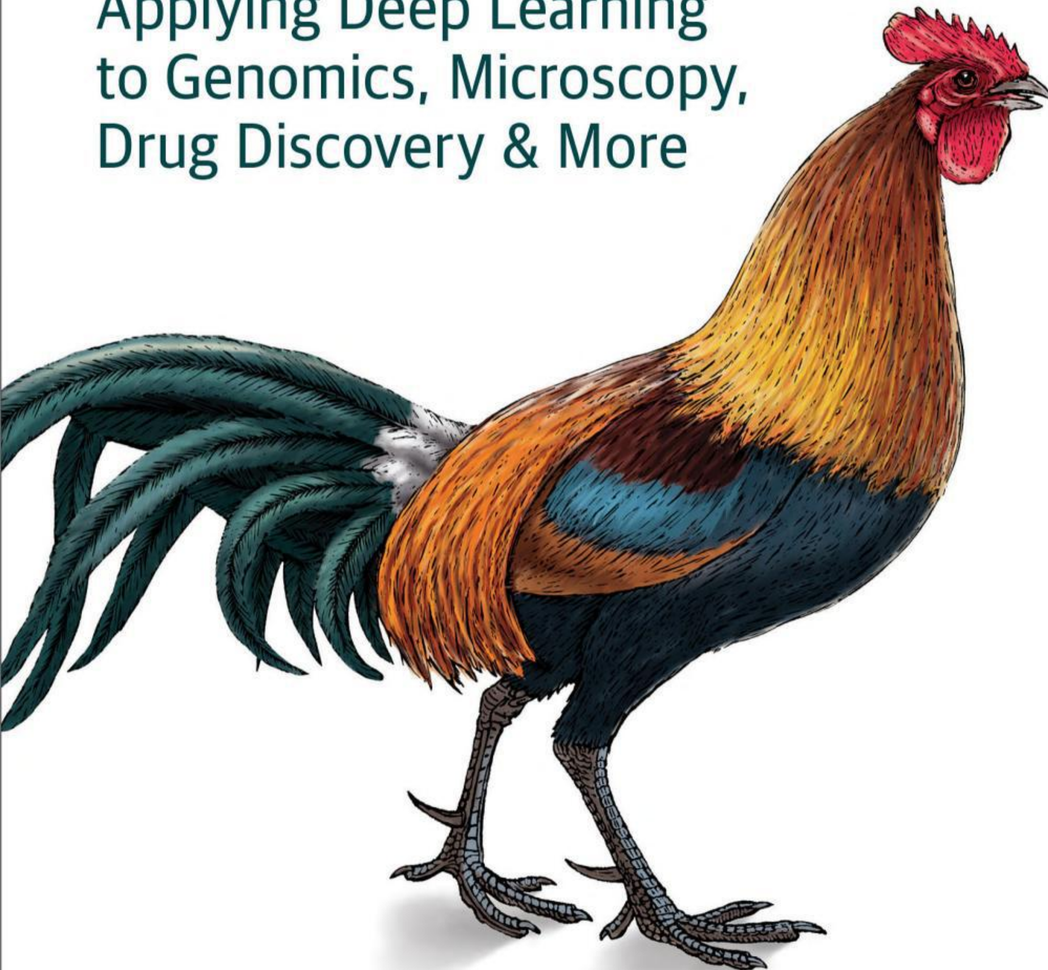
O'REILLY®

# Deep Learning for the Life Sciences

## Applying Deep Learning to Genomics, Microscopy, Drug Discovery & More



Bharath Ramsundar, Peter Eastman,
Patrick Walters & Vijay Pande

Copyrighted material

# Deep Learning for the Life Sciences

*Applying Deep Learning to Genomics, Microscopy, Drug Discovery, and More*

*Bharath Ramsundar, Peter Eastman, Patrick Walters, and Vijay Pande*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Copyrighted material

**Deep Learning for the Life Sciences**

by Bharath Ramsundar, Peter Eastman, Patrick Walters, and Vijay Pande

Copyright © 2019 Bharath Ramsundar, Peter Eastman, Patrick Walters, and Vijay Pande. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Development Editor:** Nicole Tache
**Acquisitions Editor:** Mike Loukides
**Production Editor:** Katherine Tozer
**Copyeditor:** Rachel Head
**Proofreader:** Zachary Corleissen
**Indexer:** Ellen Troutman-Zaig
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery
**Illustrator:** Rebecca Demarest

April 2019: First Edition

**Revision History for the First Edition**

2019-03-27: First Release

See *http://bit.ly/deep-learning-life-science* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Deep Learning for the Life Sciences*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-03983-9

[LSI]

Copyrighted material

# Table of Contents

**Preface** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **vii**

**1. Why Life Science?** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**
Why Deep Learning? 1
Contemporary Life Science Is About Data 2
What Will You Learn? 3

**2. Introduction to Deep Learning** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7**
Linear Models 8
Multilayer Perceptrons 10
Training Models 13
Validation 15
Regularization 15
Hyperparameter Optimization 17
Other Types of Models 18
Convolutional Neural Networks 18
Recurrent Neural Networks 19
Further Reading 21

**3. Machine Learning with DeepChem** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **23**
DeepChem Datasets 24
Training a Model to Predict Toxicity of Molecules 25
Case Study: Training an MNIST Model 32
The MNIST Digit Recognition Dataset 33
A Convolutional Architecture for MNIST 34
Conclusion 39



Copyrighted material

**4. Machine Learning for Molecules. . . . . . . . . . 41**
What Is a Molecule? 42
What Are Molecular Bonds? 44
Molecular Graphs 46
Molecular Conformations 47
Chirality of Molecules 48
Featurizing a Molecule 49
SMILES Strings and RDKit 49
Extended-Connectivity Fingerprints 50
Molecular Descriptors 51
Graph Convolutions 51
Training a Model to Predict Solubility 52
MoleculeNet 54
SMARTS Strings 54
Conclusion 57

**5. Biophysical Machine Learning. . . . . . . . . . 59**
Protein Structures 61
Protein Sequences 63
A Short Primer on Protein Binding 66
Biophysical Featurizations 67
Grid Featurization 68
Atomic Featurization 73
The PDBBind Case Study 73
PDBBind Dataset 73
Featurizing the PDBBind Dataset 77
Conclusion 81

**6. Deep Learning for Genomics. . . . . . . . . . 85**
DNA, RNA, and Proteins 85
And Now for the Real World 87
Transcription Factor Binding 90
A Convolutional Model for TF Binding 90
Chromatin Accessibility 93
RNA Interference 96
Conclusion 99

**7. Machine Learning for Microscopy. . . . . . . . . . 101**
A Brief Introduction to Microscopy 103
Modern Optical Microscopy 104
The Diffraction Limit 107
Electron and Atomic Force Microscopy 108



Copyrighted material

Super-Resolution Microscopy 110
Deep Learning and the Diffraction Limit? 112
Preparing Biological Samples for Microscopy 112
Staining 112
Sample Fixation 113
Sectioning Samples 114
Fluorescence Microscopy 115
Sample Preparation Artifacts 117
Deep Learning Applications 118
Cell Counting 118
Cell Segmentation 121
Computational Assays 126
Conclusion 126

**8. Deep Learning for Medicine. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129**
Computer-Aided Diagnostics 129
Probabilistic Diagnoses with Bayesian Networks 131
Electronic Health Record Data 132
The Dangers of Large Patient EHR Databases? 135
Deep Radiology 136
X-Ray Scans and CT Scans 138
Histology 141
MRI Scans 142
Learning Models as Therapeutics 143
Diabetic Retinopathy 144
Conclusion 147
Ethical Considerations 147
Job Losses 148
Summary 149

**9. Generative Models. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151**
Variational Autoencoders 151
Generative Adversarial Networks 153
Applications of Generative Models in the Life Sciences 154
Generating New Ideas for Lead Compounds 155
Protein Design 155
A Tool for Scientific Discovery 156
The Future of Generative Modeling 156
Working with Generative Models 157
Analyzing the Generative Model's Output 158
Conclusion 161



Copyrighted material

**10. Interpretation of Deep Models. . . . . . . . . . 165**
Explaining Predictions 165
Optimizing Inputs 169
Predicting Uncertainty 172
Interpretability, Explainability, and Real-World Consequences 176
Conclusion 177

**11. A Virtual Screening Workflow Example. . . . . . . . . . 179**
Preparing a Dataset for Predictive Modeling 180
Training a Predictive Model 186
Preparing a Dataset for Model Prediction 191
Applying a Predictive Model 195
Conclusion 202

**12. Prospects and Perspectives. . . . . . . . . . 203**
Medical Diagnosis 203
Personalized Medicine 205
Pharmaceutical Development 206
Biology Research 208
Conclusion 209

**Index. . . . . . . . . . 211**



Copyrighted material

# Preface

In recent years, life science and data science have converged. Advances in robotics and automation have enabled chemists and biologists to generate enormous amounts of data. Scientists today are capable of generating more data in a day than their predecessors 20 years ago could have generated in an entire career. This ability to rapidly generate data has also created a number of new scientific challenges. We are no longer in an era where data can be processed by loading it into a spreadsheet and making a couple of graphs. In order to distill scientific knowledge from these datasets, we must be able to identify and extract nonobvious relationships.

