here:

https://www.researchgate.net/publication/251231364_FinancialPhraseBank-v10. In our experiments, this is the dataset we used to perform knowledge distillation on FinBERT.

To ensure a fair comparison, we used the same train-test split specified by the original authors, as provided in their GitHub repository, FinBERT: Financial Sentiment Analysis with BERT (<https://github.com/ProsusAI/finBERT>). This allowed us to report evaluation metrics that are comparable to those in the original paper. The evaluation metrics we used to assess performance are the same as those in the original study, namely accuracy, and macro F1 average.

Our first step was to reproduce the results reported in the original paper, which we successfully did on the test portion of the TRC2-financial sentiment dataset.

For the student model, we selected distilbert/distilbert-base-uncased, provided by Hugging Face at <https://huggingface.co/distilbert/distilbert-base-uncased>.

The Transformers library makes it easy to download models

from HuggingFace. More information about this process can be found in the accompanying notebook for this chapter and the HuggingFace tutorial in the book repository.

In terms of performance, our DistilledFinBERT slightly outperforms the teacher model, FinBERT, which is an excellent result. Detailed results can be found in [Table 10.1](#).

TABLE 10.1

Performance metrics of teacher vs. student models.

Model	Precision	Recall	F1-Score
FinBERT	0.85	0.84	0.84
DistilledFinBERT	0.86	0.85	0.85

Since we managed to maintain or even improve performance, how does it compare in terms of inference speed? The results are shown in [Table 10.2](#).

TABLE 10.2

Inference speed of teacher vs. student models.

Model	Average Latency (ms)
FinBERT	0.034
DistilledFinBERT	0.016

DistilledFinBERT achieves a 2.125 times faster inference speed compared to the original FinBERT. In conclusion, our DistilledFinBERT not only matches the performance of the original model on the same test set, but it also offers significantly faster inference, making it a more efficient option for deployment.

10.4 Model Quantization

Quantization refers to the process of reducing the set of values a variable can take so that the variable can be represented using fewer bits, thereby reducing its precision. The key question behind quantization is: Can deep neural networks operate effectively with lower precision?

In deep learning, we are particularly interested in applying quantization to the weights, biases, and activations of our models. Reducing the bit precision of these elements leads to improvements in memory storage, computational speed, and energy consumption. Modern hardware is equipped with specialized instructions for integer arithmetic, which further enhance computational speed when operating in lower precision. However, this reduction in precision can come at the cost of accuracy, as lower bit representation might lose information.

Training and inference in deep neural networks have distinct precision requirements. During training—a highly dynamic process involving a wide range of weight values—a large dynamic range is important. On the other hand, inference often prioritizes precision over a small dynamic range, as the model makes predictions based on pre-trained weights that often concentrates around a particular value. By understanding the different requirements of training and inference, we can design data types optimized for each stage.

As of the time of writing, there are two main quantization techniques: K-Means quantization and linear quantization. In this section, we will focus on linear quantization and explore how it can be applied to achieve efficient model inference.

10.4.1 Linear Quantization

Linear quantization applies an affine mapping from the space of integers to the space of real numbers (or floating-point space) using the following formula: $r = S(q - Z)$, as described in Jacob et al. (2017). Here:

- S is the scale, a floating-point value.
- Z is the zero point, an integer that represents the offset or bias.
- q is the quantized number, an integer.
- r is the real value, a floating-point number.

The formula is a reconstruction formula, mapping quantized values q back to real values r . In practice, though, we usually start with the real value r and need to find its quantized counterpart, q .

We can view this equation as a mapping from the integer range $[q_{min}, q_{max}]$ to the floating-point range $[r_{min}, r_{max}]$. The range of the integer representation depends on the number of bits used. For example, in signed integer representations, the range of values for different bit widths is shown in [Table 10.3](#):

TABLE 10.3

Integer range for different bit widths.

Bit width	q_{min}	q_{max}
2	-1	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1} - 1$

The floating-point range $[r_{min}, r_{max}]$ is determined by the statistics of the real values input to be quantized. To determine the scale S , we impose that q_{min} maps to r_{min} and q_{max} maps to r_{max} , solving the following system of equations:

$$\begin{aligned} r_{max} &= S(q_{max} - Z) \\ r_{min} &= S(q_{min} - Z) \end{aligned}$$

Subtracting these two equations gives us the value of S :

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

The zero point Z is found by solving for Z using $r_{min} = S(q_{min} - Z)$, which results in:

$$Z = \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$

Since Z is an integer, we round it. To compute the quantized value q , we use:

$$q = r/S + Z$$

Finally, q is clamped to the range $[q_{min}, q_{max}]$ and rounded to the nearest integer. Although there are different methods for rounding, this example uses the simplest case, but adaptive rounding methods also exist.

10.4.1.1 Example of Linear Quantization.

Let's illustrate linear quantization with a toy example to give a better understanding of the operations involved. Consider a weight matrix that we want to quantize using 2 bits:

```

weights = np.array([
    [-1.0856306  0.99734545  0.2829785 ],
    [-1.50629471 -0.57860025  1.65143654],
    [-2.42667924 -0.42891263  1.26593626]
])

```

First, we compute the scale S . The dynamic range of the weight matrix, defined by its minimum and maximum values, is $r_{min} = -2.42667924$ and $r_{max} = 1.65143654$. Using a 2-bit representation, the integer range is $[-2, 1]$. From this, we compute the scale as $S = 1.35937$. We can then find the zero point $Z = 0$. Finally, we compute the quantized weights using the equation provided:

```

quantized_weights = (weights / scale +
zero_point).clamp(qmin, qmax).round()

>>> quantized_weights
np.array([
    [[-1.  1.  0.],
     [-1. -0.  1.],
     [-2. -0.  1.]]])

```

If we attempt to reconstruct the original matrix from the quantized version, we obtain:

```

>>> reconstructed_weights
np.array([
    [-1.3594  1.3594  0.      ],
    [-1.3594 -0.      1.3594],
    [-2.7187 -0.      1.3594],
])

```

The reconstruction error is the difference between the original and reconstructed weights:

```
>>> reconstruction_error
np.array([
    [ 0.2737 -0.362   0.283 ],
    [-0.1469 -0.5786  0.2921],
    [ 0.2921 -0.4289 -0.0934]
])
```

10.4.2 Quantizing an Attention Layer in Distilled FinBERT

Now, let's apply this process to one of the attention layers in our Distilled FinBERT model using 8 bits. [Figure 10.5](#) is a distribution of the parameters of the layer, following a similar setup to that in Tunstall et al. (2022).

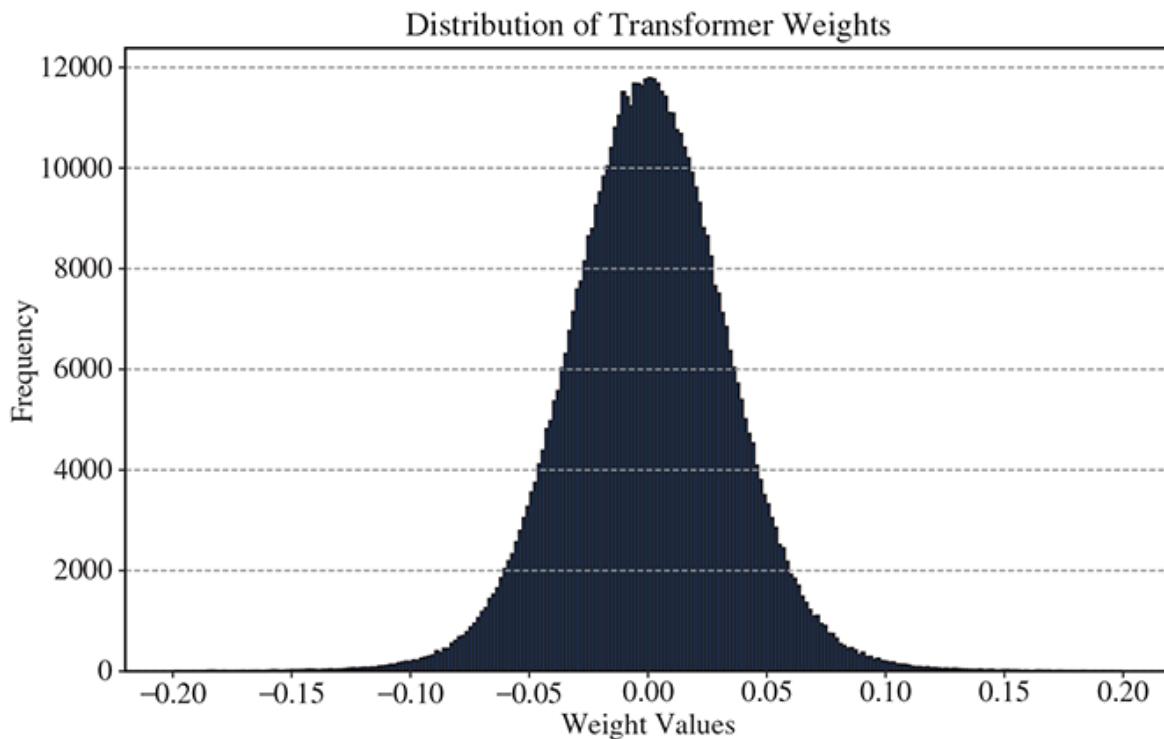


Figure 10.5 Weights distribution.

As shown in [Figure 10.5](#), the weights are concentrated within a narrow range of values, exhibiting a small dynamic range. Since we are using signed 8-bit integers, the range is

$[-127, 128]$. The scale factor and zero point are calculated similarly to the previous example.

Next, we compute the quantized weights using the formula $q = r/S + Z$, clamp the values, round them to the nearest integer, and store them in the `torch.int8` data type:

```
manually_quantized_weights = (weights / scale +
    zero_point).clamp(qmin, qmax).round().char()
>>> manually_quantized_weights
tensor([[ -5,   -8,    0, ...,   -6,   -3,    8],
        [  8,    3,    1, ...,   -4,    7,    1],
        [ -9,   -5,    5, ...,    0,    6,   -3],
        ...,
        [  5,    0,   13, ...,    0,    6,   -1],
        [  0,   -2,  -12, ...,   12,   -8,  -13],
        [-13,   -1,   -9, ...,    8,    2,   -2]], dtype=tor
ch.int8)
```

Torch also provides a built-in function, `torch.quantize_per_tensor`, for this process. Here's how the result compares:

```

from torch import quantize_per_tensor
# torch as a function which performs this operation for
# us
dtype = torch.qint8
quantized_weights =
quantize_per_tensor(weights, scale, zero_point, dtype)

quantized_weights.int_repr()

tensor([[ -5,   -8,    0,    ... ,   -6,   -3,    8],
       [  8,    3,    1,    ... ,   -4,    7,    1],
       [ -9,   -5,    5,    ... ,    0,    6,   -3],
       ... ,
       [  5,    0,   13,    ... ,    0,    6,   -1],
       [  0,   -2,  -12,    ... ,   12,   -8,  -13],
       [-13,   -1,   -9,    ... ,    8,    2,   -2]], dtype=tor
ch.int8)

```

Finally, let's compare the results:

```

# make sure they are equivalent

assert abs(quantized_weights.int_repr() -
manually_quantized_weights).sum().item() <= 1e-5

```

[Figure 10.6](#) is the distribution of the quantized weights.

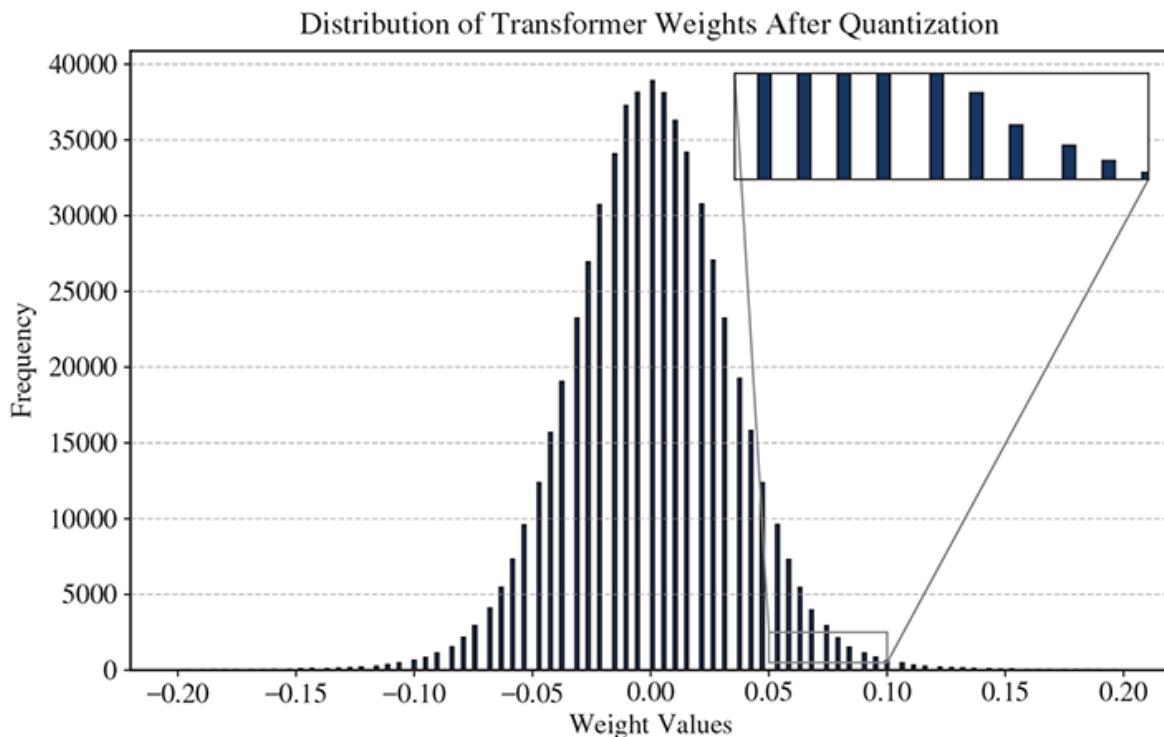


Figure 10.6 Distribution of quantized weights.

To verify the gain in memory efficiency from moving from FP32 to INT8, we can measure it directly in code:

```
import sys

# 4x time compression!

sys.getsizeof(weights.storage()) /
sys.getsizeof(quantized_weights.storage())

>>> 3.999755879241598
```

We also measure the speedup of performing the operation `weights*weights`, as shown in [Table 10.4](#).

TABLE 10.4

Performance table for speedups because of quantization.	
Data Type	Time (s)
float32	0.006926
int8	0.000266
Speedup	26.06x

As demonstrated, quantization provides significant improvements in both memory footprint and computational speed. For more details, refer to the associated notebook for this chapter.

10.4.3 Experiment Results with Linear Quantization on Distilled FinBERT

In this section, we apply linear quantization to the entire Distilled FinBERT model and evaluate the improvements in memory storage and computational speed, along with any potential loss in accuracy due to representing the model's weights with lower precision. (As a side note, in these experiments, quantization is applied only during the inference process.) Since at the time of writing PyTorch currently supports linear quantization only on CPU devices, the results presented here are for CPU-based inference. The results are summarized in [Table 10.5](#).

TABLE 10.5**Inference speed after quantization.**

Model	Accuracy (%)	Average Latency per Sentence (s)
FinBERT	83.9175	0.1533
DistilledFinBERT	85.3608	0.1060
QuantizedDistilledFinBERT	83.7113	0.0458

As shown, the quantization process introduces some loss in accuracy, with the model's performance dropping from 85.36% in the distilled version to 83.71% in the quantized version. However, the quantized Distilled FinBERT still matches the original FinBERT's accuracy (83.92%), with only a slight difference of 0.2%. In contrast, the quantized model achieves a significant speedup, performing inference 3.35 times faster than the original model.

This result demonstrates the power of efficient inference: maintaining nearly identical performance while delivering significantly faster predictions. Details on how to apply quantization to the model are provided in the accompanying chapter notebook.

10.5 Customizing Your LLM: Adapting Models to Your Needs

LLMs are incredibly powerful tools that can speed up many kinds of tasks. In earlier chapters, we showed how they can help generate code for trading strategies or assist in developing trading ideas. However, as we have seen, getting satisfactory answers often requires extensive prompt engineering and adding extra context to guide the model to produce relevant outputs.

Sometimes, though, you might already have a lot of useful information, like a codebase or internal documentation, that you would like the LLM to access directly. By customizing the LLM to work with this knowledge, you can reduce the amount of prompt engineering and get answers that are much more tailored to your specific needs. For example, this would allow software engineers, quant researchers, or discretionary traders to interact with a knowledge base more effectively, getting responses that are more relevant and aligned with their work.

As of the time of writing, there are two main ways to customize an LLM to work with your own data. One involves fine-tuning, also known as post-training, which as we have seen involves retraining the model with your specific data to make it better for your tasks. The other approach does not require retraining the LLM but instead enriches the user's prompt by appending extra information as part of the input, allowing the LLM to use that data to answer specific questions.

A popular method in this second category is Retrieval Augmented Generation (RAG). In RAG, contextual information is automatically retrieved from an external database and appended to the user's prompt, helping the LLM generate more relevant or "grounded" responses.

The contextual information is dynamically retrieved from the database by measuring the similarity between the user's input prompt and the files in the database. The main idea is to represent both the files in the database and the user's prompt as mathematical objects, such as vectors (embeddings). Once we have these representations, a similarity measure can be applied to identify and retrieve the most relevant files for the input prompt. For more information about RAG, please refer to the OpenAI blog post Retrieval-augmented Generation (RAG) and Semantic Search for GPTs at <https://help.openai.com/en/articles/8868588->

[retrieval-augmented-generation-rag-and-semantic-search-for-gpts.](#)

In the next section, we will provide a brief overview of techniques for fine-tuning (or post-training) to customize your LLMs. While we will not delve into the technical details—that is beyond the scope of this book—we will cover the key concepts and focus on one specific technique to build a practical example you can try yourself.

10.5.1 Fine-tuning Techniques

10.5.1.1 Traditional Fine-tuning (FT).

FT involves retraining all the parameters of the model, which can be computationally expensive and time-consuming as models scale. Additionally, it comes with the risk of catastrophic forgetting, where the model loses a considerable part of the knowledge gained during pre-training, leading to degraded generalization performance.

To address these challenges, academia and industry have developed techniques that involve adapting only a subset of a model's parameters during training. These techniques fall under the category of parameter-efficient fine-tuning (PEFT), making models faster to adapt, more memory-efficient, and capable of achieving comparable or even better performance than traditional fine-tuning.

10.5.1.2 Parameter-efficient Fine-tuning (PEFT).

PEFT focuses on adapting a subset of model's parameters during fine-tuning, which reduces computational requirements, storage needs, and training time compare with FT. Following are some key PEFT methods.

10.5.1.2.1 BitFit.

BitFit updates only the bias terms of the network rather than all the model’s parameters; see Zaken et al. (2021). The authors have shown that applying BitFit to BERT-like architectures—models we’ve explored in this and the previous chapter—achieves performance comparable to full fine-tuning on small-to-medium datasets. Remarkably, for models like BERTBase and BERTLarge, the bias terms represent less than 0.1% of the model’s parameters, yet modifying only this small percentage appears to be enough for such datasets. For more details, refer to Zaken et al. (2021).

10.5.1.2.2 Adapters.

Adapters (Houlsby et al., 2019) introduce trainable layers into the network. During adaptation, only these newly added layers are updated, while the original model parameters remain frozen. Authors have shown that adapters achieve near state-of-the-art performance at the time of publication. However, adding new layers to the network increases both storage requirements and inference latency. More details can be found in Houlsby et al. (2019).

10.5.1.2.3 Prompt-tuning.

Prompt-tuning involves enriching the input prompt with a learnable prompt to help the LLM perform well on a downstream task, such as computing the sentiment of a given text, see the paper *The Power of Scale for Parameter-efficient Prompt Tuning* by Lester et al. (2021). The authors of this paper show that prompt-tuning achieves accuracy

comparable to traditional fine-tuning as the model size increases. An extension of prompt-tuning is prefix-tuning, introduced in the paper *Prefix-tuning: Optimizing Continuous Prompts for Generation* by Li and Liang (2021), which adds learnable prompts to each layer of the transformer. However, this method introduces additional inference latency due to the learnable prompts and reduces the available input prompt length.

10.5.1.3 LoRA (Low-rank Adaptation of Large Language Models).

LoRA, see Hu et al. (2021), is similar to adapters in the sense that it introduces additional learnable layers, but with the difference that it does so as a parallel branch (see [Figure 10.7](#)), which provides the advantage of not increasing inference latency. Like adapters, LoRA keeps the original network parameters fixed and trains additional parameters, which are then combined with the original network via addition.

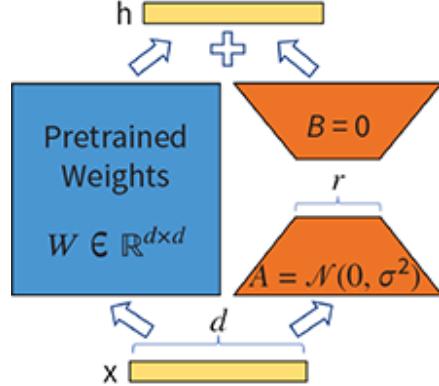


Figure 10.7 Figure 1 from Hu et al. (2021): This figure illustrates the reparameterization of new layers added as a parallel branch using LoRA. Only the matrices A and B are trained, while pretrained weights W are not updated. The figure also shows the initialization of matrices A and B .

For example, following the terminology of the paper, consider a dense layer in the network with parameters W . If the input to this layer is x , the output is denoted as $h = Wx$. In LoRA, an additional layer, ΔW , is introduced as a parallel branch. The output of LoRA then becomes:

$$h = Wx + \Delta Wx$$

During training, the parameters of W remain fixed (frozen), while the parameters of ΔW are adapted.

The key aspect of LoRA is that the matrix ΔW is constrained to be a low-rank matrix, defined as $\Delta W = BA$. For example, if W has dimensions $d \times k$, ΔW must also have dimensions $d \times k$. However, B and A have dimensions $d \times r$ and $r \times k$, respectively, where $r \ll \min(d, k)$. This, in LoRA, the forward pass is modified as follows:

$$h = Wx + \Delta Wx = Wx + BAx = (W + BA)x = W'x$$

So, once the weights B and A are learned and combined with the original network to form W' , no extra latency is introduced at inference time because W and W' have the same dimensions. Authors in Hu et al. (2021), propose to initialize A with random Gaussian values and B with zeros, so making sure that $\Delta W = BA$ starts as zero at the beginning of training (see [Figure 10.7](#)).

10.5.1.3.1 Efficiency of LoRA.

Authors have shown that for large models like GPT-3 (175B parameters), a very low rank r , such as 1 or 2, suffices even when the full rank d is as high as 12,288. This makes LoRA highly storage and compute efficient.

In their experiments, the authors demonstrated that, compared to fine-tuning GPT-3 175B with Adam optimizer, LoRA reduces the number of trainable parameters by 10,000 times and the GPU memory requirements by three times. See Hu et al. (2021) for more details on the results and experiments.

10.5.1.3.2 QLoRA.

QLoRA, see Dettmers et al. (2024), extends LoRA by incorporating quantization, making it even more storage and compute efficient. More details about QLoRA can be found at Dettmers et al. (2024).

Fortunately for us, most of these PEFT techniques have already been implemented by the Hugging Face team. You can learn more in their blog post announcing the release of the PEFT library: Parameter-Efficient Fine-Tuning of Billion-Scale Models on Low-Resource Hardware at

<https://huggingface.co/blog/peft>. For a more in-depth exploration, visit the PEFT GitHub repository at <https://github.com/huggingface/peft>, which provides implementations of state-of-the-art PEFT methods fully compatible with the Hugging Face Library. Also, if you want to learn more about efficient deep learning, check out the MIT course *TinyML and Efficient Deep Learning Computing* at <https://hanlab.mit.edu/courses/2024-fall-65940>. This excellent resource covers more topics than we can address in a single chapter.

As an example, in the accompanying notebook for this chapter, we will illustrate how to fine-tune the Llama 3 model using LoRA. This approach enables can be extended by you to develop various applications, such as customized Q&A systems for internal use or customer-facing applications.

10.5.1.4 Aligning Your LLM with Human Preferences.

Another key aspect of fine-tuning is aligning LLM responses with human preferences. After the initial pre-training phase, model outputs can sometimes be unhelpful or even toxic.

Techniques like **Reinforcement Learning from Human Feedback** (RLHF), introduced by Ouyang et al. (2022), help language models generate responses that are more human-like, creative, truthful, and helpful. RLHF also helps to reduce issues like biased or inaccurate content and was one of the major advancements used by OpenAI that transformed GPT-3, into the remarkable ChatGPT that we all love.

A newer approach, direct preference optimization (DPO), simplifies the RLHF process while improving stability, performance, and computational efficiency. For more details about DPO see Rafailov et al. (2024).

Interestingly, PEFT methods—like the ones we talked about earlier—can also be used with RLHF and DPO, making fine-tuning more efficient.

These techniques are important for customizing LLMs to specific use cases. However, since they are beyond the scope of this book—at least in this edition—we will not go into the details. That being said, they are worth keeping an eye on as this field continues to evolve.

For a practical example of applying PEFT techniques with RLHF, we encourage readers to check out the Hugging Face blog post, Fine-tuning 20B LLMs with RLHF on a 24GB consumer GPU at <https://huggingface.co/blog/trl-peft>, which can be especially useful if you have limited computational resources—basically, if you don't have a cluster of GPUs just available to you to fine-tune your models.

10.6 Conclusions

In this chapter, we highlighted the importance of model scaling for improving performance, not only in a wide range of downstream tasks but also in unlocking emergent abilities. However, as demonstrated, scaling models come with trade-offs, such as increased latency and higher deployment costs. These challenges necessitate modern solutions for efficient production deployment, such as knowledge distillation and model quantization.

We showcased the effectiveness of these techniques using a real-world case study with FinBERT, a BERT-based model specialized for sentiment analysis in financial data. Through distillation and quantization, we achieved a nearly threefold increase in inference speed while maintaining the same level of accuracy.

Additionally, we explored techniques for customizing your own LLM to make it more tailored to your specific data. Among these, we delved into LoRA and QLoRA, which, at the time of writing, stand out as some of the most efficient methods for fine-tuning LLMs. We also briefly mentioned the

two major techniques currently used to align LLM outputs with human preferences, which have been crucial in the development of widely adopted LLM solutions like OpenAI's ChatGPT. These techniques are versatile and can be applied across various models, offering powerful solutions for optimizing performance and inference costs in real-world applications.

Chapter 11

Afterword

In [Part 1](#) the book, we demonstrated how LLMs can assist with a no-code approach to developing trading applications, such as helping traders with code generation or generating ideas for trading strategies. While these tools can significantly accelerate the development process, like in typical software engineering tasks in other fields, we found that in financial applications at least, domain expertise is still crucial. Users need a solid understanding of their goals to generate meaningful trading strategies. LLMs can serve as copilots, assisting with coding tasks and clarifying ideas, but their outputs must be critically evaluated by a knowledgeable user. Techniques like Retrieval Augmented Generation (RAG) can enhance LLMs by grounding their responses in specific content, but when it comes to generating trading strategies, human insight and domain expertise remain indispensable.

We then transitioned from the no-code approach to a whirlwind tour of the modeling techniques that are well known to quantitative traders and hedge fund managers. We explored the advantages and limitations of unsupervised and supervised learning, tree-based and neural-network-based methods, RNNs, and Reinforcement Learning, highlighting the importance and nuances of data and feature engineering. On the way, we also discussed training methods such as stochastic gradient descent that are widely used in deep learning.

In [Part 2](#), we explore the power of Deep Generative Models and their applications in trading and asset management. These models excel at handling complex, multimodal, high-

dimensional data. First, we explained the difference between discriminative vs generative models. Then, we examined several families of generative models, some of which possess core abilities like generating new data, computing probability densities, and discovering new representations, though not all model families are capable of performing all these tasks, with some excelling in specific areas like data generation, probability density approximation, or representation learning. As a reminder, representation learning is all about finding a new representation for the data space, typically referred to as the latent space. Ideally, this latent representation should possess desirable properties, such as a lower dimensionality compared to the data space or “independent” axes of variation. The goal is for this new data representation to serve as a compressed form of the data and that can be used for solving downstream tasks, such as prediction, anomaly detection, or data generation. The Variational Autoencoder (VAE) is an example of a model that provides such a representation. These capabilities are crucial for developing applications in fields like finance, where high-dimensional, multimodal complex data—such as multivariate time series—are common, and where uncertainty quantification is critical for making informed decisions.

We discussed the pros and cons of both explicit and implicit deep generative models for high-dimensional data. Remember that explicit models propose a tractable probability density model to approximate the data distribution, in contrast to implicit models, which do not define a probability density function. Examples of explicit models discussed in this book include Autoregressive Models, the well-known VAE and Flow Models. For implicit models, we discussed Generative Adversarial Networks

(GANs). These models have valuable applications in forecasting and risk management.

These techniques, which allow you to model high-dimensional multimodal data, are not just simple extensions of classical parametric models found in classical statistics, such as regression models under Gaussian or t-distribution assumptions. Also, due to the large number of parameters involved to model complex data distributions, and the amount of data required for training them, these models are designed not only to achieve good performance, but also to be efficient at training time, inference time, and sample generation. Sometimes, achieving all these benefits simultaneously in a single solution is very challenging. However, techniques have been developed to improve specific aspects, such as the one described in [Chapter 10](#) to improve inference speed.