One technique that has emerged over the last few years as a powerful tool for identifying patterns and relationships in data is *deep learning*, a class of algorithms that have revolutionized approaches to problems such as image analysis, language translation, and speech recognition. Deep learning algorithms excel at identifying and exploiting patterns in large datasets. For these reasons, deep learning has broad applications across life science disciplines. This book provides an overview of how deep learning has been applied in a number of areas including genetics, drug discovery, and medical diagnosis. Many of the examples we describe are accompanied by code examples that provide a practical introduction to the methods and give the reader a starting point for future research and exploration.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.



Copyrighted material

**`Constant width bold`**
: Shows commands or other text that should be typed literally by the user.

*`Constant width italic`*
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

## Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/deepchem/DeepLearningLifeSciences*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Deep Learning for the Life Sciences* by Bharath Ramsundar, Peter Eastman, Patrick Walters, and Vijay Pande (O'Reilly). Copyright 2019 Bharath Ramsundar, Karl Leswing, Peter Eastman, and Vijay Pande, 978-1-492-03983-9."



Copyrighted material

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

## O'Reilly Online Learning

O'REILLY®

For almost 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/deep-lrng-for-life-science*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*



Copyrighted material

## Acknowledgments

We would like to thank Nicole Tache, our editor at O'Reilly, as well as the tech reviewers and beta reviewers for their valuable contributions to the book. In addtion, we would like to thank Karl Leswing and Zhenqin (Michael) Wu for their contributions to the code and Johnny Israeli for valuable advice on the genomics chapter.

Bharath would like to thank his family for their support and encouragement during many long weekends and nights working on this book.

Peter would like to thank his wife for her constant support, as well as the many colleagues from whom he has learned so much about machine learning.

Pat would like to thank his wife Andrea, and his daughters Alee and Maddy, for their love and support. He would also like to acknowledge past and present colleagues at Vertex Pharmaceuticals and Relay Therapeutics, from whom he has learned so much.

Finally, we want to thank the DeepChem open source community for their encouragement and support throughout this project.



Copyrighted material

CHAPTER 1

# Why Life Science?

While there are many directions that those with a technical inclination and a passion for data can pursue, few areas can match the fundamental impact of biomedical research. The advent of modern medicine has fundamentally changed the nature of human existence. Over the last 20 years, we have seen innovations that have transformed the lives of countless individuals. When it first appeared in 1981, HIV/AIDS was a largely fatal disease. Continued development of antiretroviral therapies has dramatically extended the life expectancy for patients in the developed world. Other diseases, such as hepatitis C, which was considered largely untreatable a decade ago, can now be cured. Advances in genetics are enabling the identification and, hopefully soon, the treatment of a wide array of diseases. Innovations in diagnostics and instrumentation have enabled physicians to specifically identify and target disease in the human body. Many of these breakthroughs have benefited from and will continue to be advanced by computational methods.

## Why Deep Learning?

Machine learning algorithms are now a key component of everything from online shopping to social media. Teams of computer scientists are developing algorithms that enable digital assistants such as the Amazon Echo or Google Home to understand speech. Advances in machine learning have enabled routine on-the-fly translation of web pages between spoken languages. In addition to machine learning's impact on everyday life, it has impacted many areas of the physical and life sciences. Algorithms are being applied to everything from the detection of new galaxies from telescope images to the classification of subatomic interactions at the Large Hadron Collider.

One of the drivers of these technological advances has been the development of a class of machine learning methods known as deep neural networks. While the tech-



Copyrighted material

nological underpinnings of artificial neural networks were developed in the 1950s and refined in the 1980s, the true power of the technique wasn't fully realized until advances in computer hardware became available over the last 10 years. We will provide a more complete overview of deep neural networks in the next chapter, but it is important to acknowledge some of the advances that have occurred through the application of deep learning:

- Many of the developments in speech recognition that have become ubiquitous in cell phones, computers, televisions, and other internet-connected devices have been driven by deep learning.
- Image recognition is a key component of self-driving cars, internet search, and other applications. Many of the same developments in deep learning that drove consumer applications are now being used in biomedical research, for example, to classify tumor cells into different types.
- Recommender systems have become a key component of the online experience. Companies like Amazon use deep learning to drive their "customers who bought this also bought" approach to encouraging additional purchases. Netflix uses a similar approach to recommend movies that an individual may want to watch. Many of the ideas behind these recommender systems are being used to identify new molecules that may provide starting points for drug discovery efforts.
- Language translation was once the domain of very complex rule-based systems. Over the last few years, systems driven by deep learning have outperformed systems that had undergone years of manual curation. Many of the same ideas are now being used to extract concepts from the scientific literature and alert scientists to journal articles that they may have missed.

These are just a few of the innovations that have come about through the application of deep learning methods. We are at an interesting time when we have a convergence of widely available scientific data and methods for processing that data. Those with the ability to combine data with new methods for learning from patterns in that data can make significant scientific advances.

# Contemporary Life Science Is About Data

As mentioned previously, the fundamental nature of life science has changed. The availability of robotics and miniaturized experiments has brought about dramatic increases in the amount of experimental data that can be generated. In the 1980s a biologist would perform a single experiment and generate a single result. This sort of data could typically be manipulated by hand with the possible assistance of a pocket calculator. If we fast-forward to today's biology, we have instrumentation that is capable of generating millions of experimental data points in a day or two. Experiments



Copyrighted material

like gene sequencing, which can generate huge datasets, have become inexpensive and routine.

The advances in gene sequencing have led to the construction of databases that link an individual's genetic code to a multitude of health-related outcomes, including diabetes, cancer, and genetic diseases such as cystic fibrosis. By using computational techniques to analyze and mine this data, scientists are developing an understanding of the causes of these diseases and using this understanding to develop new treatments.

Disciplines that once relied primarily on human observation are now utilizing datasets that simply could not be analyzed manually. Machine learning is now routinely used to classify images of cells. The output of these machine learning models is used to identify and classify cancerous tumors and to evaluate the effects of potential disease treatments.

Advances in experimental techniques have led to the development of several databases that catalog the structures of chemicals and the effects that these chemicals have on a wide range of biological processes or activities. These structure–activity relationships (SARs) form the basis of a field known as chemical informatics, or *cheminformatics*. Scientists mine these large datasets and use the data to build predictive models that will drive the next generation of drug development.

With these large amounts of data comes a need for a new breed of scientist who is comfortable in both the scientific and computational domains. Those with these hybrid capabilities have the potential to unlock structure and trends in large datasets and to make the scientific discoveries of tomorrow.

## What Will You Learn?

In the first few chapters of this book, we provide an overview of deep learning and how it can be applied in the life sciences. We begin with machine learning, which has been defined as "the science (and art) of programming computers so that they can learn from data."[1]

Chapter 2 provides a brief introduction to deep learning. We begin with an example of how this type of machine learning can be used to perform a simple task like linear regression, and progress to more sophisticated models that are commonly used to solve real-world problems in the life sciences. Machine learning typically proceeds by initially splitting a dataset into a training set that is used to generate a model and a test set that is used to assess the performance of the model. In Chapter 2 we discuss

1 Furbush, James. "Machine Learning: A Quick and Simple Definition." *https://www.oreilly.com/ideas/machine-learning-a-quick-and-simple-definition*. 2018.



Copyrighted material

some of the details surrounding the training and validation of predictive models. Once a model has been generated, its performance can typically be optimized by varying a number of characteristics known as *hyperparameters*. The chapter provides an overview of this process. Deep learning is not a single technique, but a set of related methods. Chapter 2 concludes with an introduction to a few of the most important deep learning variants.

In Chapter 3, we introduce DeepChem, an open source programming library that has been specifically designed to simplify the creation of deep learning models for a variety of life science applications. After providing an overview of DeepChem, we introduce our first programming example, which demonstrates how the DeepChem library can be used to generate a model for predicting the toxicity of molecules. In a second programming example, we show how DeepChem can be used to classify images, a common task in modern biology. As briefly mentioned earlier, deep learning is used in a variety of imaging applications, ranging from cancer diagnosis to the detection of glaucoma. This discussion of specific applications then motivates an explanation of some of the inner workings of deep learning methods.

Chapter 4 provides an overview of how machine learning can be applied to molecules. We begin by introducing molecules, the building blocks of everything around us. Although molecules can be considered analogous to building blocks, they are not rigid. Molecules are flexible and exhibit dynamic behavior. In order to characterize molecules using a computational method like deep learning, we need to find a way to represent molecules in a computer. These encodings can be thought of as similar to the way in which an image can be represented as a set of pixels. In the second half of Chapter 4, we describe a number of ways that molecules can be represented and how these representations can be used to build deep learning models.

Chapter 5 provides an introduction to the field of biophysics, which applies the laws of physics to biological phenomena. We start with a discussion of proteins, the molecular machines that make life possible. A key component of predicting the effects of drugs on the body is understanding their interactions with proteins. In order to understand these effects, we begin with an overview of how proteins are constructed and how protein structures differ. Proteins are entities whose 3D structure dictates their biological function. For a machine learning model to predict the impact of a drug molecule on a protein's function, we need to represent that 3D structure in a form that can be processed by a machine learning program. In the second half of Chapter 5, we explore a number of ways that protein structures can be represented. With this knowledge in hand, we then review another code example where we use deep learning to predict the degree to which a drug molecule will interact with a protein.

Genetics has become a key component of contemporary medicine. The genetic sequencing of tumors has enabled the personalized treatment of cancer and has the



Copyrighted material

potential to revolutionize medicine. Gene sequencing, which used to be a complex process requiring huge investments, has now become commonplace and can be routinely carried out. We have even reached the point where dog owners can get inexpensive genetic tests to determine their pets' lineage. In Chapter 6, we provide an overview of genetics and genomics, beginning with an introduction to DNA and RNA, the templates that are used to produce proteins. Recent discoveries have revealed that the interactions of DNA and RNA are much more complex than originally believed. In the second half of Chapter 6, we present several code examples that demonstrate how deep learning can be used to predict a number of factors that influence the interactions of DNA and RNA.

Earlier in this chapter, we alluded to the many advances that have come about through the application of deep learning to the analysis of biological and medical images. Many of the phenomena studied in these experiments are too small to be observed by the human eye. In order to obtain the images used with deep learning methods, we need to utilize a microscope. Chapter 7 provides an overview of microscopy in its myriad forms, ranging from the simple light microscope we all used in school to sophisticated instruments that are capable of obtaining images at atomic resolution. This chapter also covers some of the limitations of current approaches, and provides information on the experimental pipelines used to obtain the images that drive deep learning models.

One area that offers tremendous promise is the application of deep learning to medical diagnosis. Medicine is incredibly complex, and no physician can personally embody all of the available medical knowledge. In an ideal situation, a machine learning model could digest the medical literature and aid medical professionals in making diagnoses. While we have yet to reach this point, a number of positive steps have been made. Chapter 8 begins with a history of machine learning methods for medical diagnosis and charts the transition from hand-encoded rules to statistical analysis of medical outcomes. As with many of the topics we've discussed, a key component is representing medical information in a format that can be processed by a machine learning program. In this chapter, we provide an introduction to electronic health records and some of the issues surrounding these records. In many cases, medical images can be very complex and the analysis and interpretation of these images can be difficult for even skilled human specialists. In these cases, deep learning can augment the skills of a human analyst by classifying images and identifying key features. Chapter 8 concludes with a number of examples of how deep learning is used to analyze medical images from a variety of areas.

As we mentioned earlier, machine learning is becoming a key component of drug discovery efforts. Scientists use deep learning models to evaluate the interactions between drug molecules and proteins. These interactions can elicit a biological response that has a therapeutic impact on a patient. The models we've discussed so far are *discriminative models*. Given a set of characteristics of a molecule, the model gen-



Copyrighted material

erates a prediction of some property. These predictions require an input molecule, which may be derived from a large database of available molecules or may come from the imagination of a scientist. What if, rather than relying on what currently exists, or what we can imagine, we had a computer program that could "invent" new molecules? Chapter 9 presents a type of deep learning program called a *generative model*. A generative model is initially trained on a set of existing molecules, then used to generate new molecules. The deep learning program that generates these molecules can also be influenced by other models that predict the activity of the new molecules.

Up to now, we have discussed deep learning models as "black boxes." We present the model with a set of input data and the model generates a prediction, with no explanation of how or why the prediction was generated. This type of prediction can be less than optimal in many situations. If we have a deep learning model for medical diagnosis, we often need to understand the reasoning behind the diagnosis. An explanation of the reasons for the diagnosis will provide a physician with more confidence in the prediction and may also influence treatment decisions. One historic drawback to deep learning has been the fact that the models, while often reliable, can be difficult to interpret. A number of techniques are currently being developed to enable users to better understand the factors that led to a prediction. Chapter 10 provides an overview of some of these techniques used to enable human understanding of model predictions. Another important aspect of predictive models is the accuracy of a model's predictions. An understanding of a model's accuracy can help us determine how much to rely on that model. Given that machine learning can be used to potentially make life-saving diagnoses, an understanding of model accuracy is critical. The final section of Chapter 10 provides an overview of some of the techniques that can be used to assess the accuracy of model predictions.

In Chapter 11 we present a real-world case study using DeepChem. In this example, we use a technique called virtual screening to identify potential starting points for the discovery of new drugs. Drug discovery is a complex process that often begins with a technique known as *screening*. Screening is used to identify molecules that can be optimized to eventually generate drugs. Screening can be carried out experimentally, where millions of molecules are tested in miniaturized biological tests known as assays, or in a computer using virtual screening. In virtual screening, a set of known drugs or other biologically active molecules is used to train a machine learning model. This machine learning model is then used to predict the activity of a large set of molecules. Because of the speed of machine learning methods, hundreds of millions of molecules can typically be processed in a few days of computer time.

The final chapter of the book examines the current impact and future potential of deep learning in the life sciences. A number of challenges for current efforts, including the availability and quality of datasets, are discussed. We also highlight opportunities and potential pitfalls in a number of other areas including diagnostics, personalized medicine, pharmaceutical development, and biology research.



Copyrighted material

CHAPTER 2

# Introduction to Deep Learning

The goal of this chapter is to introduce the basic principles of deep learning. If you already have lots of experience with deep learning, you should feel free to skim this chapter and then go on to the next. If you have less experience, you should study this chapter carefully as the material it covers will be essential to understanding the rest of the book.

In most of the problems we will discuss, our task will be to create a mathematical function:

$$\mathbf{y} = f(\mathbf{x})$$

Notice that **x** and **y** are written in bold. This indicates they are vectors. The function might take many numbers as input, perhaps thousands or even millions, and it might produce many numbers as outputs. Here are some examples of functions you might want to create:

- **x** contains the colors of all the pixels in an image. $f(\mathbf{x})$ should equal 1 if the image contains a cat and 0 if it does not.
- The same as above, except $f(\mathbf{x})$ should be a vector of numbers. The first element indicates whether the image contains a cat, the second whether it contains a dog, the third whether it contains an airplane, and so on for thousands of types of objects.
- **x** contains the DNA sequence for a chromosome. **y** should be a vector whose length equals the number of bases in the chromosome. Each element should equal 1 if that base is part of a region that codes for a protein, or 0 if not.
- **x** describes the structure of a molecule. (We will discuss various ways of representing molecules in later chapters.) **y** should be a vector where each element



Copyrighted material

describes some physical property of the molecule: how easily it dissolves in water, how strongly it binds to some other molecule, and so on.

As you can see, $f(\mathbf{x})$ could be a very, very complicated function! It usually takes a long vector as input and tries to extract information from it that is not at all obvious just from looking at the input numbers.

The traditional approach to solving this problem is to design a function by hand. You would start by analyzing the problem. What patterns of pixels tend to indicate the presence of a cat? What patterns of DNA tend to distinguish coding regions from noncoding ones? You would write computer code to recognize particular types of features, then try to identify combinations of features that reliably produce the result you want. This process is slow and labor-intensive, and depends heavily on the expertise of the person carrying it out.

Machine learning takes a totally different approach. Instead of designing a function by hand, you allow the computer to learn its own function based on data. You collect thousands or millions of images, each labeled to indicate whether it includes a cat. You present all of this training data to the computer, and let it search for a function that is consistently close to 1 for the images with cats and close to 0 for the ones without.

What does it mean to "let the computer search for a function"? Generally speaking, you create a *model* that defines some large class of functions. The model includes *parameters*, variables that can take on any value. By choosing the values of the parameters, you select a particular function out of all the many functions in the class defined by the model. The computer's job is to select values for the parameters. It tries to find values such that, when your training data is used as input, the output is as close as possible to the corresponding targets.

## Linear Models

One of the simplest models you might consider trying is a linear model:

$$\mathbf{y} = \mathbf{M}\mathbf{x} + \mathbf{b}$$

In this equation, $\mathbf{M}$ is a matrix (sometimes referred to as the "weights") and $\mathbf{b}$ is a vector (referred to as the "biases"). Their sizes are determined by the numbers of input and output values. If $\mathbf{x}$ has length T and you want $\mathbf{y}$ to have length S, then $\mathbf{M}$ will be an $S \times T$ matrix and $\mathbf{b}$ will be a vector of length S. Together, they make up the parameters of the model. This equation simply says that each output component is a linear combination of the input components. By setting the parameters ($\mathbf{M}$ and $\mathbf{b}$), you can choose any linear combination you want for each component.



Copyrighted material

This was one of the very earliest machine learning models. It was introduced back in 1957 and was called a *perceptron*. The name is an amazing piece of marketing: it has a science fiction sound to it and seems to promise wonderful things, when in fact it is nothing more than a linear transform. In any case, the name has managed to stick for more than half a century.