We illustrated their potential in finance through use cases such as multivariate time-series forecasting and synthetic multivariate time-series generation. We emphasize that despite being called “Generative AI,” as previously mentioned, the proposed models are not just for generating synthetic multivariate time series. They can also be used for forecasting these series or as powerful building blocks for solving a variety of downstream tasks, like the use case presented in [Chapter 9](#). These downstream tasks may even be trained with data augmentation, using synthetic data generated by these models to improve performance, or be used to manage risk by simulating new scenarios based on this data.

In the final chapters, [Chapters 9](#) and [10](#) we discussed how these models can be used to process alternative data and generate trading signals, showing their flexibility and potential in generating alternative features and trading strategy development. In particular, we explored how to

build a system that takes speech data as input and outputs trading signals. This system uses transformer models to convert audio signals to text and applies LLMs to assign sentiment scores to text, which are then used to generate trading signals.

Last but not least, we covered some important special topics. One of them is the modern approach of pre-training plus adaptation, which allows you to adapt or fine-tune powerful pre-trained models for your custom applications. This technique enables good performance even when we have limited data available to train our model for a specific task. We also discussed fine-tuning techniques that help improve the efficiency of the fine-tuning process.

Additionally, we covered compression techniques for optimizing model scalability, speeding up inference, reducing inference latency, and lowering the associated deployment costs, which are critical for making these models work in practical settings. Finally, we explored the scaling laws of LLMs and their emergent abilities. Scaling laws of LLMs state that increasing the number of model parameters, the amount of training data, and the training duration will lead to significant gains in performance on these models. These laws provide a straightforward way to improve performance in LLMs: if you increase your resources for data and training, you will increase model's performance. However, we still don't know if these laws will reach a saturation point where the "law" starts to break down.

Emergent abilities, such as the ability of these models to solve math problems, only appear after the model has reached a certain scale. These abilities are a surprise for practitioners and researchers, and at the time of writing, we are not able to predict when they will appear.

While no single deep generative model family is universally superior across all tasks and datasets, different families excel in different domains. For example, autoregressive models have proven highly effective for text generation, while diffusion models are, as of this writing, the leading approach for image generation. However, in data generation for financial applications—especially for generating synthetic univariate or multivariate time series—there is still no clear winner. The low signal-to-noise ratio, non-stationarity of financial data, and the relatively small datasets available for training, compared to domains like text and images, pose significant challenges. These factors make the problem more difficult, suggesting that more tailored solutions or additional empirical work need to be done for this specific use case.

Despite these challenges, generative models hold immense potential for financial applications, particularly in generating features from alternative data sources, which in the past required large teams of highly specialized talent and significant infrastructure investing. The features generated by these systems can be integrated into more advanced trading strategies, providing a competitive edge in an increasingly data-driven industry.

While we covered a wide range of topics, there are several promising areas of future exploration in finance where generative AI models could play a significant role, which we did not have the time to cover in this edition:

11.1 Diffusion Models

Diffusion Models are currently considered state-of-the-art in image generation applications and are integrated into well-known products such as DALL·E from OpenAI. These models, also known as denoising diffusion probabilistic

models (DDPMs), see Sohl-Dickstein et al. (2015) and Ho et al. (2020).

Diffusion Models take a different approach to data generation compared to the methods we have explored so far. For instance, GANs generate samples in a single step by feeding noise into the generator, which produces the output image in one go. In contrast, Diffusion Models generate data through an iterative refinement process consisting of multiple steps. This iterative procedure is considered one of the key reasons for their superior performance.

Diffusion Models are explicit models with approximate density estimation—recalling VAEs. In fact, their learning process can be seen as analogous to training only the decoder of a VAE.

These models achieve equal or better results comparable to GANs in data generation, but they offer a more stable training process and with additional advantage of performing density estimation. However, sampling from them is relatively slow.

More details about diffusion models can be found at Sohl-Dickstein et al. (2015), Ho et al. (2020), and Bishop and Bishop (2023).

Given the success of Diffusion Models in generating image data, researchers have started applying them to multiple applications, including, multivariate time-series forecasting and generation. Details can be found in the following surveys: Yang et al. (2023, 2024). We anticipate further developments in this area.

11.2 Combining Generative Model Variants

Throughout the book, we have seen that some generative model families are very good at modeling correlation between the dimensions of a single point in our datasets (e.g. cross-sectional co-dependence of returns across assets). However, by design, their structure assumes independence between consecutive observations.

Throughout the book, we have discussed various approaches to adapt these models to capture this structure, the time-series dynamics, which can be crucial when modeling time series at certain time scales where dependencies between observations are significant.

Some of these solutions involve combining different model families, which, as we've mentioned, can serve as powerful building blocks when used together. For example:

- Combining Autoregressive Models with VAEs: This was achieved by adding sequential encoders and decoders to VAEs, giving rise to models like the Variational Recurrent Autoencoder (VRAE), as described in [Chapter 6](#).
- Combining Autoregressive Models with Flow Models: In [Chapter 7](#), we explore the combination of RNNs and Transformer models with flows.
- Combining Autoregressive Models with GANs: In [Chapter 8](#), we examine using RNNs with GANs, resulting in models such as RCGAN or TimeGAN. TimeGAN, in particular, employs a sequence generator and discriminator to model time-series dynamics in the latent space.

These are some of the variants discussed throughout the book. However, there are well-known variations used in other fields that we did not cover. A few of them consists on combining flows with VAEs (see Kingma et al. [2016]); GANs with VAEs, called InfoGAN (see Chen et al. [2016]). We expect that more variants may be developed, inspired, or applied in the context of time series.

11.3 LLMs as Financial Advisors

Similar to how large language models (LLMs) are being tested on medical university examinations to evaluate their potential as doctor assistants, Lo and his team are exploring whether LLMs can demonstrate the same domain knowledge humans exhibit when passing typical financial examinations, such as the CFA test; see <https://mitsloan.mit.edu/ideas-made-to-matter/can-generative-ai-provide-trusted-financial-advice>. The goal of this approach is to assess whether LLMs are capable of providing relevant financial advice.

According to Lo, his team's research shows promising results thus far. He stated, "We're actually able to generate passing domain-specific knowledge among large language models," as reported in the cited MIT Sloan web article. We expect to see a growing trend in developing LLMs that are specific to finance, for example, whether as advisors or, as demonstrated in [Chapter 9](#), by fine-tuning LLMs to perform sentiment analysis specific to financial contexts, among others.

This is a rapidly evolving field, with new approaches and advancements being released in an almost daily basis. In this book, our goal was to provide a solid understanding of the core building blocks of generative models, enabling you to navigate, analyze, innovate, and apply these innovations

in trading and asset management. We have sprinkled code samples throughout the book. They can be downloaded from the book's GitHub repository:

<https://github.com/genai-for-traders/genai-for-trading-and-am>. Also, we welcome reader's questions and comments on our Twitter/X account (**echanQT**, and **hamletjmedina**) and at our blog <https://substack.com/@gatambook>. We are sure that by the time this book is published, we have already gained better understanding and better tools in applying AI to finance, and we will post our latest learnings to those media.

References

- Abbeel, P., Chen, P., Ho, J., and Sriniva, A. (2020). Lecture slides on deep unsupervised learning, CS294-158-SP20. Course, University of California, Berkeley.
<https://sites.google.com/view/berkeley-cs294-158-sp20/home>.
- Alshahwan, N., Chheda, J., Finegenova, A., Gokkaya, B., Harman, M., Harper, I., et al. (2024). *This automated unit test improvement using large language models at Meta*. <https://arxiv.org/abs/2402.09171>.
- Ang, A. (2014). *Asset management: A systematic approach to factor investing*. Oxford University Press.
- Ansari, A.F., Stella, L., Turkmen, C., Zhang, X., Mercado, P., et al. (2024). *Chronos: Learning the language of time series*. <https://arxiv.org/abs/2403.07815>.
- Araci, D. (2019). *FinBERT: Financial sentiment analysis with pre-trained language models*.
<https://arxiv.org/abs/1908.10063>.
- Arik, S. and Pfister, T. (2020). *TabNet: Attentive interpretable tabular learning*.
<https://arxiv.org/abs/1908.07442>.
- Arjovsky, M., Chintala, S., and Bottou, L. (2017). *Wasserstein GAN*. <https://arxiv.org/abs/1701.07875>.
- Baldridge, R. (2024). *Top 10 U.S. hedge funds of December 2024*. <https://www.forbes.com/advisor/investing/top-hedge-funds/>.

- Belov, S., Chan, E., Jetha, N., and Nautiyal, A. (2023). *Applying corrective AI to daily seasonal Forex trading*.
<https://predictnow.ai/applying-corrective-ai-to-daily-seasonal-forex-trading-paper/>.
- Bishop, C.M. (2006). *Pattern recognition and machine learning (Information science and statistics)*. Berlin, Heidelberg: Springer-Verlag.
- Bishop, C.M. and Tipping, M.E. (2001). Probabilistic principal component analysis. *Journal of the Royal Statistical Society, Series B*, 61, part 3, pp. 611-622.
- Bishop, C. M. and Bishop, H. (2023). *Deep learning: Foundations and concepts*. Springer Nature.
- Bloomberg News. (2023). *JPMorgan AI analyzes Fed speeches for trading signals*.
<https://www.bloomberg.com/news/articles/2023-04-26/jpmorgan-s-ai-puts-25-years-of-federal-reserve-talk-into-a-hawk-dove-score>.
- Bodie, Z., Marcus, A.J., and Kane, A. (2018). *Investments*. New York: McGraw-Hill/Irwin.
- Botte, A. and Bao, Ds. (2021). *A machine learning approach to regime modeling*.
<https://www.twosigma.com/articles/a-machine-learning-approach-to-regime-modeling/>.
- Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software.
- Brubaker, M. A. and Köthe, U. (2021). *Normalizing flows and invertible neural networks in computer vision*. CVPR 2021 Tutorial.

- Brukbaker, B. (2024). *How chain-of-thought reasoning helps neural networks compute.*
<https://www.quantamagazine.org/how-chain-of-thought-reasoning-helps-neural-networks-compute-20240321/>.
- Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E. et al. (2023). *Sparks of Artificial General Intelligence: Early experiments with GPT-4.*
<https://arxiv.org/abs/2303.12712>.
- Catello, L., Ruggiero, L., Schiavone, L., and Valentino, M. (2023). *Hidden Markov models for stock market prediction.* <https://arxiv.org/pdf/2310.03775.pdf>.
- Chan, E. (2013). *Algorithmic trading: Winning strategies and their rationale.* New York: Wiley.
- Chan, E. (2021). *Quantitative trading: How to build your own algorithmic trading business*, 2nd ed. New York: Wiley.
- Chan, E. (2022). *What is Corrective AI?*
<https://predictnow.ai/what-is-corrective-ai/>.
- Chan, E., Fan, H., Sawal, S., and Viville, Q. (2023). *Conditional portfolio optimization: Using machine learning to adapt capital allocations to market regimes.*
<https://ssrn.com/abstract=4383184>.
- Chan, E.P. (2017). *Machine trading: Deploying computer algorithms to conquer the markets.* New York: Wiley.
- Chen, R.T.Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D.K. (2018). Neural Ordinary differential equations.
Advances in Neural Information Processing Systems, 31.
- Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable

representation learning by information maximizing generative adversarial nets. *Advances in Neural Information Processing Systems*, 29.

Claburn, T. (2024). *Simon Willison interview: AI software still needs the human touch.*

https://www.theregister.com/2024/01/24/willison_ai_software_development/?utm_source=tldrnewsletter.

Cong, L. W., Feng, G., He, J., and He, X. (2024). Growing the Efficient Frontier on Panel Trees. *Journal of Financial Economics*, Forthcoming,
<https://ssrn.com/abstract=3949463>.

Cong, L. W., Tang, K., Wang, J., and Zhang, Y. (2021). AlphaPortfolio: *Direct construction through deep reinforcement learning and interpretable AI.*
<https://ssrn.com/abstract=3554486>.

Denev, A. and Amen, S. (2020). *The book of alternative data: A guide for investors, traders and risk managers.* New York: John Wiley & Sons.

Deng, R., Chang, B., Brubaker M.A., Mori, G., and Lehrmann, A. (2020). Modeling continuous stochastic processes with dynamic normalizing flows. *Advances in Neural Information Processing Systems*, 33, pp. 7805-7815.

Desai, A., Freeman, C., Wang, Z., and Beaver, I. (2021). *Timevae: A variational auto-encoder for multivariate time series generation.* arXiv Preprint arXiv:2111.08095.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2024). Qlora: Efficient finetuning of quantized LLMS. *Advances in Neural Information Processing Systems*, 36.

Devlin, J., Chang M.-W., Lee, K., and Toutanova, K. (2019). *BERT: Pre-training of deep bidirectional transformers for language understanding.*
<https://arxiv.org/abs/1810.04805>.

Dinh, L., Krueger, D., and Bengio, Y. (2015). *NICE: Non-linear independent components estimation.*
<https://arxiv.org/abs/1410.8516>.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). *Density estimation using real NVP.*
<https://arxiv.org/abs/1605.08803>.

Dixon, M. F., Halpern, I., and Bilokon, P. (2020). *Machine learning in finance: From theory to practice*. New York: Springer.

Esteban, C., Hyland, S.L., and Rätsch, G. (2017). *Real-valued (medical) time series generation with recurrent conditional GANS*. arXiv Preprint arXiv:1706.02633.

Fabius, O. and Van Amersfoort, J.R. (2014). *Variational recurrent auto-encoders*. arXiv Preprint arXiv:1412.6581.

Fama, E. F. and French, K. R. (1995). Size and book-to-market factors in earnings and returns. *Journal of Finance*, 50, pp. 131–155.

Fortune. (2023). *JPMorgan created a Fed-whispering AI model to help investors stay on top of the market.*
<https://fortune.com/2023/04/28/jpmorgan-artificial-intelligence-chatgpt-llm-hawk-dove-federal-reserve-speak/>.

Frey, B.J., Hinton, G.E., and Dayan, P. (1995). Does the wake-sleep algorithm produce good density estimators? *Advances in Neural Information Processing Systems*, 8.

- Führer, B., Tessler, C., and Dala, G. (2024). *Gradient Boosting reinforcement learning*.
<https://arxiv.org/abs/2407.08250>.
- Gatheral, J. (2006). *The volatility surface: A practitioner's guide*. New York: Wiley.
- Germain, M., Gregor, K., Murray, I., and Larochelle, H. (2015). *MADE: Masked Autoencoder for Distribution Estimation*. <https://arxiv.org/abs/1502.03509>.
- Grinsztajn, L., Oyallon, E., and Varoquaux, G. (2022). *Why do tree-based models still outperform deep learning on tabular data?* <https://arxiv.org/abs/2207.08815>.
- Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT Press.
- Goodfellow, I.J. Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Sherjil et al. (2014). *Generative adversarial networks*. <https://arxiv.org/abs/1406.2661>.
- Goodfellow, I. (2019). *Adversarial machine learning*. Invited talk at ICLR.
- Gou, J., Baosheng, Y., Maybank, S.J., and Tao, D. (2021). Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6), pp. 1789–1819.
<https://doi.org/10.1007/s11263-021-01453-z>.
- Guida, Ty. (2019). *Big data and machine learning in quantitative investment*. New York: John Wiley & Sons.
- Guizarro-Ordonez, J., Pelger, M., and Zanotti, G. (2022). *Deep learning statistical arbitrage*.
<https://arxiv.org/abs/2106.04028>.