The linear model is very easy to formulate in a completely generic way. It has exactly the same form no matter what problem you apply it to. The only differences between linear models are the lengths of the input and output vectors. From there, it is just a matter of choosing the parameter values, which can be done in a straightforward way with generic algorithms. That is exactly what we want for machine learning: a model and algorithms that are independent of what problem you are trying to solve. Just provide the training data, and parameters are automatically determined that transform the generic model into a function that solves your problem.

Unfortunately, linear models are also very limited. As demonstrated in Figure 2-1, a linear model (in one dimension, that means a straight line) simply cannot fit most real datasets. The problem becomes even worse when you move to very high-dimensional data. No linear combination of pixel values in an image will reliably identify whether the image contains a cat. The task requires a much more complicated nonlinear model. In fact, any model that solves that problem will necessarily be *very* complicated and *very* nonlinear. But how can we formulate it in a generic way? The space of all possible nonlinear functions is infinitely complex. How can we define a model such that, just by choosing values of parameters, we can create almost any nonlinear function we are ever likely to want?

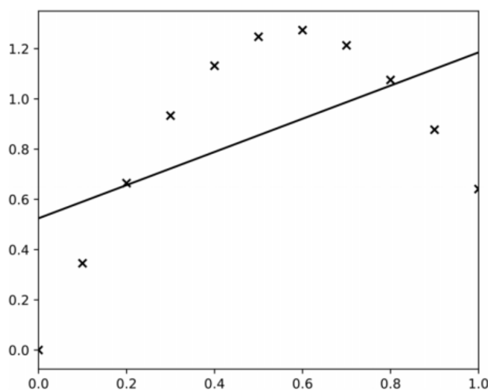



*Figure 2-1. A linear model cannot fit data points that follow a curve. This requires a nonlinear model.*



Copyrighted material

## Multilayer Perceptrons

A simple approach is to stack multiple linear transforms, one after another. For example, we could write:

$$\mathbf{y} = \mathbf{M}_2\varphi\left(\mathbf{M}_1\mathbf{x} + \mathbf{b}_1\right) + \mathbf{b}_2$$

Look carefully at what we have done here. We start with an ordinary linear transform, $\mathbf{M}_1\mathbf{x} + \mathbf{b}_1$. We then pass the result through a nonlinear function $\varphi(x)$, and then apply a second linear transform to the result. The function $\varphi(x)$, which is known as the *activation function*, is an essential part of what makes this work. Without it, the model would still be linear, and no more powerful than the previous one. A linear combination of linear combinations is itself nothing more than a linear combination of the original inputs! By inserting a nonlinearity, we enable the model to learn a much wider range of functions.

We don't need to stop at two linear transforms. We can stack as many as we want on top of each other:

$$\mathbf{h}_1 = \varphi_1\left(\mathbf{M}_1\mathbf{x} + \mathbf{b}_1\right)$$

$$\mathbf{h}_2 = \varphi_2\left(\mathbf{M}_2\mathbf{h}_1 + \mathbf{b}_2\right)$$

$$\ldots$$

$$\mathbf{h}_{n-1} = \varphi_{n-1}\left(\mathbf{M}_{n-1}\mathbf{h}_{n-2} + \mathbf{b}_{n-1}\right)$$

$$\mathbf{y} = \varphi_n\left(\mathbf{M}_n\mathbf{h}_{n-1} + \mathbf{b}_n\right)$$

This model is called a *multilayer perceptron*, or MLP for short. The middle steps $h_i$ are called *hidden layers*. The name refers to the fact that they are neither inputs nor outputs, just intermediate values used in the process of calculating the result. Also notice that we have added a subscript to each $\varphi(x)$. This indicates that different layers might use different nonlinearities.

You can visualize this calculation as a stack of layers, as shown in Figure 2-2. Each layer corresponds to a linear transformation followed by a nonlinearity. Information flows from one layer to another, the output of one layer becoming the input to the next. Each layer has its own set of parameters that determine how its output is calculated from its input.



Copyrighted material

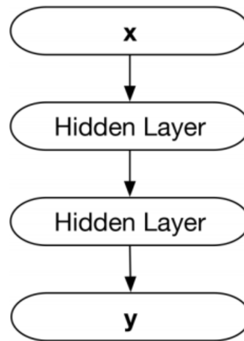



*Figure 2-2. A multilayer perceptron, viewed as a stack of layers with information flowing from one layer to the next.*

Multilayer perceptrons and their variants are also sometimes called *neural networks*. The name reflects the parallels between machine learning and neurobiology. A biological neuron connects to many other neurons. It receives signals from them, adds the signals together, and then sends out its own signals based on the result. As a very rough approximation, you can think of MLPs as working the same way as the neurons in your brain!

What should the activation function $\varphi(\mathbf{x})$ be? The surprising answer is that it mostly doesn't matter. Of course, that is not entirely true. It obviously does matter, but not as much as you might expect. Nearly any reasonable function (monotonic, reasonably smooth) can work. Lots of different functions have been tried over the years, and although some work better than others, nearly all of them can produce decent results.

The most popular activation function today is probably the *rectified linear unit* (ReLU), $\varphi(x) = \max(0, x)$. If you aren't sure what function to use, this is probably a good default. Other common choices include the *hyperbolic tangent*, $\tanh(x)$, and the *logistic sigmoid*, $\varphi(x) = 1/\left(1 + e^{-x}\right)$. All of these functions are shown in Figure 2-3.

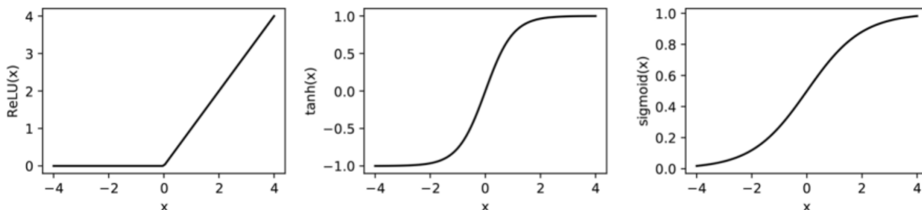



*Figure 2-3. Three common activation functions: the rectified linear unit, hyperbolic tangent, and logistic sigmoid.*

We also must choose two other properties for an MLP: its *width* and its *depth*. With the simple linear model, we had no choices to make. Given the lengths of **x** and **y**, the



Copyrighted material

sizes of **M** and **b** were completely determined. Not so with hidden layers. Width refers to the size of the hidden layers. We can choose each $\mathbf{h}_i$ to have any length we want. Depending on the problem, you might want them to be much larger or much smaller than the input and output vectors.

Depth refers to the number of layers in the model. A model with only one hidden layer is described as *shallow*. A model with many hidden layers is described as *deep*. This is, in fact, the origin of the term "deep learning"; it simply means "machine learning using models with lots of layers."

Choosing the number and widths of layers in your model involves as much art as science. Or, to put it more formally, "This is still an active field of research." Often it just comes down to trying lots of combinations and seeing what works. There are a few principles that may provide guidance, however, or at least help you understand your results in hindsight:

1. An MLP with one hidden layer is a *universal approximator*.

   This means it can approximate any function at all (within certain fairly reasonable limits). In a sense, you never need more than one hidden layer. That is already enough to reproduce any function you are ever likely to want. Unfortunately, this result comes with a major caveat: the accuracy of the approximation depends on the width of the hidden layer, and you may need a very wide layer to get sufficient accuracy for a given problem. This brings us to the second principle.

2. Deep models tend to require fewer parameters than shallow ones.

   This statement is intentionally somewhat vague. More rigorous statements can be proven for particular special cases, but it does still apply as a general guideline. Here is perhaps a better way of stating it: every problem requires a model with a certain depth to efficiently achieve acceptable accuracy. At shallower depths, the required widths of the layers (and hence the total number of parameters) increase rapidly. This makes it sound like you should always prefer deep models over shallow ones. Unfortunately, it is partly contradicted by the third principle.

3. Deep models tend to be harder to train than shallow ones.

   Until about 2007, most machine learning models were shallow. The theoretical advantages of deep models were known, but researchers were usually unsuccessful at training them. Since then, a series of advances has gradually improved the usefulness of deep models. These include better training algorithms, new types of models that are easier to train, and of course faster computers combined with larger datasets on which to train the models. These advances gave rise to "deep learning" as a field. Yet despite the improvements, the general principle remains true: deeper models tend to be harder to train than shallower ones.



Copyrighted material

## Training Models

This brings us to the next subject: just how do we train a model anyway? MLPs provide us with a (mostly) generic model that can be used for any problem. (We will discuss other, more specialized types of models a little later.) Now we want a similarly generic algorithm to find the optimal values of the model's parameters for a given problem. How do we do that?

The first thing you need, of course, is a collection of data to train it on. This dataset is known as the *training set*. It should consist of a large number of ($\mathbf{x}$,$\mathbf{y}$) pairs, also known as *samples*. Each sample specifies an input to the model, and what you want the model's output to be when given that input. For example, the training set could be a collection of images, along with labels indicating whether or not each image contains a cat.