- Guizarro-Ordonez, J., Pelger, M., and Zanotti, G. (2021). *Deep learning statistical arbitrage*. arXiv Preprint arXiv:2106.04028.
- Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). *Improved training of Wasserstein GANs*. <https://arxiv.org/abs/1704.00028>.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The elements of statistical learning*. 2nd ed. New York: Springer.
- Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2018). *GANs trained by a two time-scale update rule converge to a local Nash equilibrium*. <https://arxiv.org/abs/1706.08500>.
- Hinton, G., Vinyals, O., and Dean, J. (2015). *Distilling the knowledge in a neural network*. <https://arxiv.org/abs/1503.02531>.
- Ho, J., Jain, A., and Abbeel, P. (2020). Denoising diffusion probabilistic models. *Advances in Neural Information Processing Systems*, 33, pp. 6840–6851.
- Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5), pp. 359–366.
- Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., et al. (2019). Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, pp. 2790–2799. PMLR.
- Hu, E.J., Shen, Y., Wallis, P. Allen-Zhu, Z., Li, Y., Wang, S. et al. (2021). *LORA: Low-rank adaptation of large language models*. arXiv Preprint arXiv:2106.09685.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. et al. (2017). *Quantization and training of neural networks for efficient integer-arithmetic-only inference.* <https://arxiv.org/abs/1712.05877>.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., Chess, B., Child, R. et al. (2020). *Scaling laws for neural language models.* <https://arxiv.org/abs/2001.08361>.

Kıcıman, E., Ness, R., Sharma, A., and Tan, C. (2023). *Causal reasoning and large language models: Opening a new frontier for causality.* [https://arxiv.org/abs/\(2305\).00050](https://arxiv.org/abs/(2305).00050).

Kingma, D.P. (2013). *Auto-encoding variational Bayes.* arXiv Preprint arXiv:1312.6114.

Kingma, D.P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., and Welling, M. (2016). Improved variational inference with inverse autoregressive flow. *Advances in Neural Information Processing Systems*, 29.

Kinlaw, W., Kritzman, M.P., and Turkington, D. (2021). *Asset allocation: From theory to practice and beyond.* New York: John Wiley & Sons.

Kobyzev, I., Prince, S.J.D., and Brubaker, M.A. (2021). Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43 (11), pp. 3964–3979. [https://doi.org/10.1109/tpami.\(2020\).2992934](https://doi.org/10.1109/tpami.(2020).2992934).

Kolm, P., Turiel, J., and Westray, N. (2021). *Deep order flow imbalance: Extracting alpha at multiple horizons from the Limit Order Book.* SSRN: <https://ssrn.com/abstract=3900141>.

- Larochelle, H. and Murray, I. (2011). The neural autoregressive distribution estimator. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 29–37. JMLR Workshop; Conference Proceedings.
- Lester, B., Al-Rfou, R., and Constant, N. (2021). *The power of scale for parameter-efficient prompt tuning*. arXiv Preprint arXiv:2104.08691.
- Levine, S. (2021). Lecture slides on designing, visualizing and understanding deep neural networks, CS W182/282A. Course, University of California, Berkeley. <https://cs182sp21.github.io>.
- Li, X.L. and Liang, P. (2021). *Prefix-tuning: Optimizing continuous prompts for generation*. arXiv Preprint arXiv:2101.00190.
- Liu, S., Yu, H., Liao, C., Li, J., Lin, W., Liu, A.X. et al. (2021). Pyraformer: Low-complexity pyramidal attention for long-range time series modeling and forecasting. In *International Conference on Learning Representations*.
- López de Prado, M. (2015). The probability of backtest overfitting. *Journal of Computational Finance (Risk Journals)*, 34, pp. 123–137.
- López de Prado, M. (2018). *Advances in financial machine learning*. New York: Wiley.
- López de Prado, M.M. (2020). *Machine learning for asset managers. Elements in quantitative finance*. Cambridge University Press.
- Malo, P., Sinha, A., Korhonen, P., Wallenius, J., and Takala, P. (2014). Good debt or bad debt: Detecting semantic orientations in economic texts. *Journal of the Association*

for Information Science and Technology, 65 (4), pp. 782-796.

Man, X. and Chan, E. P. (2021a). The best way to select features? Comparing MDA, LIME, and SHAP. *The Journal of Financial Data Science*, 3(1), pp. 127-139.

Man, X. and Chan, E. (2021b). Cluster-based feature selection. *Market Technician*, 90, pp. 11-22.

Mao, X., Li, Q., Xie, H., Lau, R.Y.K., Wang, Z., and Smolley, S.P. (2017). *Least squares generative adversarial networks*. <https://arxiv.org/abs/1611.04076>.

Medina Ruiz, H. J. (2019). Predicting Criteo's internet traffic load using Bayesian structural time series models. Presented at the Strata Data Conference.

<https://conferences.oreilly.com/strata/strata-ny-2019/public/schedule/detail/77395.html>.

Mescheder, L., Andreas G., and Sebastian N. (2018). Which training methods for GANs do actually converge? <https://arxiv.org/abs/1801.04406>.

Miyato, T., Kataoka, T., Koyama, M., and Yoshida, Y. (2018). *Spectral normalization for generative adversarial networks*. <https://arxiv.org/abs/1802.05957>.

Murphy, K. (2012). *Machine learning: A probabilistic perspective*. Boston, MA: MIT Press.

Murphy, K.P. (2022). *Probabilistic machine learning: An introduction*. Boston, MA: MIT Press. probml.ai.

Murphy, K.P. (2023). *Probabilistic machine learning: Advanced topics*. Boston, MA: MIT Press.
<http://probml.github.io/book2>.

Nautiyal, A. and Chan, E. (2021). *Metalabeling and the duality between cross-sectional and time-series factors*. Retrieved from <https://predictnow.ai/metalabeling-and-the-duality-between-cross-sectional-and-time-series-factors/>.

Nalisnick, E., Matsukawa, A., Teh, Y.W., Gorur, D., and Lakshminarayanan, B. (2019). Hybrid models with deep and invertible features. In *International Conference on Machine Learning*, pp. 4723-4732. PMLR.

Nguyen, A., Yosinski, J., and Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 427-436.

Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A. et al. (2016a). *WaveNet: A generative model for raw audio*. <https://arxiv.org/abs/1609.03499>.

Oord, A., Kalchbrenner, N., and Kavukcuoglu, K. (2016b). *Pixel recurrent neural networks*.
<https://arxiv.org/abs/1601.06759>.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P. et al. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, pp. 27730-27744.

Paleologo, G.A. (2021). *Advanced portfolio management: A quant's guide for fundamental investors*. New York: Wiley.

Papamakarios, G., Nalisnick, E., Rezende, D.J., Mohamed, S., and Lakshminarayanan, B. (2021). Normalizing flows

for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57), pp. 1-64.

Papamakarios, G., Pavlakou, T., and Murray, I. (2017). Masked autoregressive flow for density estimation. *Advances in Neural Information Processing Systems*, 30.

Part Time Larry. (2022). 2. *OpenAI whisper—Fed speech recognition*. <https://www.youtube.com/watch?v=Wc4bQxuypo0>.

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K. et al. (2018). *Deep contextualized word representations*. [https://arxiv.org/abs/\(1802\).05365](https://arxiv.org/abs/(1802).05365).

Pik, J., Chan, E., Broad, J. and Sun, P. (2025) *Hands-On AI Trading with Python, QuantConnect and AWS*. New York: Wiley.

Prado, R. and West, M. (2010). *Time series: Modeling, computation, and inference*. Chapman; Hall/CRC.

Radford, A., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P. et al. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), p. 9.

Radford, A., Kim, J.W., Xu, T., Brockman, G., McLeavey, C., Sutskever, I. (2022). *Robust speech recognition via large-scale weak supervision*.
[https://arxiv.org/abs/\(2212\).04356](https://arxiv.org/abs/(2212).04356).

Radford, A. Narasimhan, K., Salimans, T., Sutskever, I. (2018). *Improving language understanding by generative pre-training*.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C.D., Ermon, S., and Finn, C. (2024). Direct preference optimization: your language model is secretly a reward

model. *Advances in Neural Information Processing Systems*, 36.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M. et al. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140), pp. 1–67.

Rasul, K., Sheikh, A-S., Schuster, I., Bergmann, U.M., and Vollgraf, R. (2021). Multivariate probabilistic time series forecasting via conditioned normalizing flows. In *International Conference on Learning Representations*. <https://openreview.net/forum?id=WIGQBFuVRv>.

Rasul, K., Ashok, A., Williams, A.R., Ghonia, H., Bhagwatkar, R., Khorasani, A. et al. (2024). *Lag-Llama: Towards foundation models for probabilistic time series forecasting*. [https://arxiv.org/abs/\(2310\).08278](https://arxiv.org/abs/(2310).08278).

Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., and Bengio, Y. (2015). *FitNets: Hints for thin deep nets*. [https://arxiv.org/abs/\(1412\).6550](https://arxiv.org/abs/(1412).6550).

Ruppert, D., and Matteson, D. S. (2015). *Statistics and data analysis for financial engineering with R examples*. 2nd ed. New York: Springer.

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). *Improved techniques for training GANs*. [https://arxiv.org/abs/\(1606\).03498](https://arxiv.org/abs/(1606).03498).

Shwartz-Ziv, R. and Armon, A. (2021). *Tabular data: Deep learning is not all you need*. <https://arxiv.org/abs/2106.03253>.

Simon, D.P. and Campasano, J. (2014). *The VIX futures basis: evidence and trading strategies*. <https://www.efmaefm.org/0EFMAMEETINGS/EFMA%20>

[ANNUAL%20MEETINGS/2013-
Reading/papers/EFMA2013_0164_fullpaper.pdf.](#)

Sirignano, J. and Cont, R. (2018). *Universal features of price formation in financial markets: Perspectives from deep learning*. <https://arxiv.org/pdf/1803.06917.pdf>.

Smith, L. N. (2017, March). *Cyclical learning rates for training neural networks*. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464-472. IEEE.
<https://doi.org/10.1109/WACV.2017.58>.

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., and Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics. In *International Conference on Machine Learning*, 2256-2265. PMLR.

Sood, S., Papasotiriou, K., Vaiciulis, M., and Balch, T. (2023). Deep reinforcement learning for optimal portfolio allocation: A comparative study with mean-variance optimization. https://icaps23.icaps-conference.org/papers/finplan/FinPlan23_paper_4.pdf.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1), pp. 1929-1958.

Torralba, A., Isola, P., and Freeman, W.T. (2024). *Foundations of computer vision*. Boston, MA: MIT Press.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T. et al. (2023). *LLaMA: Open and efficient foundation language models*. arXiv. arXiv Preprint arXiv:2302.13971.

Tunstall, L., Von Werra, L., and Wolf, T. (2022). *Natural language processing with transformers*. O'Reilly Media, Inc.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., and Polosukhin, I. (2023). *Attention is all you need*.
<https://arxiv.org/abs/1706.03762>.

Vig, J. (2019). A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 37–42. Florence, Italy: Association for Computational Linguistics. <https://doi.org/10.18653/v1/P19-3007>.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F. et al. (2023). *Chain-of-thought prompting elicits reasoning in large language models*.
<https://arxiv.org/abs/2201.11903>.

Wei, J. Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud. S. et al. (2022). *Emergent abilities of large language models*. <https://arxiv.org/abs/2206.07682>.

Wen, Q., Zhou, T., Zhang, C., Chen, W., Ma, Z., Yan, J., and Sun, L. (2023). *Transformers in time series: A survey*.
<https://arxiv.org/abs/2202.07125>.

WGN News. (2022). *WATCH LIVE | Fed Chair Powell Discusses Latest Fed Rate Hike*.
<https://www.youtube.com/watch?v=NT2H9iyd-ms>.

Williams, R.J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, pp. 229–256.

WWW '18: Companion Proceedings of the Web Conference.
(2018). Republic; Canton of Geneva, CHE: International
World Wide Web Conferences Steering Committee.

Xue, C., Lute, J., Schonfeld, D., Han, G., and Dahlmann, L.
(2023). *How eBay created a language model with three
billion item titles*.

<https://innovation.ebayinc.com/tech/engineering/how-ebay-created-a-language-model-with-three-billion-item-titles/>.

Yang, L., Zhang, Z., Song, Y., Hong, S., Xu, R., Zhao, Y. et
al. (2023). Diffusion models: A comprehensive survey of
methods and applications. *ACM Computing Surveys*,
56(4), pp. 1-39.

Yang, Y., Jin, M., Wen, H., Zhang, C., Liang, Y., Ma, L. et al.
(2024). *A survey on diffusion models for time series and
spatio-temporal data*. arXiv Preprint arXiv:2404.18886.

Yoon, J., Jarrett, D., and van der Schaar, M. (2019). Time-
series generative adversarial networks. In H. Wallach, H.
Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R.
Garnett, Eds. *Advances in neural information processing
systems* (Vol. 32). Curran Associates, Inc.

https://proceedings.neurips.cc/paper_files/paper/2019/file/c9efe5f26cd17ba6216bbe2a7d26d490-Paper.pdf.

Zaken, E.B., Ravfogel, S., and Goldberg, Y. (2021). *Bitfit:
Simple parameter-efficient fine-tuning for transformer-
based masked language-models*. arXiv Preprint
arXiv:2106.10199.

Zhang, Z., Zohren, S., and Roberts, S. (2019). deep
reinforcement learning for trading.
<https://arxiv.org/pdf/1911.10107.pdf>.

Zhou, H., Zhang, S., Peng, J., Zhang, S., Li, J., Xiong, H. et al. (2021). *Informer: Beyond efficient transformer for long sequence time-series forecasting.*
<https://arxiv.org/abs/2012.07436>.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). *Aligning books and movies: Towards story-like visual explanations by watching movies and reading books.*
<https://arxiv.org/abs/1506.06724>.

Zhuang, Y., Liu, L, Singh, C., Shang, J., and Gao, J. (2024). *Learning a decision tree algorithm with transformers.*
<https://arxiv.org/abs/2402.03774>.

Zuckerman, G. (2019). *The man who solved the market: How Jim Simons launched the quant revolution.* Penguin Publishing Group. <https://books.google.fr/books?id=2vm0DwAAQBAJ>.