Next you need to define a loss function $L(\mathbf{y}, \widehat{\mathbf{y}})$, where $\mathbf{y}$ is the actual output from the model and $\widehat{\mathbf{y}}$ is the target value specified in the training set. This is how you measure whether the model is doing a good job of reproducing the training data. It is then averaged over every sample in the training set:

$$\text{average loss} = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{y}_i, \widehat{\mathbf{y}}_i)$$

$L(\mathbf{y}, \widehat{\mathbf{y}})$ should be small when its arguments are close together and large when they are far apart. In other words, we take every sample in the training set, try using each one as an input to the model, and see how close the output is to the target value. Then we average this over the whole training set.

An appropriate loss function needs to be chosen for each problem. A common choice is the Euclidean distance (also known as the $L_2$ distance), $L(\mathbf{y}, \widehat{\mathbf{y}}) = \sqrt{\Sigma_i (y_i - \widehat{y}_i)^2}$. (In this expression, $y_i$ means the $i$'th component of the vector $\mathbf{y}$.) When $\mathbf{y}$ represents a probability distribution, a popular choice is the cross entropy, $L(\mathbf{y}, \widehat{\mathbf{y}}) = -\Sigma_i y_i \log \widehat{y}_i$. Other choices are also possible, and there is no universal "best" choice. It depends on the details of your problem.

Now that we have a way to measure how well the model works, we need a way to improve it. We want to search for the parameter values that minimize the average loss over the training set. There are many ways to do this, but most work in deep learning uses some variant of the *gradient descent* algorithm. Let $\theta$ represent the set of all parameters in the model. Gradient descent involves taking a series of small steps:

$$\theta \leftarrow \theta - \epsilon \frac{\partial}{\partial \theta} \langle L \rangle$$



Copyrighted material

where $\langle L \rangle$ is the average loss over the training set. Each step moves a tiny distance in the "downhill" direction. It changes each of the model's parameters by a little bit, with the goal of causing the average loss to decrease. If all the stars align and the phase of the moon is just right, this will eventually produce parameters that do a good job of solving your problem. $\epsilon$ is called the *learning rate*, and it determines how much the parameters change on each step. It needs to be chosen very carefully: too small a value will cause learning to be very slow, while too large a value will prevent the algorithm from learning at all.

This algorithm really does work, but it has a serious problem. For every step of gradient descent, we need to loop over every sample in the training set. That means the time required to train the model is proportional to the size of the training set! Suppose that you have one million samples in the training set, that computing the gradient of the loss for one sample requires one million operations, and that it takes one million steps to find a good model. (All of these numbers are fairly typical of real deep learning applications.) Training will then require *one quintillion* operations. That takes quite a long time, even on a fast computer.

Fortunately, there is a better solution: estimate $\langle L \rangle$ by averaging over a much smaller number of samples. This is the basis of the *stochastic gradient descent* (SGD) algorithm. For every step, we take a small set of samples (known as a *batch*) from the training set and compute the gradient of the loss function, averaged over only the samples in the batch. We can view this as an estimate of what we would have gotten if we had averaged over the entire training set, although it may be a very noisy estimate. We perform a single step of gradient descent, then select a new batch of samples for the next step.

This algorithm tends to be much faster. The time required for each step depends only on the size of each batch, which can be quite small (often on the order of 100 samples) and is independent of the size of the training set. The disadvantage is that each step does a less good job of reducing the loss, because it is based on a noisy estimate of the gradient rather than the true gradient. Still, it leads to a much shorter training time overall.

Most optimization algorithms used in deep learning are based on SGD, but there are many variations that improve on it in different ways. Fortunately, you can usually treat these algorithms as black boxes and trust them to do the right thing without understanding all the details of how they work. Two of the most popular algorithms used today are called Adam and RMSProp. If you are in doubt about what algorithm to use, either one of those will probably be a reasonable choice.



Copyrighted material

## Validation

Suppose you have done everything described so far. You collected a large set of training data. You selected a model, then ran a training algorithm until the loss became very small. Congratulations, you now have a function that solves your problem!

Right?

Sorry, it's not that simple! All you really know for sure is that the function works well *on the training data*. You might hope it will also work well on other data, but you certainly can't count on it. Now you need to validate the model to see whether it works on data that it hasn't been specifically trained on.

To do this you need a second dataset, called the *test set*. It has exactly the same form as the training set, a collection of $(\mathbf{x}, \mathbf{y})$ pairs, but the two should have no samples in common. You train the model on the training set, then test it on the test set. This brings us to one of the most important principles in machine learning:

- You must not use the test set in any way while designing or training the model.

In fact, it is best if you never even look at the data in the test set. Test set data is only for testing the fully trained model to find out how well it works. If you allow the test set to influence the model in any way, you risk getting a model that works better on the test set than on other data that was not involved in creating the model. It ceases to be a true test set, and becomes just another type of training set.

This is connected to the mathematical concept of *overfitting*. The training data is supposed to be representative of a much larger data distribution, the set of all inputs you might ever want to use the model on. But you can't train it on all possible inputs. You can only create a finite set of training samples, train the model on those, and hope it learns general strategies that work equally well on other samples. Overfitting is what happens when the training picks up on specific features of the training samples, such that the model works better on them than it does on other samples.

## Regularization

Overfitting is a major problem for anyone who uses machine learning. Given that, you won't be surprised to learn that lots of techniques have been developed for avoiding it. These techniques are collectively known as *regularization*. The goal of any regularization technique is to avoid overfitting and produce a trained model that works well on any input, not just the particular inputs that were used for training.

Before we discuss particular regularization techniques, there are two very important points to understand about it.



Copyrighted material

First, the best way to avoid overfitting is *almost always* to get more training data. The bigger your training set, the better it represents the "true" data distribution, and the less likely the learning algorithm is to overfit. Of course, that is sometimes impossible: maybe you simply have no way to get more data, or the data may be very expensive to collect. In that case, you just have to do the best you can with the data you have, and if overfitting is a problem, you will have to use regularization to avoid it. But more data will probably lead to a better result than regularization.

Second, there is no universally "best" way to do regularization. It all depends on the problem. After all, the training algorithm doesn't know that it's overfitting. All it knows about is the training data. It doesn't know how the true data distribution differs from the training data, so the best it can do is produce a model that works well on the training set. If that isn't what you want, it's up to you to tell it.

That is the essence of any regularization method: biasing the training process to prefer certain types of models over others. You make assumptions about what properties a "good" model should have, and how it differs from an overfit one, and then you tell the training algorithm to prefer models with those properties. Of course, those assumptions are often implicit rather than explicit. It may not be obvious what assumptions you are making by choosing a particular regularization method. But they are always there.

One of the simplest regularization methods is just to train the model for fewer steps. Early in training, it tends to pick up on coarse properties of the training data that likely apply to the true distribution. The longer it runs, the more likely it is to start picking up on fine details of particular training samples. By limiting the number of training steps, you give it less opportunity to overfit. More formally, you are really assuming that "good" parameter values should not be too different from whatever values you start training from.

Another method is to restrict the magnitude of the parameters in the model. For example, you might add a term to the loss function that is proportional to $|\theta|^2$, where $\theta$ is a vector containing all of the model's parameters. By doing this, you are assuming that "good" parameter values should not be any larger than necessary. It reflects the fact that overfitting often (though not always) involves some parameters becoming very large.

A very popular method of regularization is called *dropout*. It involves doing something that at first seems ridiculous, but actually works surprisingly well. For each hidden layer in the model, you randomly select a subset of elements in the output vector $h_i$ and set them to 0. On every step of gradient descent, you pick a different random subset of elements. This might seem like it would just break the model: how can you expect it to work when internal calculations keep randomly getting set to 0? The mathematical theory for why dropout works is a bit complicated. Very roughly speaking, by using dropout you are assuming that no individual calculation within the



Copyrighted material

model should be too important. You should be able to randomly remove any individual calculation, and the rest of the model should continue to work without it. This forces it to learn redundant, highly distributed representations of data that make overfitting unlikely. If you are unsure of what regularization method to use, dropout is a good first thing to try.

## Hyperparameter Optimization

By now you have probaly noticed that there are a lot of choices to make, even when using a supposedly generic model with a "generic" learning algorithm. Examples include:

- The number of layers in the model
- The width of each layer
- The number of training steps to perform
- The learning rate to use during training
- The fraction of elements to set to 0 when using dropout

These options are called *hyperparameters*. A hyperparameter is any aspect of the model or training algorithm that must be set in advance rather than being learned by the training algorithm. But how are you supposed to choose them—and isn't the whole point of machine learning to select settings automatically based on data?

This brings us to the subject of *hyperparameter optimization*. The simplest way of doing it is just to try lots of values for each hyperparameter and see what works best. This becomes very expensive when you want to try lots of values for lots of hyperparameters, so there are more sophisticated approaches, but the basic idea remains the same: try different combinations and see what works best.

But how can you tell what works best? The simplest answer would be to just see what produces the lowest value of the loss function (or some other measure of accuracy) on the training set. But remember, that isn't what we really care about. We want to minimize error on the test set, not the training set. This is especially important for hyperparameters that affect regularization, such as the dropout rate. A low training set error might just mean the model is overfitting, optimizing for the precise details of the training data. So instead we want to try lots of hyperparameter values, then use the ones that minimize the loss on the test set.