Appendix

A.1 Retrieving Adjusted Closing Prices and Computing Daily Returns

It is often a more reliable way to get correct responses from ChatGPT if we ask for codes instead of actual numerical responses. We display the results here.

User Prompt

Display the Python codes that let us retrieve 100 most recent days of adjusted closing prices of SPY from Yahoo Finance.

Assuming that you are not an expert programmer, at least not in Python, we can also prompt for instructions of installing Python, which we displayed in [Appendix A.2](#). As suggested there, we can copy and paste the following commands in Jupyter Notebook which runs in your internet browser or run it in the Spyder IDE.

ChatGPT Response

```
import yfinance as yf
import pandas as pd
# Define the ticker symbol
tickerSymbol = 'SPY'

# Get data on this ticker
tickerData = yf.Ticker(tickerSymbol)

# Get the historical prices for this ticker
tickerDf = tickerData.history(period='1d',
                                start='2023-08-25', end='2023-12-12')

# Only keep the Adjusted Close column
adjusted_close = tickerDf['Close']

# Display the last 100 days of adjusted closing
# prices
print(adjusted_close.tail(100))
```

It goes on to respond that

ChatGPT Response

This script does the following:

Imports the necessary libraries (yfinance and pandas).

Defines the ticker symbol for the SPY ETF.

Fetches the historical data for SPY from Yahoo Finance.

Extracts the Adjusted Close prices.

Prints the last 100 days of adjusted closing prices.

Make sure to adjust the start and end dates in the history method to match the current date and go back 100 days from there. The yfinance library automatically adjusts for dividends and stock splits.

Is it really true that “The yfinance library automatically adjusts for dividends and stock splits”? We have checked the first line of the output,

```
2023-08-25 00:00:00-04:00      436.653168
```

indeed matched the adjusted closing price displayed on Yahoo Finance at the time of this writing. How do we download the printed data in the browser into a csv file?

User Prompt

How do we download the data generated in the previous program into a csv file with the first column being date and second column the price?

ChatGPT Response

...

```
# Save the last 100 days of adjusted closing prices  
# to a CSV file  
adjusted_close.tail(100).to_csv('SPY_last_100_days.csv', header=True)  
print("Data saved to 'SPY_last_100_days.csv'")
```

You can check that SPY_last_100_days.csv is now created in the book4_py/ folder where you created the Python environment. Now we want to compute daily returns.

User Prompt

Great! Please display the codes that use the retrieved prices to compute daily returns.

ChatGPT Response

```
...  
# Compute daily returns  
daily_returns = adjusted_close.pct_change()  
  
# Display the last 100 days of daily returns  
print(daily_returns.tail(100))
```

Of course, we can print that to a csv file 'SPY_returns_last_100_days.csv' using the same method as above. You can easily spot check that the daily returns are computed correctly.

A.2 Installing Python

As we mentioned in [Appendix A.1](#), we wanted to use Python to retrieve historical prices. To do that, we first need to install Python on our Windows desktop, and then to install the yfinance (Yahoo Finance) package in Step 4 that follows.

(Obviously, if you are already a Python programmer, you can skip this section. Remember: We are writing this chapter from the point of view of a novice Python programmer.)

User Prompt

How do I install a python environment on my Windows desktop?

ChatGPT Response

Installing Python Installing a Python environment on your Windows desktop is a straightforward process. Here's a step-by-step guide to get you started:

A.2.1 Step 1: Download Python

1. **Visit the Official Python Website:** Go to python.org.
2. **Navigate to Downloads:** The website typically detects your operating system and displays the latest version of Python for Windows.
3. **Download Python:** Click on the download link for Windows. It's usually recommended to download the latest stable version.

A.2.2 Step 2: Install Python

1. **Run the Installer:** Once the download is complete, open the installer.
2. **Select Installation Options:**
 - **Check “Add Python to PATH”:** This is crucial as it allows you to run Python from the Command Prompt.
 - **Choose Customize Installation (Optional):** You can customize features if needed, but the default settings are usually sufficient for most users.
3. **Proceed with the Installation:** Click on “Install Now” and wait for the installation to complete.
4. **Verify Installation:**
 - Open Command Prompt.
 - Type `python --version` and press Enter. This should display the Python version, confirming that

it's installed.

A.2.3 Step 3: Set Up a Virtual Environment (Optional but Recommended)

Python's virtual environments allow you to manage separate package installations for different projects. They prevent conflicts between project dependencies.

1. **Open Command Prompt:** Navigate to the folder where you want to create your project.
2. **Create a Virtual Environment:**
 - Run `python -m venv myenv` (replace `myenv` with your desired environment name).
 - This creates a folder named `myenv` in your project directory containing the virtual environment.
3. **Activate the Virtual Environment:**
 - For Windows, run `myenv\Scripts\activate`.
 - You'll know it's activated when the command line prefix changes to show the name of your environment.

A.2.4 Step 4: Install Packages with pip

With Python installed, you can use pip to install packages:

- To install a package, use `pip install package-name`.
- Remember to activate your virtual environment first if you're using one.

[Author's Note: this should be `pip install yfinance`.]

A.2.5 Step 5: Consider an Integrated Development Environment (IDE)

While not necessary, an IDE can make coding in Python much easier. Popular choices include:

- **PyCharm**
- **Visual Studio Code**
- **Jupyter Notebook** (great for data science projects)

[Author's Note: Why it doesn't recommend Spyder that automatically comes with the Anaconda library is beyond us. For a quick and easy start, we recommend Jupyter Notebook since it can run in a web browser. How do we do that? See the next prompt.]

A.2.6 Additional Tips

- **Regularly Update Python:** Keep your Python version up-to-date for security and feature improvements.
- **Explore Python Libraries:** Python has a vast ecosystem of libraries for different purposes—web development, data analysis, machine learning, etc.
- **Learn Python Basics:** If you're new to Python, consider going through some tutorials to understand the basics.

By following these steps, you should have a functional Python environment set up on your Windows desktop.

User Prompt

Assuming that I have installed a Python Virtual Environment called book4_py on the Windows desktop, how can I install Jupyter Notebook as an IDE?

ChatGPT Response

Install Jupyter Notebook:

Once your virtual environment is active, you can install Jupyter Notebook using pip. Run the following command:

```
pip install notebook
```

This command will download and install Jupyter Notebook and its dependencies in your book4_py virtual environment.

Launch Jupyter Notebook:

After the installation is complete, you can start Jupyter Notebook by running:

```
jupyter notebook
```

You will find a new browser window opens with Jupyter notebook running. You just need to navigate to whatever folder that you want to download your adjusted closing price data and copy and paste the code in [Appendix A.1](#).

A.3 Plotting the Risk-free-rate over the Years

Many financial applications such as Sharpe ratio calculation require the use of the risk-free rate (R_f). However, ChatGPT found it difficult to locate a website to download this for free and in a convenient format. So we use the returns of the short-term treasury bills ETF called BIL as a proxy.

```

import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# Load your CSV file
file_path = 'BIL.csv' # Replace with your file path
data_plot = pd.read_csv(file_path)

# Converting 'Date' column to datetime format
data_plot['Date'] = pd.to_datetime(data_plot['Date'])

# Calculating daily returns and annualizing them
data_plot['Daily Returns'] =
data_plot['Close'].pct_change()
data_plot['Annualized Returns'] = data_plot['Daily
Returns'] * 252 # Annualizing the returns

# Plotting the annualized returns
plt.figure(figsize=(12, 6))
plt.plot(data_plot['Date'], data_plot['Annualized
Returns'], color='green', marker='o', linestyle='--',
linewidth=2)
plt.title('Annualized Returns Over Time')
plt.xlabel('Year')
plt.ylabel('Annualized Returns')
plt.grid(True)
plt.xticks(rotation=45)
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter
r('%Y')) # Only year as x-axis labels
plt.gca().xaxis.set_major_locator(mdates.YearLocator())
plt.tight_layout()

# Show plot
plt.show()

```

If you want to download the plot in [Figure A.1](#) instead of just displaying it in Jupyter Notebook, prompt ChatGPT on the instructions. It will respond with

ChatGPT Response

```
# Saving the plot as a PNG file
plot_file_path = 'your_plot.png'
...
# Save the figure
plt.savefig(plot_file_path)
```

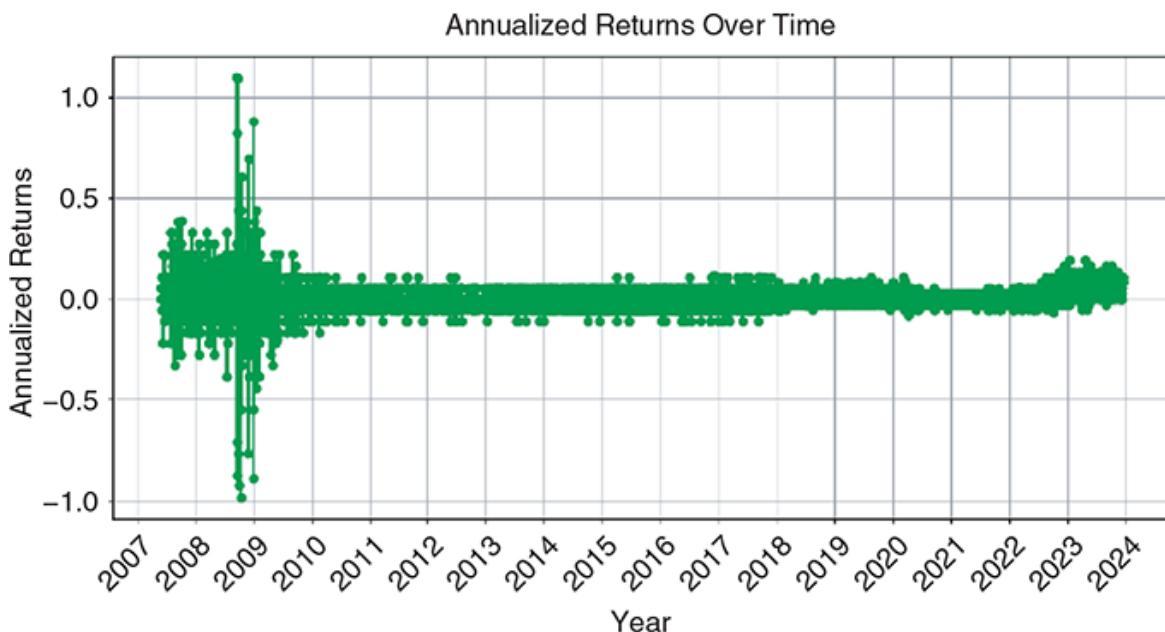


Figure A.1 BIL annualized returns.

Though the chart was plotted correctly, the daily returns show too much volatility for us to gain any insights and more importantly, check whether the returns are sensible. (After all, this is an ETF that is traded every microsecond, not the Federal Funds Rate set by the Federal Open Market Committee.) A moving average over approximately a month should look better.

User Prompt

Replot the above with 20-day moving average

Unfortunately, it couldn't for some technical reasons, but it did generate codes that we can use. The key line is

```
# Calculate the 20-day moving average  
data_plot['20-Day MA'] = data_plot['Annualized  
Returns'].  
rolling(window=20).mean()
```

Now we can see in [Figure A.2](#) clearly the almost zero interest rate due to Quantitative Easing since 2008's Great Financial Crisis, until it gradually rose starting in 2017 as the economy recovered, and then another plunge to zero during the 2020 COVID era, before it steadily rose again in 2022 in response to the inflationary pressure due to the enormous fiscal stimulus, settling at above 5% at the end of 2023.

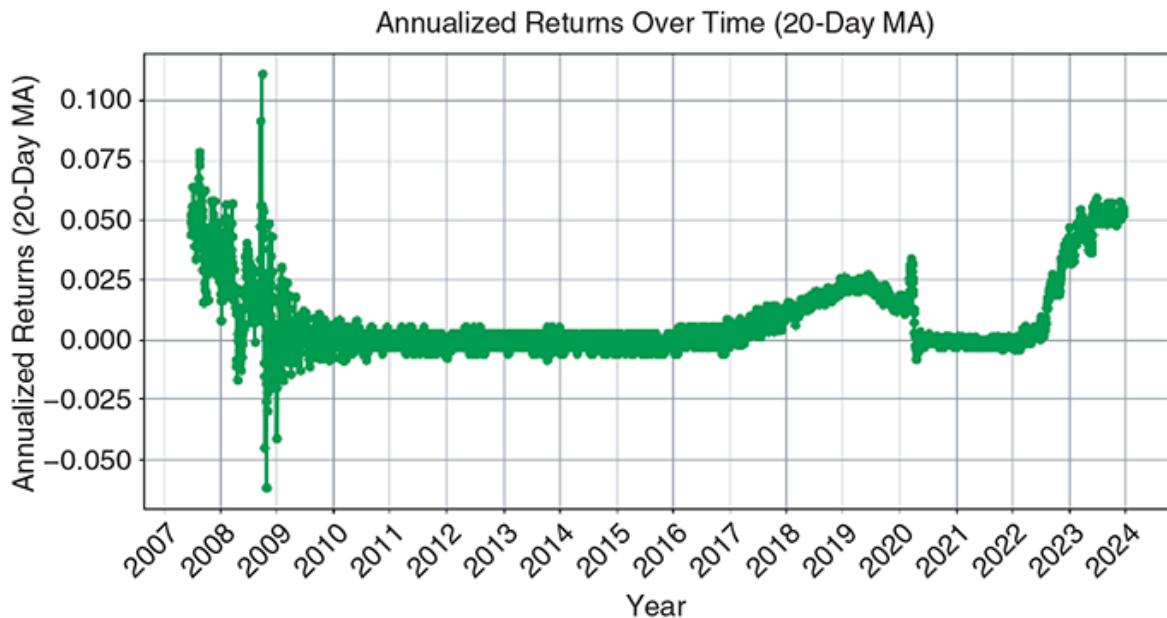


Figure A.2 Twenty-day moving average of annualized BIL returns.

So, after a lot of work, we are satisfied that the BIL data is retrieved correctly and daily returns computed correctly. We will also use this 20-day moving average returns as the R_f for other applications. So

User Prompt

How can I download the dates and moving average numbers to a csv file with 2 columns?

ChatGPT Response

Select only the 'Date' and '20-Day MA' columns for export

```
export_data = data_plot[['Date', '21-Day MA']]
```

Save to a CSV file

```
export_file_path =
'BIL_annualized_returns_20_day_MA.csv'
export_data.to_csv(export_file_path, index=False)
```

A.4 Computing the Sharpe Ratio of SPY

We computed the daily returns of SPY in [Appendix A.1](#), and the risk-free rate (Rf) in [Appendix A.3](#). Now let's use them to compute the Sharpe ratio of SPY.

User Prompt

Suppose a daily returns series A is stored in A.csv, and the daily risk free rate is stored in B.csv, and the dates of A and B are not identical though they overlap, how do we compute the Sharpe ratio of A? Please display the Python code.