But we mustn't do that! Remember: you must not use the test set in any way while designing or training the model. Its job is to tell you how well the model is likely to work on new data it has never seen before. Just because a particular set of hyperparameters happens to work best on the test set doesn't guarantee those values will always



Copyrighted material

work best. We must not allow the test set to influence the model, or it is no longer an unbiased test set.

The solution is to create yet another dataset, which is called the *validation set*. It must not share any samples with either the training set or the test set. The full procedure now works as follows:

1. For each set of hyperparameter values, train the model on the training set, then compute the loss on the validation set.
2. Whichever set of hyperparameters give the lowest loss on the validation set, accept them as your final model.
3. Evaluate that final model on the test set to get an unbiased measure of how well it works.

## Other Types of Models

This still leaves one more decision you need to make, and it is a huge subject in itself: what kind of model to use. Earlier in this chapter we introduced multilayer perceptrons. They have the advantage of being a generic class of models that can be applied to many different problems. Unfortunately, they also have serious disadvantages. They require a huge number of parameters, which makes them very susceptible to overfitting. They become difficult to train when they have more than one or two hidden layers. In many cases, you can get a better result by using a less generic model that takes advantage of specific features of your problem.

Much of the content of this book consists of discussing particular types of models that are especially useful in the life sciences. Those can wait until later chapters. But for the purposes of this introduction, there are two very important classes of models we should discuss that are widely used in many different fields. They are called convolutional neural networks and recurrent neural networks.

### Convolutional Neural Networks

*Convolutional neural networks* (CNNs for short) were one of the very first classes of deep models to be widely used. They were developed for use in image processing and computer vision. They remain an excellent choice for many kinds of problems that involve continuous data sampled on a rectangular grid: audio signals (1D), images (2D), volumetric MRI data (3D), and so on.

They are also a class of models that truly justify the term "neural network." The design of CNNs was originally inspired by the workings of the feline visual cortex. (Cats have played a central role in deep learning from the dawn of the field.) Research performed from the 1950s to the 1980s revealed that vision is processed through a



Copyrighted material

series of layers. Each neuron in the first layer takes input from a small region of the visual field (its *receptive field*). Different neurons are specialized to detect particular local patterns or features, such as vertical or horizontal lines. Cells in the second layer take input from local clusters of cells in the first layer, combining their signals to detect more complicated patterns over a larger receptive field. Each layer can be viewed as a new representation of the original image, described in terms of larger and more abstract patterns than the ones in the previous layer.

CNNs mirror this design, sending an input image through a series of layers. In that sense, they are just like MLPs, but the structure of each layer is very different. MLPs use *fully connected layers*. Every element of the output vector depends on every element of the input vector. CNNs use *convolutional layers* that take advantage of spatial locality. Each output element corresponds to a small region of the image, and only depends on the input values in that region. This enormously reduces the number of parameters defining each layer. In effect, it assumes that most elements of the weight matrix $\mathbf{M}_i$ are 0, since each output element only depends on a small number of input elements.

Convolutional layers take this a step further: they assume the parameters are the same for *every local region of the image*. If a layer uses one set of parameters to detect horizontal lines at one location in the image, it also uses exactly the same parameters to detect horizontal lines everywhere else in the image. This makes the number of parameters for the layer independent of the size of the image. All it has to learn is a single *convolutional kernel* that defines how output features are computed from any local region of the image. That local region is often very small, perhaps 5 by 5 pixels. In that case, the number of parameters to learn is only 25 times the number of output features for each region. This is tiny compared to the number in a fully connected layer, making CNNs much easier to train and much less susceptible to overfitting than MLPs.

## Recurrent Neural Networks

*Recurrent neural networks* (RNNs for short) are a bit different. They are normally used to process data that takes the form of a sequence of elements: words in a text document, bases in a DNA molecule, etc. The elements in the sequence are fed into the network's input one at a time. But then the network does something very different: the output from each layer is fed back into its own input on the next step! This allows RNNs to have a sort of memory. When an element (word, DNA base, etc.) from the sequence is fed into the network, the input to each layer depends on that element, but also on all of the previous elements (Figure 2-4).



Copyrighted material

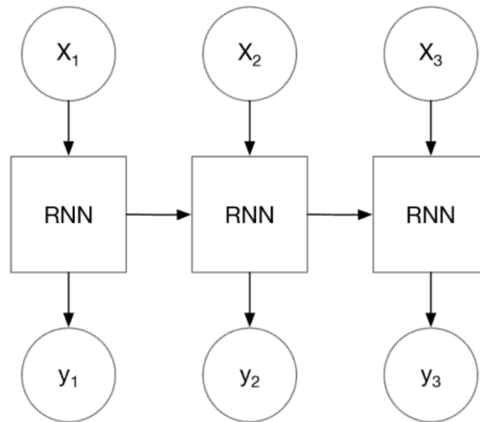



*Figure 2-4. A recurrent neural network. As each element ($x_1$, $x_2$, ...) of the sequence is fed into the input, the output ($y_1$, $y_2$, ...) depends both on the input element and on the RNN's own output during the previous step.*

So, the input to a recurrent layer has two parts: the regular input (that is, the output from the previous layer in the network) and the recurrent input (which equals its own output from the previous step). It then needs to calculate a new output based on those inputs. In principle you could use a fully connected layer, but in practice that usually doesn't work very well. Researchers have developed other types of layers that work much better in RNNs. The two most popular ones are called the *gated recurrent unit* (GRU) and the *long short-term memory* (LSTM). Don't worry about the details for now; just remember that if you are creating an RNN, you should usually build it out of one of those types of layers.

Having memory makes RNNs fundamentally different from the other models we have discussed. With a CNN or MLP, you simply feed a value into the network's input and get a different value out. The output is entirely determined by the input. Not so with an RNN. The model has its own internal state, composed of the outputs of all its layers from the most recent step. Each time you feed a new value into the model, the output depends not just on the input value but also on the internal state. Likewise, the internal state is altered by each new input value. This makes RNNs very powerful, and allows them to be used for lots of different applications.



Copyrighted material

## Further Reading

Deep learning is a huge subject, and this chapter has only given the briefest introduction to it. It should be enough to help you read and understand the rest of this book, but if you plan to do serious work in the field, you will want to acquire a much more thorough background. Fortunately, there are many excellent deep learning resources available online. Here are some suggestions for material you might consult:

- *Neural Networks and Deep Learning* by Michael Nielsen (Determination Press) covers roughly the same material as this chapter, but goes into far more detail on every subject. If you want a solid working knowledge of the fundamentals of deep learning, sufficient to make use of it in your own work, this is an excellent place to start.
- *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (MIT Press) is a more advanced introduction written by some of the top researchers in the field. It expects the reader to have a background similar to that of a graduate student in computer science and goes into far more detail on the mathematical theory behind the subject. You can easily use deep models without understanding all of the theory, but if you want to do original research in deep learning (rather than just using deep models as a tool to solve problems in other fields), this book is a fantastic resource.
- *TensorFlow for Deep Learning* by Bharath Ramsundar and Reza Zadeh (O'Reilly) provides a practitioner's introduction to deep learning that seeks to build intuition about the core concepts without delving too deeply into the mathematical underpinnings of such models. It might be a useful reference for those who are interested in the practical aspects of deep learning.



Copyrighted material

Copyrighted material

## CHAPTER 3

# Machine Learning with DeepChem

This chapter provides a brief introduction to machine learning with DeepChem, a library built on top of the TensorFlow platform to facilitate the use of deep learning in the life sciences. DeepChem provides a large collection of models, algorithms, and datasets that are suited to applications in the life sciences. In the remainder of this book, we will use DeepChem to perform our case studies.

#### Why Not Just Use Keras, TensorFlow, or PyTorch?

This is a common question. The short answer is that the developers of these packages focus their attention on supporting certain types of use cases that prove useful to their core users. For example, there's extensive support for image processing, text handling, and speech analysis. But there's often not a similar level of support in these libraries for molecule handling, genetic datasets, or microscopy datasets. The goal of DeepChem is to give these applications first-class support in the library. This means adding custom deep learning primitives, support for needed file types, and extensive tutorials and documentation for these use cases.

DeepChem is also designed to be well integrated with the TensorFlow ecosystem, so you should be able to mix and match DeepChem code with your other TensorFlow application code.

In the rest of this chapter, we will assume that you have DeepChem installed on your machine and that you are ready to run the examples. If you don't have DeepChem installed, never fear. Just head over to the DeepChem website and follow the installation directions for your system.



Copyrighted material

**Windows Support for DeepChem**

At present, DeepChem doesn't support installation on Windows. If possible, we recommend that you work through the examples in this book using a Mac or Linux workstation. We have heard from our users that DeepChem works on the Windows Subsystem for Linux (WSL) in more modern Windows distributions.

If it's not feasible for you to get access to a Mac or Linux machine or work with WSL, we'd love to have your help getting Windows support for DeepChem. Please contact the authors with the specific issues you're seeing, and we will try to address them. Our hope is to remove this restriction in a future edition of the book and support Windows for future readers.

## DeepChem Datasets

DeepChem uses the basic abstraction of the`Dataset` object to wrap the data it uses for machine learning. A `Dataset` contains the information about a set of samples: the input vectors `x`, the target output vectors `y`, and possibly other information such as a description of what each sample represents. There are subclasses of `Dataset` corresponding to different ways of storing the data. The `NumpyDataset` object in particular serves as a convenient wrapper for NumPy arrays and will be used extensively. In this section, we will walk through a simple code case study of how to use `NumpyDataset`. All of this code can be entered in the interactive Python interpreter; where appropriate, the output is shown.

We start with some simple imports:

```
import deepchem as dc
import numpy as np
```

Let's now construct some simple NumPy arrays:

```
x = np.random.random((4, 5))
y = np.random.random((4, 1))
```

This dataset will have four samples. The array `x` has five elements ("features") for each sample, and `y` has one element for each sample. Let's take a quick look at the actual arrays we've sampled (note that when you run this code locally, you should expect to see different numbers since your random seed will be different):

```
In : x
Out:
array([[0.960767 , 0.31300931, 0.23342295, 0.59850938, 0.30457302],
   [0.48891533, 0.69610528, 0.02846666, 0.20008034, 0.94781389],
   [0.17353084, 0.95867152, 0.73392433, 0.47493093, 0.4970179 ],
   [0.15392434, 0.95759308, 0.72501478, 0.38191593, 0.16335888]])
```



Copyrighted material

```
In : y
Out:
array([[0.00631553],
   [0.69677301],
   [0.16545319],
   [0.04906014]])
```

Let's now wrap these arrays in a `NumpyDataset` object:

```
dataset = dc.data.NumpyDataset(x, y)
```

We can unwrap the `dataset` object to get at the original arrays that we stored inside:

```
In : print(dataset.X)
[[0.960767 0.31300931 0.23342295 0.59850938 0.30457302]
[0.48891533 0.69610528 0.02846666 0.20008034 0.94781389]
[0.17353084 0.95867152 0.73392433 0.47493093 0.4970179 ]
[0.15392434 0.95759308 0.72501478 0.38191593 0.16335888]]

In : print(dataset.y)
[[0.00631553]
[0.69677301]
[0.16545319]
[0.04906014]]
```

Note that these arrays are the same as the original arrays `x` and `y`:

```
In : np.array_equal(x, dataset.X)
Out : True

In : np.array_equal(y, dataset.y)
Out : True
```

**Other Types of Datasets**

DeepChem has support for other types of `Dataset` objects, as mentioned previously. These types primarily become useful when dealing with larger datasets that can't be entirely stored in computer memory. There is also integration for DeepChem to use TensorFlow's `tf.data` dataset loading utilities. We will touch on these more advanced library features as we need them.

## Training a Model to Predict Toxicity of Molecules

In this section, we will demonstrate how to use DeepChem to train a model to predict the toxicity of molecules. In a later chapter, we will explain how toxicity prediction for molecules works in much greater depth, but in this section, we will treat it as a black-box example of how DeepChem models can be used to solve machine learning challenges. Let's start with a pair of needed imports:



Copyrighted material

```python
import numpy as np
import deepchem as dc
```

The next step is loading the associated toxicity datasets for training a machine learning model. DeepChem maintains a module called `dc.molnet` (short for MoleculeNet) that contains a number of preprocessed datasets for use in machine learning experimentation. In particular, we will make use of the `dc.molnet.load_tox21()` function, which will load and process the Tox21 toxicity dataset for us. When you run these commands for the first time, DeepChem will process the dataset locally on your machine. You should expect to see processing notes like the following:

```
In : tox21_tasks, tox21_datasets, transformers = dc.molnet.load_tox21()
Out: Loading raw samples now.
shard_size: 8192
About to start loading CSV from /tmp/tox21.CSV.gz
Loading shard 1 of size 8192.
Featurizing sample 0
Featurizing sample 1000
Featurizing sample 2000
Featurizing sample 3000
Featurizing sample 4000
Featurizing sample 5000
Featurizing sample 6000
Featurizing sample 7000
TIMING: featurizing shard 0 took 15.671 s
TIMING: dataset construction took 16.277 s
Loading dataset from disk.
TIMING: dataset construction took 1.344 s
Loading dataset from disk.
TIMING: dataset construction took 1.165 s
Loading dataset from disk.
TIMING: dataset construction took 0.779 s
Loading dataset from disk.
TIMING: dataset construction took 0.726 s
Loading dataset from disk.
```

The process of *featurization* is how a dataset containing information about molecules is transformed into matrices and vectors for use in machine learning analyses. We will explore this process in greater depth in subsequent chapters. Let's start here, though, by taking a quick peek at the data we've processed.

The `dc.molnet.load_tox21()` function returns multiple outputs: `tox21_tasks`, `tox21_datasets`, and `transformers`. Let's briefly take a look at each:

```
In : tox21_tasks
Out:
['NR-AR',
'NR-AR-LBD',
'NR-AhR',
'NR-Aromatase',
'NR-ER',
```



Copyrighted material

```
'NR-ER-LBD',
'NR-PPAR-gamma',
'SR-ARE',
'SR-ATAD5',
'SR-HSE',
'SR-MMP',
'SR-p53']

In : len(tox21_tasks)
Out: 12
```

Each of the 12 tasks here corresponds with a particular biological experiment. In this case, each of these tasks is for an *enzymatic assay* which measures whether the molecules in the Tox21 dataset bind with the *biological target* in question. The terms `NR-AR` and so on correspond with these targets. In this case, each of these targets is a particular enzyme believed to be linked to toxic responses to potential therapeutic molecules.

### How Much Biology Do I Need to Know?

For computer scientists and engineers entering the life sciences, the array of biological terms can be dizzying. However, it's not necessary to have a deep understanding of biology in order to begin making an impact in the life sciences. If your primary background is in computer science, it can be useful to try understanding biological systems in terms of computer scientific analogues. Imagine that cells or animals are complex legacy codebases that you have no control over. As an engineer, you have a few experimental measurements of these systems (assays) which you can use to gain some understanding of the underlying mechanics. Machine learning is an extraordinarily powerful tool for understanding biological systems since learning algorithms are capable of extracting useful correlations in a mostly automatic fashion. This allows even biological beginners to sometimes find deep biological insights.

In the remainder of this book, we discuss basic biology in brief asides. These notes can serve as entry points into the vast biological literature. Public references such as Wikipedia often contain a wealth of useful information, and can help bootstrap your biological education.

Next, let's consider `tox21_datasets`. The use of the plural is a clue that this field is actually a tuple containing multiple`dc.data.Dataset` objects:

```
In : tox21_datasets
Out:
(<deepchem.data.datasets.DiskDataset at 0x7f9804d6c390>,
<deepchem.data.datasets.DiskDataset at 0x7f9804d6c780>,
<deepchem.data.datasets.DiskDataset at 0x7f9804c5a518>)
```



Copyrighted material

In this case, these datasets correspond to the training, validation, and test sets you learned about in the previous chapter. You might note that these are `DiskDataset` objects; the `dc.molnet` module caches these datasets on your disk so that you don't need to repeatedly refeaturize the Tox21 dataset. Let's split up these datasets correctly:

```
train_dataset, valid_dataset, test_dataset = tox21_datasets
```

When dealing with new datasets, it's very useful to start by taking a look at their shapes. To do so, inspect the `shape` attribute:

```
In : train_dataset.X.shape
Out: (6264, 1024)

In : valid_dataset.X.shape
Out: (783, 1024)

In : test_dataset.X.shape
Out: (784, 1024)
```

The `train_dataset` contains a total of 6,264 samples, each of which has an associated feature vector of length 1,024. Similarly, `valid_dataset` and `test_dataset`contain respectively 783 and 784 samples. Let's now take a quick look at the `y` vectors for these datasets:

```
In : np.shape(train_dataset.y)
Out: (6264, 12)

In : np.shape(valid_dataset.y)
Out: (783, 12)

In : np.shape(test_dataset.y)
Out: (784, 12)
```

There are 12 data points, also known as *labels*, for each sample. These correspond to the 12 tasks we discussed earlier. In this particular dataset, the samples correspond to molecules, the tasks correspond to biochemical assays, and each label is the result of a particular assay on a particular molecule. Those are what we want to train our model to predict.

There's a complication, however: the actual experimental dataset for Tox21 did not test every molecule in every biological experiment. That means that some of these labels are meaningless placeholders. We simply don't have any data for some properties of some molecules, so we need to ignore those elements of the arrays when training and testing the model.