ChatGPT Response

To compute the Sharpe Ratio for a series of daily returns (Series A) with a corresponding series of daily risk-free rates (Series B), you'll need to align the two series on their common dates. The Sharpe Ratio is typically calculated as the average return of the investment minus the risk-free rate, divided by the standard deviation of the investment's returns. Here's a step-by-step guide to perform this calculation in Python:

Load both datasets. Align the datasets on their dates. Compute the excess returns by subtracting the risk-free rate from Series A's returns. Calculate the Sharpe Ratio using the formula: (Average Excess Returns) / (Standard Deviation of Excess Returns).

The Python code to achieve this would look something like this:

```
import pandas as pd

# Load Series A (investment returns) and Series B
# (risk-free rate)
series_a = pd.read_csv('path_to_your_csv/A.csv')
series_b = pd.read_csv('path_to_your_csv/B.csv')

# Convert the 'Date' columns to datetime and ensure
# they are the index
series_a['Date'] = pd.to_datetime(series_a['Date'])
series_b['Date'] = pd.to_datetime(series_b['Date'])
series_a.set_index('Date', inplace=True)
series_b.set_index('Date', inplace=True)

# Align the two series on their dates
aligned_a, aligned_b = series_a.align(series_b,
join='inner')

# Calculate excess returns
"""
[Author Note: remembering that the RiskFreeRate
is annualized, we manually insert /252 in the
line below]

[Author Note: beware that the header for the
returns columns may not be called Returns or
RiskFreeRate in your files. Adapt accordingly.]
"""
excess_returns = aligned_a['Returns'] -
aligned_b['RiskFreeRate']/252

# Compute Sharpe Ratio
sharpe_ratio = excess_returns.mean() /
excess_returns.std()

# Adjust for annualization if necessary
# If returns are daily, multiply Sharpe Ratio by
sqrt(number of trading days)
from math import sqrt
trading_days = 252
annualized_sharpe_ratio = sharpe_ratio *
```

```
sqrt(trading_days)

print("Sharpe Ratio:", sharpe_ratio)
print("Annualized Sharpe Ratio:",
annualized_sharpe_ratio)
```

It turns out the code isn't completely correct: `aligned_a` and `aligned_b` contain only dates, not the returns.

User Prompt

Something wrong with the code above: `aligned_a` and `aligned_b` do not contain the returns, only the dates.

ChatGPT replaces the code with

ChatGPT Response

```
...
# Merge the datasets on the 'Date' column
merged_data = pd.merge(series_a, series_b,
on='Date', how='inner')

# Assuming your returns column in series_a is named
'Returns' and
# the risk-free rate column in series_b is named
'RiskFreeRate'

# Calculate excess returns
merged_data['ExcessReturns'] =
merged_data['Returns'] - merged_data['RiskFreeRate']

# Compute Sharpe Ratio
sharpe_ratio = merged_data['ExcessReturns'].mean() /
merged_data['ExcessReturns'].std()
...
```

which finally produced

```
Sharpe Ratio: 0.06826677308094933  
Annualized Sharpe Ratio: 1.0837014262864237
```

A.5 Matlab Code for Computing Efficient Frontier and Finding the Tangency Portfolio

This is the code from Machine Trading (Chan, 2017). For simplicity, we have assumed the risk-free rate (Rf) to be 0. It is also available for download from our GitHub repos.

```

% ef.m
% Find efficient frontier of a set of ETFs
% Note in this program mean and sd of returns are row
vectors,

clear;

load('inputDataOHLCDailyETF_20150417', 'stocks',
'tday', 'cl');

% Remove EWZ and FXI
stocks=stocks(~strcmp('EWZ', stocks) & ~strcmp('FXI',
stocks));
cl=cl(:, ~strcmp('EWZ', stocks) & ~strcmp('FXI',
stocks));

R=calculateReturns(cl, 1); % 1-period "net" returns
R(1, :)=[]; % Skip first row since returns there are
NaN

mi=mean(R, 1); % average return of each stock i.
C=cov(R); % covariance of returns

m=[min(mi):(max(mi)-min(mi))/20:max(mi)]; % prepare
different target mean portfolio returns for efficient
frontier
v=NaN(size(m));

% Variance of portfolio to be minimized
H=2*C;

% short sale constraint
A=-eye(length(mi));
b=zeros(length(mi), 1);

% No linear term in minimization
f=zeros(1, length(mi));

% Fixing portfolio mean return and the normalization
constraint
Aeq=[mi; ones(1, length(mi))];

```

```

for i=1:length(m)
    beq=[m(i); 1];
    [F, v(i)]=quadprog(H, f, A, b, Aeq, beq);
end

sd=sqrt(v);
scatter(sd, m);
hold on;

% Find tangency portfolio
sharpeRatio=m./sd;
[~, idx]=max(sharpeRatio);
scatter(sd(idx), m(idx), 'red');

beq=[m(idx); 1];
[F]=quadprog(H, f, A, b, Aeq, beq)

% F =
%
%      0.451338068065785
%      0.263444604411169
%      0.000019328104256
%      0.000004913963978
%      0.000007497342085
%      0.285185588112728

% Find minimum variance portfolio
[~, idxMin]=min(sd);
scatter(sd(idxMin), m(idxMin), 'green', 'filled');

beq=[m(idxMin); 1];
[F]=quadprog(H, f, A, b, Aeq, beq)

% F =
%
%      0.381518036613685
%      0.000033166607068
%      0.604011066007258
%      0.000133505575004

```

```
%      0.014299849485899
%      0.000004375711085
```

It produces the nice plot in [Figure A.3](#), with the green dot indicating the minimum variance portfolio and the red circle indicating the tangency portfolio (with maximum Sharpe ratio).

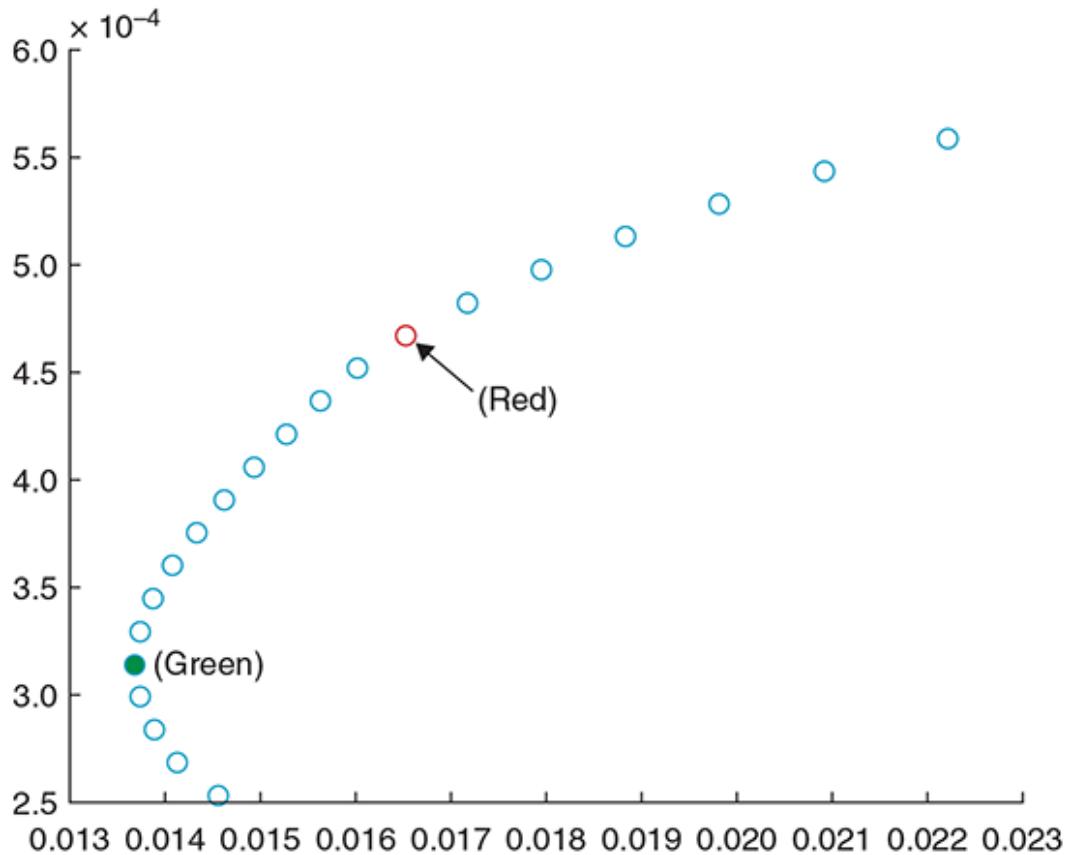


Figure A.3 Efficient Frontier produced by Matlab codes.

Index

A

- accuracy of prediction, [85](#)–86
- ADF (augmented Dickey-Fuller test), [103](#)
- alternative data input, [47](#)
- amortized inference, [174](#)
- Ancestral sampling, [130](#)
- Anscombe quartet, [85](#)

AR (autoregressive) models, [125](#)

CMNN (Causal Masked Neural Network), [128](#), [131](#)-133

WaveNet, [132](#)-133

logistic regression, [128](#)

logistic regression model, [128](#)-129

MADE (Masked Autoencoder for Density Estimation), [130](#)-[131](#)

NN (neural networks), [128](#)

CMNN (Causal Masked Neural Network), [128](#), [131](#)-133

RNN (Recurrent Neural Network), [128](#)

representation complexity, [126](#)

reduction, [127](#)-128

RNN (Recurrent Neural Network), [133](#)-134

transformer models, [134](#)-135

Add blocks, [148](#)

attention, [135](#)-144

multi-headed, [146](#)-147

Encoder layer, [148](#)-149

feed-forward layer, [147](#)-148

model objective, [149](#)-150

Norm blocks, [148](#)

positional encodings, [145](#)-146

ASR (automatic speech recognition), [222](#)

ATM (At-The-Money) option

Matlab Financial Toolbox, [60](#)–61

OTM options, [63](#)

term structure, [58](#)

attention, transformer models

BERT (Bidirectional Encoder Representations from Transformers) matrix, [136](#)–139

BertViz, [137](#)–140

context dependence, [135](#)

hard attention, [140](#)

Hugging Face library, [135](#)–137

multi-headed, [146](#)–147

NLP (Natural Language Processing) and, [135](#)

scaled dot-product, [141](#)–143

soft attention, [140](#)

token embeddings, [135](#)

Token IDs, [135](#)–137

tokenization, [135](#)

unnormalized scores, [142](#)

visualizing, [137](#)–140

augmented Dickey-Fuller test (ADF), [103](#)

automatic speech recognition (ASR), [222](#)

Autoregressive Flows, MAF (Masked Autoregressive Flow), [195](#)

autoregressive models, [155](#). *See also* [Maximum Likelihood models](#)

B

backpropagation, [93](#)–95
backtesting code, [29](#), [44](#)–45
 creating from trading strategies, [34](#)–45
backwardation, [36](#)–37
 VIX term structure, [37](#)
BERT (Bidirectional Encoder Representations from Transformers), [134](#)–135
 bidirectional context, [226](#)
 FinBERT, [229](#)–231
 input representation, [226](#)–227
 language modeling, [149](#)–150
 output representation, [227](#)–228
 sentiment analysis, [225](#)–231
beta neutral (BN) trading strategies, [34](#)–35
betas, constant_maturity_betas, [35](#)
bidirectional context, BERT, [226](#)
BIL (SPDR Bloomberg 1-3 Month T-Bill ETF), [7](#)
binary MNIST, [128](#)–129
Black-Scholes options pricing model, [50](#)–66
 implied volatility, [63](#)
 stochastic volatility, [63](#)
BN (beta neutral) trading strategies, [34](#)–35
boosted trees, [89](#)–91

butterfly spreads in time, term structure arbitrage strategies, [51](#)

C

CAI (Corrective AI), [45](#), [72](#), [78](#)

calendar spreads, term structure arbitrage strategies

backtesting and, [102](#)

butterfly spreads, [51](#)

diagonal spreads, [51](#)

dynamic hedging strategies, [51](#)

horizontal spreads, [50](#)

implementation, [51](#)–[52](#)

interest rate options arbitrage, [51](#)

Long-term Option (buy), [52](#)

OHLC trade bars, [102](#)

Short-Term Option (sell), [52](#)

variance swaps, [51](#)

volatility surface arbitrage, [51](#)

Calmar ratio, [84](#)

CAPM (Capital Asset Pricing Model), [19](#)

CART, tree-based models, [89](#)–[90](#)

boosted trees, [89](#)

random forest, [89](#)

Causal Masked Neural Network (CMNN), [128](#), [131](#)–[133](#)

WaveNet, [132](#)–[133](#)

ChatGPT

- as autoregressive model, [114](#)-118
- conceptual intelligence, [29](#)
- need for human interaction, [64](#)-65
- prompt examples, [4](#)
- response examples, [4](#)
- sampling strategies, [118](#)
- statistical information extraction, [10](#)
- technical details, issues with, [45](#)

ChatGPT GPT4o, [3](#)

- response example, [115](#)
- closing prices, adjusting, [263](#)-265
- cMDA (cluster-based Feature Selection), [71](#), [76](#)-77, [85](#)
- CMNN (Causal Masked Neural Network), [128](#), [131](#)-133
 - WaveNet, [132](#)-133
- codes, from strategy specification, [19](#)-33
- conditional generative models, [111](#)-112
- conditional probability, [115](#)-117
- confusion matrix, [87](#)
- constant_maturity_betas, [35](#)
- contango, [36](#)
 - VIX term structure, [37](#)
- Continuous Normalizing Flows, [195](#)-196
- Corrective AI (CAI), [45](#), [72](#), [78](#)
- correlation coefficient ρ , [60](#)

coupling flows, [188](#)

NICE (Nonlinear Independent Components Estimation), [189](#), [190–191](#)

Real-NVP (Non-volume Preserving Transformation), [189](#), [191–195](#)

CPO (Conditional Parameter/Portfolio Optimization), [45](#), [72](#), [78–79](#)

cross-sectional features, [26](#)

NICE (Nonlinear Independent Components Estimation), [196](#)

Real-NVP (Non-volume Preserving Transformation), [196](#)
time-series, [104](#), [178](#)

csv files, output, [6](#)

cumulative returns curve, [33](#)

D

daily returns

adjusted closing prices and, [263–265](#)

backtesting and, [4](#)

computing requirements, [4](#)

factor data and, [68–69](#)

MSFT daily returns, [96](#)

SPDR Bloomberg 1-3 Month T-Bill ETF, [7](#)

SPY daily returns, [96](#)

DALL-E, [260](#)

data, non-numeric, [9–10](#)

data engineering

 dividends, [101](#)

 look-ahead bias, [101](#)-102

 split adjustments, [101](#)

 survivorship bias, [101](#)

 unique identifiers, [100](#)-101

data formatting, [8](#)-10

decision boundary, [87](#)

dendrogram for stock tickers, [74](#)

denoising, [75](#)

denormalization, time series transformers, [151](#)

dequantization, [151](#)

derived distribution, [184](#)

design matrix, [79](#)

diagonal spreads, term structure arbitrage strategies, [51](#)

Diffusion Models, [260](#)

discriminative models, [110](#)-111

 generative models hybrid, [119](#)

dividends, data engineering and, [101](#)

DLGM (deep latent Gaussian model), [170](#)-171

DLMs (dynamic linear models), [178](#)

DLVM (deep latent variable models), [161](#), [170](#)-171

DNN (Deep Neural Networks), [120](#)-121, [183](#)-184

dollar-neutral portfolio, Sharpe ratio, [31](#)

DPO (Direct Preference Optimization), [256](#)

DRL (Deep Reinforcement Learning)
portfolio optimization and, [46](#)–49
Sharpe ratio, [99](#)–100
dropout, [95](#)
dynamic hedging strategies, term structure arbitrage strategies, [51](#)

E

EDA (exploratory data analysis), [84](#)–85, [104](#)
efficient frontier computing, [272](#)–274
EH (Expectations Hypothesis), [34](#)–35
ELBO (Evidence Lower Bound), [172](#), [173](#)–174
ELMo Open AI GPT, [227](#)
ES futures, VIX futures HR (hedge ratio), [41](#)–43
ETF SPY, daily prices, [5](#)–8
ETFs (exchange-traded funds)
 BIL (SPDR Bloomberg 1-3 Month T-Bill ETF), [7](#)
 MATLAB code, [11](#)–16
 portfolio, [4](#)
 event input data, [47](#)
 excess returns, Sharpe ratio, [31](#)
Explicit Density Models
 autoregressive models, [123](#)
 normalizing flows, [123](#)
VAEs (Variational Autoencoders), [123](#)

F

F1 score, [86](#)

FA (Factor Analysis), [75](#), [161](#), [162](#)-163

Fama-French data file, [21](#)

Fama-French factors, [19](#)-33, [68](#)-69, [161](#)

Fama, Eugene F., [19](#)

feature engineering, [102](#)

ADF (augmented Dickey-Fuller test), [103](#)

Generative AI as generator, [105](#)

importance ranking, [105](#)-106

stationarity, [103](#)

third-party feature validation, [104](#)-105

time series

merging, frequencies and, [103](#)-104

versus cross-sectional features, [104](#)

feedforward neural networks, [92](#)

FFmpeg, [224](#)-225

FinBERT, [229](#)-231

Distilled FinBERT, [249](#)-252

speed, [240](#)-247

flow models, [183](#)

Autoregressive Flows, MAF (Masked Autoregressive Flow), [195](#)

conditioned normalizing flows, [197](#)–198

Continuous Normalizing Flows, [195](#)–196

coupling flows, [188](#)

NICE, [189](#), [190](#)–191

Real-NVP, [189](#), [191](#)–195

derived distribution, [184](#)

linear flows, [185](#)–187

model training, [185](#)

nonlinear flows, [187](#)–188

normalizing flow, [184](#)–185

taxonomy, [184](#)

time series

domain knowledge, [198](#)

flows, adapting, [196](#)–197

image data transition, [196](#)

FN (false negatives), [86](#)

FP (false positives), [86](#)

FPR (False Positive Rate), [87](#)

French, Kenneth R., [19](#)

FT (fine-tuning), [253](#)–254

fundamental data, [46](#)–47

futures, look-ahead bias, [102](#)

FVSBN (fully visible sigmoid belief network), [129](#)–130

G

GANs (Generative Adversarial Networks), [123](#), [159](#), [201](#)
SNGAN (Spectral Normalization GAN), [211](#), [213](#)
taxonomy, [202](#)
time series, [214](#)-215
training, [202](#)-208
WGAN (Wasserstein GAN), [211](#)-212
 WGAN-GP, [213](#)-214
Gaussian Brownian Motion, [50](#), [55](#)
Gemini Pro (Google), [3](#)

Generative AI, [109](#)

- discriminative models, [110](#)-111
- generative models, [110](#)-111
 - as simulators, [113](#)-118
 - ChatGPT and, [113](#)-119
 - conditional, [111](#)-112
 - data creation, [114](#)
 - data generation, [112](#)
 - discriminative models, [110](#)-111
 - Explicit Density Models, [123](#)
 - Implicit Density Models, [123](#)
 - NLP (Natural Language Processing), [113](#)
 - probability density estimation, [112](#)
 - RAG (Retrieval-Augmented Generation), [119](#)
 - representation learning, [112](#)-113, [118](#)-119
 - taxonomy, [109](#)
- hybrid modeling, [119](#)-123

generative models

AR (autoregressive) models, [125](#)-126

as simulators, [113](#)-118

ChatGPT and

 data creation, [114](#)

 language generation, conditional, [114](#)-118

 language modeling, [113](#)-114

 conditional, [111](#)-112

 data creation, [114](#)

 data generation, [112](#)

 discriminative models, [110](#)-111, [119](#)

Explicit Density Models

 autoregressive models, [123](#)

 normalizing flows, [123](#)

 VAEs (Variational Autoencoders), [123](#)

Implicit Density Models

 Diffusion Models, [123](#)

 GANs (Generative Adversarial Networks), [123](#)

NLP (Natural Language Processing), [113](#)

probability density estimation, [112](#)

RAG (Retrieval-Augmented Generation), [119](#)

representation learning, [112](#)-113, [118](#)-119

taxonomy, [109](#)

variants, combining, [260](#)-261

GLM (Generalized Linear Model), [123](#)

GMM (Gaussian Mixture Models), [77](#), [110](#)–[111](#), [162](#), [166](#)
as HMM (Hidden Markov Model), [170](#)
limitations, [170](#)
market regime detection, [167](#)–[169](#)
GNP (gross national product), [128](#)
GPT (Generative Pre-trained Transformers), [134](#)–[135](#)
Grok (xAI), [3](#)

H

hedging market risk, mini-S&P futures and, [39](#)–[40](#)
Heston volatility model, [55](#)–[57](#)
historical market data retrieval, [4](#)–[16](#)
HML (High-Minus-Low), [20](#), [161](#)
HMM (Hidden Markov Model), [71](#), [72](#), [77](#)
 EM (Expectation-Maximization), [170](#)
 GMM (Gaussian Mixture Model as HMM), [170](#)
 Markov Property, [127](#)
 RNNs (Recurrent Neural Networks) and, [97](#)
HPO (hyperparameter optimization), [83](#)–[84](#)
HR (hedge ratio), VIX futures, [41](#)–[43](#)
HRP (Hierarchical Risk Parity), [71](#), [72](#), [73](#)–[74](#)
hyperparameter optimization, [71](#)

I

identifiers, unique identifiers, [100–101](#)
IID (independent and identically distributed) observations, [84](#)
Implicit Density Models, [109](#)
Autoregressive models, [126](#)
Flow models, [184](#)
Diffusion Models, [123](#)
GANs (Generative Adversarial Networks), [202](#), [123](#)
Variational autoencoders, [160](#)
implied volatility, [63–64](#)
inhomogeneous features, [89](#)
input data, [46–49](#)
interest rate options arbitrage, term structure arbitrage strategies, [51](#)
IQR (interquartile range), [152](#)

J

Jensen's inequality, [172](#)

K

Kalman filter, [125](#), [161](#)
Kullback-Leibler divergence, [156](#), [172](#), [202](#), [243](#)

L

L1 regularization, [81](#)-83
L2 regularization, [81](#)-83
language modeling, transformer models, [149](#)-150
Levy Process Model, [55](#)
LFSS (Latent Feature State Space) module, [47](#)-49
LightGBM, NaNs, [22](#)-23
linear flows, [185](#)-187
 linear normalizing flows, [185](#)-186
 nonlinear flows, [187](#)-188
linear regression
 binary variable, [80](#)
 L1 regularization, [81](#)-83
 L2 regularization, [81](#)-83
 LinearRegression (Scikit-learn), [23](#)-24
 MLE (maximum likelihood estimation), [80](#)
 NaNs (Not a Number), [22](#)
 Python, [23](#)
 Statsmodels, [23](#)
 supervised learning, [79](#)-81
 training, [22](#)-26
LinearRegression (Scikit-learn), [23](#)-24
LL (log likelihood), [85](#)

LLMs (Large Language Models), [119](#)
as financial advisors, [261](#)
customization, [252](#)-256
emergent abilities, [237](#)
model size impact, [237](#)-238
scaling, [236](#)-239
sentiment analysis
BERT model, [225](#)-231
Fed press conference speech, [217](#)-221
results of experiment, [232](#)-234
speech-to-text conversion, [221](#)-225
video and audio data, [221](#)
training time effects, [238](#)-239

LM (Language Modeling), [113](#), [150](#)-152
efficient inference, [239](#)-240

logistic regression model, [128](#)-129

logistic regressions, [71](#), [128](#)
supervised learning, [79](#)-81

long_maturity_tenor, [35](#)

long-term options, calendar spread strategy, [52](#)-53

look-ahead bias, [101](#)-102

LoRA (low-rank adaptation of Large Language Models),
[254](#)-256

LVMs (latent variable models), [159](#)
amortized inference, [174](#)
DLVM (deep latent variable models), [170](#)-171
ELBO (Evidence Lower Bound), [172](#), [173](#)-174
FA (Factor Analysis), [161](#), [162](#)-163
 PPCA comparison, [164](#)-165
Fama-French factors, [161](#)
GANs (Generative Adversarial Networks), [159](#)
GMM (Gaussian Mixture Models), [162](#), [166](#)-170
heteroscedastic noise, [164](#)-166
HML (High-Minus-Low), [161](#)
Jensen's inequality, [172](#)
latent space recovery, [164](#)-165
optimization, [174](#)-175
PPCA (Probabilistic Principal Component Analysis), [162](#),
[163](#)-166
 FA comparison, [164](#)-165
 proposal distribution, [173](#)-174
 SMB (Small Minus Big), [161](#)
VAEs (Variational Autoencoders), [176](#)-177
 sequential data, [177](#)-181
 time series, [177](#)
 extending, [178](#)
 superposition, [178](#)-179
 TimeVAE, [179](#)-181
variational inference approximation, [171](#)-172

M

- macro signals data, [47](#)
- MADE (Masked Autoencoder for Density Estimation), [130](#)-[131](#)
- MAF (Masked Autoregressive Flow), [195](#)
- market data retrieval, historical data, [4](#)-[5](#)
- market indicators inputs, [47](#)
- market risk (Rm-Rf), [19](#)
- Markov Property, [127](#)
- Matlab, [3](#)-[4](#), [32](#)
 - ATM (At-The-Money) option, [60](#)-[61](#)
 - codes, translating to Python, [11](#)-[16](#)
 - efficient frontier computing, [272](#)-[274](#)
 - Matlab Financial Instruments Toolbox, [61](#)
 - tangency portfolio, [272](#)-[274](#)
- Maximum Likelihood models, [155](#)
- mean reversion
 - estimates, SPX, [58](#)-[60](#)
 - Heston model, [56](#)-[57](#)
- metalabeling, [78](#)
- Microsoft Copilot, [3](#)
- mini-S&P futures, [35](#)
 - hedging market risk and, [39](#)-[40](#)
 - VIX futures HR (hedge ratios), [41](#)-[43](#)

ML (machine learning)

- bosting, [90](#)
- CAI (Corrective AI), [78](#)
- data engineering
 - dividends, [101](#)
 - look-ahead bias, [101](#)–102
 - split adjustments, [101](#)
 - survivorship bias, [101](#)
 - unique identifiers, [100](#)–101
- decision boundary, [87](#)
- design matrix, [79](#)
- feature engineering, [102](#)
 - Generative AI as generator, [105](#)
 - importance ranking, [105](#)–106
 - stationarity, [103](#)
 - third-party feature validation, [104](#)–105
 - time series, [103](#)–104
- neural networks, [89](#)

NN (neural networks), [91](#)–96
 AR (autoregressive models), [128](#)
 backpropagation, [93](#)–95
 CMNN (Causal Masked Neural Network), [128](#), [131](#)–
 133
 DNN (Deep Neural Networks), [120](#)–121
 dropout, [95](#)
 feedforward neural networks, [92](#)
 MLP (multilayer perceptron), [92](#)–93
 PyTorch, [93](#)
 regularization, [95](#)
 RNN (Recurrent Neural Network), [96](#)–98, [128](#), [133](#)–
 134
 TensorFlow, [93](#)
portfolio optimization, [45](#)–49
regularizing, [76](#)

supervised learning, [77](#)–78
 CAI (Corrective AI), [72](#)
 CPO (Conditional Parameter/Portfolio Optimization),
 [72](#)
 cross-validation, [83](#)–84
 HPO (hyperparameter optimization), [71](#), [83](#)–84
 linear regressions, [79](#)–81
 logistic regressions, [71](#), [79](#)–81
 performance metrics, [71](#), [84](#)–89
 tree-based models, [71](#)
 validation, [83](#)–84
tree-based models, [89](#)–91
unsupervised learning, [71](#)–77, [85](#)
 cMDA (cluster-based Feature Selection), [71](#), [76](#)–77
 HHM (Hidden Markov Model), [71](#), [72](#), [77](#)
 HRP (Hierarchical Risk Parity), [71](#), [72](#), [73](#)–74
 PCA (Principal Component Analysis), [71](#), [72](#), [75](#)–76
MLE (maximum likelihood estimation), [80](#), [155](#), [165](#), [201](#)
MLP (multilayer perceptron), [92](#)–93
MNIST dataset, [128](#)–131, [191](#)
MSE (mean squared errors), [80](#), [84](#)

N

- NaNs (Not a Number), [10](#)
- LightGBM, [22](#)-23
- regression models, [22](#)
- Negative Log Likelihood, [156](#)
- Neural ODE (Neural Ordinary Differential Equation), [195](#)-196
- news input data, [47](#)
- next-N-day returns, [67](#)-69
- NICE (Nonlinear Independent Components Estimation), [189](#), [190](#)-191
- NLP (Natural Language Processing)
 - attention, transformer models, [135](#)
 - generative models, [113](#)
 - language generation, [114](#)-118
 - LM (Language Modeling), [113](#)

NN (neural networks), [91](#)–96
 AR (autoregressive models), [128](#)
 backpropagation, [93](#)–95
 CMNN (Causal Masked Neural Network), [128](#), [131](#)–133
 DNN (Deep Neural Networks), [120](#)–121
 dropout, [95](#)
 feedforward neural networks, [92](#)
 HMMs (Hidden Markov Model), [97](#)–98
 MLP (multilayer perceptron), [92](#)–93
 PyTorch, [93](#)
 regularization, [95](#)
 RNN (Recurrent Neural Network), [96](#)–98, [128](#), [133](#)–134
 TensorFlow, [93](#)
non-numeric data, [9](#)–10
normalizing flow, [184](#)–185
NumPy, [11](#)

O

OHLC (Open, High, Low, Close) data, [48](#)
OOD (out-of-distribution) problem, [121](#)–122
OpenAI API, [115](#)

OpenAI

- ChatGPT access, [118–119](#)
 - data collection, [222](#)
 - DALL-E, [260](#)
 - ELMo OpenAI GPT, [227](#)
 - Embeddings API, [119](#)
 - .en models, [223](#)
 - GPT-3 transformation, [256](#)
 - GPT family, [134–135](#)
 - RAG (Retrieval-augmented Generation), [253](#)
 - tokens, [113](#)
- Open AI BERT. *See* [BERT \(Bidirectional Encoder Representations from Transformers\)](#).
- OpenAI Whisper, [218](#), [222–225](#)
 - optimization, LVMs (latent variable models), [174–175](#)
 - order book input data, [47](#)
 - orthogonal features, [75](#)
 - OTM, [53](#), [54](#), [58](#), [62](#), [63](#)
 - output, csv files, [6](#)

P

parameters

CPO (Conditional Parameter/Portfolio Optimization), [45](#), [72](#), [78](#)–79

hyperparameter optimization, [71](#)

HPO (hyperparameter optimization), [83](#)–84

MLE (maximum likelihood estimation), [155](#)

PEFT (parameter-efficient fine-tuning), [253](#)–256

PCA (Principal Component Analysis), [71](#), [72](#), [75](#)–76, [163](#)–166

PEFT (parameter-efficient fine-tuning), [253](#)–256

penalty terms, [82](#), [214](#)

performance metrics, [71](#)

accuracy of prediction, [85](#)–86

AUC, [89](#)

AUCH, [89](#)

F1 score, [89](#)

FN (false negatives), [86](#)

FP (false positives), [86](#)

macro-average, [86](#)

micro-average, [86](#)

TP (true positives), [86](#)

weighted-average, [86](#)

PoP (probability of profit), [78](#)

portfolio optimization, [45](#)–49

PPCA (Probabilistic Principal Component Analysis), [75](#), [162](#)
FA (Factor Analysis), [164](#)-166
PCA (Principal Component Analysis), [75](#)
PPO (Proximal Policy Optimization), [100](#)
predict_returns functions, [26](#)
price-based input data, [46](#)
probability density estimation, [112](#), [115](#)-117
prompt examples, [4](#)
proposal distribution, [173](#)-174
Pyraformer model, [155](#)
Python
 backtesting, [44](#)-45
 FFmpeg, [224](#)
 installation, [265](#)-267
 linear regression, [23](#)
 Matlab codes, [11](#)-16
 next-day returns, [67](#)
 Pandas dataframe, [68](#)
 Sharpe Ratio, [270](#)-271
 Whisper library, [223](#)
PyTorch, [93](#), [95](#), [98](#)
 Attention Head, [143](#)
 linear quantization, [251](#)-252
 PyTorch Lightning, [243](#)
 pytube library, [221](#), [232](#)

Q

- Q-learning, [100](#)
- quantization, linear, [247](#)-249
- Distilled FinBERT, [249](#)-252
- PyTorch, [251](#)-252

R

- RAG (Retrieval-Augmented Generation), [119](#), [252](#)-253
- Random Forest, tree-based models, [90](#)
- Real-NVP (Non-volume Preserving Transformation), [189](#),
[191](#)-195

regression

CPO (Conditional Parameter Portfolio Optimization), [78](#)-[79](#)

linear regression

binary variable, [80](#)

L1 regularization, [81](#)-[83](#)

L2 regularization, [81](#)-[83](#)

LinearRegression (Scikit-learn), [23](#)-[24](#)

MLE (maximum likelihood estimation), [80](#)

NaNs (Not a Number), [22](#)

Statsmodels, [23](#)

supervised learning, [79](#)-[81](#)

training, [22](#)-[26](#)

logistic regressions, [71](#)

supervised learning, [79](#)-[81](#)

performance metrics, [84](#)-[89](#)

regression trees, [89](#)-[91](#)

CART, [89](#)-[90](#)

Random Forest, [90](#)

regularization, [90](#)

regression models

fitted model objects, [25](#)

LinearRegression (Scikit-learn), [23](#)-[24](#)

NaNs (Not a Number), [22](#)

Statsmodels, [23](#)

training, [22](#)-[26](#)

regression trees, [89](#)–91
 CART, [89](#)–90
 Random Forest, [90](#)
regularization, [90](#)
regularization
 L1 regularization, [81](#)–83
 L2 regularization, [81](#)–83
 NN (neural networks), [95](#)
 penalty terms, [82](#)
 regression trees, [90](#)
 transformer models, [148](#)–149
relative value spreads, [34](#)
representation complexity, AR (autoregressive) models,
[126](#)–128
representation learning, [112](#)–113, [118](#)–119
responses to prompts, examples, [4](#)
results, plotting, [7](#)
returns, cumulative returns curve, [33](#)
Rf (risk-free) rate, [32](#), [268](#)
risk metrics, inputs, [47](#)
risk-free-rate plotting over years, [268](#)–270
RL (reinforcement learning), [78](#)
 DRL (Deep Reinforcement Learning), [46](#)–48, [99](#)–100
 RLHF (Reinforcement Learning from Human Feedback),
[256](#)
Rm-Rf (market risk), [19](#)

RNN (Recurrent Neural Network), [96](#)–[98](#), [128](#), [133](#)–[134](#)
HMMs (Hidden Markov Models) and, [97](#)
transformer models, [134](#)
VRAE (Variational Recurrent Autoencoder), [178](#)
ROC (Receiver Operating Characteristics) curve, [87](#)–[89](#)

S

S&P futures, hedging market risk and, [39](#)–[40](#)
sampling, [196](#)

Ancestral sampling, [130](#)
autoregressive generative models, [117](#)
down-sampling, [103](#)
frequencies, [102](#)–[103](#)
from FVSN, [130](#)
GANs, [203](#)
importance sampling, [172](#)
temperature scaling, [118](#)
top-k/top-p sampling, [118](#)
VAE and, [176](#)–[177](#)

Scikit-learn, LinearRegression, [23](#)–[24](#)

SciPy, [11](#)

library installation, [14](#)

SDEs (stochastic differential equations), [56](#)–[57](#)

securities master, [100](#)

sentiment analysis, [113](#)

Financial PhraseBank, [230](#)

FinBERT, [228](#), [229](#), [232](#), [240](#)

LLMs (Large Language Models)

BERT (Bidirectional Encoder Representations from Transformers) model, [225](#)-231

Fed press conference speech, [217](#)-221

results of experiment, [232](#)-234

speech-to-text conversion, [221](#)-225

video and audio data, [221](#)

OpenAI Whisper, [218](#), [222](#)-225

transformer models, [149](#)-150

sequence modeling, AR (autoregressive) models, [125](#)

CMNN (Causal Masked Neural Network), [128](#), [131](#)-133

logistic regression, [128](#)

logistic regression model, [128](#)-129

MADE (Masked Autoencoder for Density Estimation),
[130](#)-131

NN (neural networks), [128](#)

representation complexity, [126](#)-127

representation complexity reduction, [127](#)-128

RNN (Recurrent Neural Network), [133](#)-134

transformer models, [134](#)-150

sequential data, [125](#)

SGD (stochastic gradient descent), [93](#), [156](#)

Sharpe ratio, [3](#), [7](#)–[10](#), [16](#), [84](#)
dollar-neutral portfolio, [31](#)
DRL (Deep Reinforcement Learning), [99](#)–[100](#)
excess returns, [31](#)
Rf (risk-free rate), [268](#)
Risk-Free Rate, [31](#)
SPY, [6](#)–[7](#), [270](#)–[272](#)
strategy and leverage, [31](#)
short_maturity_tenor, [35](#)
short-term options, calendar spread strategy, [52](#)–[53](#)
sigmoid function, [80](#)
size factor, SMB (Small Minus Big), [19](#)
SkyFi, [119](#)
SMB (Small Minus Big), [19](#), [161](#)
SNGAN (Spectral Normalization GAN), [211](#), [213](#)
SPDR Bloomberg 1-3 Month T-Bill ETF. *See* [BIL](#)
split adjustments, data engineering and, [101](#)
SPX (S&P 500 Index)
 ATM (At-The-Money), [63](#)
 correlation coefficient ρ , [60](#)
 mean reversion, [58](#)–[60](#)
 options calendar, [18](#)
SPY, daily returns, [5](#)–[6](#)
 Sharpe ratio, [6](#)–[7](#)
Statsmodels, [23](#)

stochastic volatility models, [55](#)–[57](#), [63](#)–[64](#)

stock returns. *See also* [trading strategies](#)

CAPM (Capital Asset Pricing Model), [19](#)

stock tickers, dendrogram, [74](#)

stock_features function, [26](#)

strategy specification, codes, [19](#)–[33](#)

supervised learning, [77](#)–[78](#)

CAI (Corrective AI), [72](#)

CPO (Conditional Parameter/Portfolio Optimization), [72](#)

cross-validation, [83](#)–[84](#)

HPO (hyperparameter optimization), [83](#)–[84](#)

hyperparameter optimization, [71](#)

linear regression, [71](#), [79](#)–[81](#)

- binary variable, [80](#)
- L1 regularization, [81](#)–[83](#)
- L2 regularization, [81](#)–[83](#)
- MLE (maximum likelihood estimation), [80](#)

logistic regressions, [71](#), [79](#)–[81](#)

performance metrics, [71](#), [84](#)–[89](#)

tree-based models, [71](#)

validation, [83](#)–[84](#)

survivorship bias, [21](#), [101](#)

SVD (singular value decomposition), [75](#)

T

tangency portfolio, [272](#)-274
technical indicators inputs, [47](#)
tenors
 long_maturity_tenor, [35](#)
 short_maturity_tenor, [35](#)
TensorFlow, [93](#), [175](#), [243](#)
term structure arbitrage strategies
 ATM (At-The-Money) option, [60](#)-61
 butterfly spreads in time, [51](#)
 calendar spreads, [50](#)-51, [52](#)-53
 diagonal spreads, [51](#)
 dynamic hedging strategies, [51](#)
 Heston volatility model, [55](#)-57
 implementation, [51](#)-52
 implied volatilities, [54](#)
 publications, [57](#)-58
 stochastic volatility models, [55](#)-57
 volatility smile, [55](#), [56](#)-57
 interest rate options arbitrage, [51](#)
 out-of-the money put, [54](#)
 variance swaps, [51](#)
 volatility surface arbitrage, [51](#)
testing records, [134](#)-135

time series

feature engineering

merging, frequencies and, [103](#)-104

versus cross-sectional features, [104](#)

flow models

domain knowledge, [198](#)

flows, adapting, [196](#)-197

image data transition, [196](#)

GANs (Generative Adversarial Networks), [214](#)-215

transformer models

Attention Layer, [155](#)

Chronos approach, [151](#)-152

contextual information, [154](#)-155

continuous input, [152](#)-155

denormalization, [151](#)

dequantization, [151](#)

discretizing data, [151](#)-152

input normalization, [154](#)-155

Lag-Llama approach, [152](#)-155

positional encoding, [155](#)

tokenization, [151](#)-152

VAEs (Variational Autoencoders), [177](#)

extending, [178](#)

superposition, [178](#)-179

TimeVAE, [179](#)-181

time-series features, [26](#)
TimeVAE, [179](#)–181
TP (true positives), [86](#)
TPR (True Positive Rate), [87](#)
trading strategies, [17](#). *See also* [stock returns](#)
backtest code creation, [34](#)–45
BN (beta neutral), [34](#)–35
Fama-French factors, [19](#)–21
HML (high minus low), [20](#)
relative value spreads, [34](#)
Rm-Rf (market risk), [19](#)
SMB (Small Minus Big), [19](#)
summarizing, [34](#)–45

transformer models, [134](#)-135, [149](#)-150

Add blocks, [148](#)

attention, [135](#)-144

BERT matrix, [136](#)-139

BertViz, [137](#)-140

hard attention, [140](#)

Hugging Face library, [135](#)-137

multi-headed, [146](#)-147

NLP and, [135](#)

scaled dot-product, [141](#)-143

soft attention, [140](#)

token embeddings, [135](#)

Token IDs, [135](#)-137

tokenization, [135](#)

unnormalized scores, [142](#)

visualizing, [137](#)-140

BERT (Bidirectional Encoder Representations from Transformers), [134](#)-135

Encoder layer, [148](#)-149

feed-forward layer, [147](#)-148

GPT (Generative Pre-trained Transformers), [134](#)-135

language modeling, [149](#)-150

model objective, [149](#)-150

NLP (Natural Language Processing), [135](#)

Norm blocks, [148](#)

positional encodings, [145](#)-146

RNNs (Recurrent Neural Networks), [134](#)

sentiment analysis, [149](#)

time series

Attention Layer, [155](#)

Chronos approach, [151](#)-152

contextual information, [154](#)-155

continuous input, [152](#)-155

denormalization, [151](#)

dequantization, [151](#)

discretizing data, [151](#)-152

input normalization, [154](#)-155

Lag-Llama approach, [152](#)-155

positional encoding, [155](#)

tokenization, [151](#)-152

visualization, [144](#)-145

tree-based models, [71](#)

tree-based regression models, [89](#)-91

CART, [89](#)-90

Random Forest, [90](#)

U

unique identifiers, [100](#)-101

unsupervised learning, [71](#)

cMDA (cluster-based Feature Selection), [71](#), [76](#)–77, [85](#)

HHM (Hidden Markov Model), [71](#), [72](#), [77](#)

HRP (Hierarchical Risk Parity), [71](#), [72](#), [73](#)–74

PCA (Principal Component Analysis), [71](#), [72](#), [75](#)–76

V

VAEs (Variational Autoencoders), [123](#), [176](#)–177, [183](#)–184

Explicit Density Models, [123](#)

sequential data, [177](#)–181

time series, [177](#)

extending, [178](#)

superposition, [178](#)–179

TimeVAE, [179](#)–181

validation, cross-validation, [83](#)–84

value factor, HML (high minus low), [20](#)

VaR (Value at Risk), [117](#)

variables. *See also* [LVMs \(latent variable models\)](#).

variance swaps, term structure arbitrage strategies, [51](#)

visualization, transformer models, [144](#)–145

attention, [137](#)–140

VIX futures, [34](#)–39

ES futures HR (hedge ratio), [41](#)–43

mini-S&P futures HR (hedge ratio), [41](#)–43

VIX index, [38](#)

volatility smile, [55](#), [56](#)-57

volatility surface arbitrage, term structure arbitrage strategies, [51](#)

volume data inputs, [46](#)

VRAE (Variational Recurrent Autoencoder), [178](#)

VWAP (Volume-Weighted Average Price), [103](#)

W-Z

WaveNet, [132](#)-133

WGAN (Wasserstein GAN), [211](#)-214

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.