How can we find which labels were actually measured? We can check the dataset's `w` field, which records its *weights*. Whenever we compute the loss function for a model, we multiply by `w` before summing over tasks and samples. This can be used for a few purposes, one being to flag missing data. If a label has a weight of 0, that label does



Copyrighted material

not affect the loss and is ignored during training. Let's do some digging to find how many labels have actually been measured in our datasets:

```
In : train_dataset.w.shape
Out: (6264, 12)

In : np.count_nonzero(train_dataset.w)
Out: 62166

In : np.count_nonzero(train_dataset.w == 0)
Out: 13002
```

Of the 6,264 × 12 = 75,168 elements in the array of labels, only 62,166 were actually measured. The other 13,002 correspond to missing measurements and should be ignored. You might ask, then, why we still keep such entries around. The answer is mainly for convenience; irregularly shaped arrays are much harder to reason about and deal with in code than regular matrices with an associated set of weights.

### Processing Datasets Is Challenging

It's important to note here that cleaning and processing a dataset for use in the life sciences can be extremely challenging. Many raw datasets will contain systematic classes of errors. If the dataset in question has been constructed from an experiment conducted by an external organization (a contract research organization, or CRO), it's quite possible that the dataset will be systematically wrong. For this reason, many life science organizations maintain scientists in-house whose job it is to verify and clean such datasets.

In general, if your machine learning algorithm isn't working for a life science task, there's a significant chance that the root cause stems not from the algorithm but from systematic errors in the source of data that you're using.

Now let's examine `transformers`, the final output that was returned by `load_tox21()`. A *transformer* is an object that modifies a dataset in some way. DeepChem provides many transformers that manipulate data in useful ways. The data-loading routines found in MoleculeNet always return a list of transformers that have been applied to the data, since you may need them later to "untransform" the data. Let's see what we have in this case:

```
In : transformers
Out: [<deepchem.trans.transformers.BalancingTransformer at 0x7f99dd73c6d8>]
```

Here, the data has been transformed with a `BalancingTransformer`. This class is used to correct for unbalanced data. In the case of Tox21, most molecules do not bind to most of the targets. In fact, over 90% of the labels are 0. That means a model could trivially achieve over 90% accuracy simply by always predicting 0, no matter what



Copyrighted material

input it was given. Unfortunately, that model would be completely useless! Unbalanced data, where there are many more training samples for some classes than others, is a common problem in classification tasks.

Fortunately, there is an easy solution: adjust the dataset's matrix of weights to compensate. `BalancingTransformer` adjusts the weights for individual data points so that the total weight assigned to every class is the same. That way, the loss function has no systematic preference for any one class. The loss can only be decreased by learning to correctly distinguish between classes.

Now that we've explored the Tox21 datasets, let's start exploring how we can train models on these datasets. DeepChem's `dc.models` submodule contains a variety of different life science–specific models. All of these various models inherit from the parent class `dc.models.Model`. This parent class is designed to provide a common API that follows common Python conventions. If you've used other Python machine learning packages, you should find that many of the `dc.models.Model` methods look quite familiar.

In this chapter, we won't really dig into the details of how these models are constructed. Rather, we will just provide an example of how to instantiate a standard DeepChem model, `dc.models.MultitaskClassifier`. This model builds a fully connected network (an MLP) that maps input features to multiple output predictions. This makes it useful for *multitask* problems, where there are multiple labels for every sample. It's well suited for our Tox21 datasets, since we have a total of 12 different assays we wish to predict simultaneously. Let's see how we can construct a `MultitaskClassifier` in DeepChem:

```
model = dc.models.MultitaskClassifier(n_tasks=12,
n_features=1024,
layer_sizes=[1000])
```

There are a variety of different options here. Let's briefly review them. `n_tasks` is the number of tasks, and `n_features` is the number of input features for each sample. As we saw earlier, the Tox21 dataset has 12 tasks and 1,024 features for each sample. `layer_sizes` is a list that sets the number of fully connected hidden layers in the network, and the width of each one. In this case, we specify that there is a single hidden layer of width 1,000.

Now that we've constructed the model, how can we train it on the Tox21 datasets? Each `Model` object has a `fit()` method that fits the model to the data contained in a `Dataset` object. Fitting our `MultitaskClassifier` object is then a simple call:

```
model.fit(train_dataset, nb_epoch=10)
```

Note that we added on a flag here. `nb_epoch=10` says that 10 epochs of gradient descent training will be conducted. An *epoch* refers to one complete pass through all the samples in a dataset. To train a model, you divide the training set into batches and



Copyrighted material

take one step of gradient descent for each batch. In an ideal world, you would reach a well-optimized model before running out of data. In practice, there usually isn't enough training data for that, so you run out of data before the model is fully trained. You then need to start reusing data, making additional passes through the dataset. This lets you train models with smaller amounts of data, but the more epochs you use, the more likely you are to end up with an overfit model.

Let's now evaluate the performance of the trained model. In order to evaluate how well a model works, it is necessary to specify a metric. The DeepChem class `dc.metrics.Metric` provides a general way to specify metrics for models. For the Tox21 datasets, the ROC AUC score is a useful metric, so let's do our analysis using it. However, note a subtlety here: there are multiple Tox21 tasks. Which one do we compute the ROC AUC on? A good tactic is to compute the mean ROC AUC score across all tasks. Luckily, it's easy to do this:

```
metric = dc.metrics.Metric(dc.metrics.roc_auc_score, np.mean)
```

Since we've specified `np.mean`, the mean of the ROC AUC scores across all tasks will be reported. DeepChem models support the evaluation function `model.evaluate()`, which evaluates the performance of the model on a given dataset and metric:

**ROC AUC**

We want to classify molecules as toxic or nontoxic, but the model outputs continuous numbers, not discrete predictions. In practice, you pick a threshold value and predict that a molecule is toxic whenever the output is greater than the threshold. A low threshold will produce many false positives (predicting a safe molecule is actually toxic). A higher threshold will give fewer false positives but more false negatives (incorrectly predicting that a toxic molecule is safe).

The *receiver operating characteristic* (ROC) curve is a convenient way to visualize this trade-off. You try many different threshold values, then plot a curve of the true positive rate versus the false positive rate as the threshold is varied. An example is shown in Figure 3-1.

The ROC AUC is the total area under the ROC curve. The *area under the curve* (AUC) provides an indication of the model's ability to distinguish different classes. If there exists any threshold value for which every sample is classified correctly, the ROC AUC score is 1. At the other extreme, if the model outputs completely random values unrelated to the true classes, the ROC AUC score is 0.5. This makes it a useful number for summarizing how well a classifier works. It's just a heuristic, but it's a popular one.



Copyrighted material

```
train_scores = model.evaluate(train_dataset, [metric], transformers)
test_scores = model.evaluate(test_dataset, [metric], transformers)
```

Now that we've calculated the scores, let's take a look!

```
In : print(train_scores)
...: print(test_scores)
Out
{'mean-roc_auc_score': 0.9659541853946179}
{'mean-roc_auc_score': 0.7915464001982299}
```

Notice that our score on the training set (0.96) is much better than our score on the test set (0.79). This shows the model has been overfit. The test set score is the one we really care about. These numbers aren't the best possible on this dataset—at the time of writing, the state of the art ROC AUC scores for the Tox21 dataset are a little under 0.9—but they aren't bad at all for an out-of-the-box system. The complete ROC curve for one of the 12 tasks is shown in Figure 3-1.

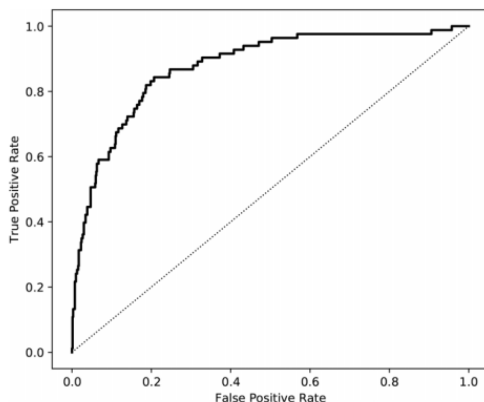



*Figure 3-1. The ROC curve for one of the 12 tasks. The dotted diagonal line shows what the curve would be for a model that just guessed at random. The actual curve is consistently well above the diagonal, showing that we are doing much better than random guessing.*

## Case Study: Training an MNIST Model

In the previous section, we covered the basics of training a machine learning model with DeepChem. However, we used a premade model class, `dc.models.MultitaskClassifier`. Sometimes you may want to create a new deep learning architecture instead of using a preconfigured one. In this section, we discuss how to train a convolutional neural network on the MNIST digit recognition dataset. Instead of using a premade architecture like in the previous example, this time we will specify the full deep learning architecture ourselves. To do so, we will introduce the



Copyrighted material

`dc.models.TensorGraph` class, which provides a framework for building deep architectures in DeepChem.

**When Do Canned Models Make Sense?**

In this section, we're going to use a custom architecture on MNIST. In the previous example, we used a "canned" (that is, predefined) architecture instead. When does each alternative make sense? If you have a well-debugged canned architecture for a problem, it will likely make sense to use it. But if you're working on a new dataset where no such architecture has been put together, you'll often have to create a custom architecture. It's important to be familiar with using both canned and custom architectures, so we've included an example of each in this chapter.

## The MNIST Digit Recognition Dataset

The MNIST digit recognition dataset (see Figure 3-2) requires the construction of a machine learning model that can learn to classify handwritten digits correctly. The challenge is to classify digits from 0 to 9 given 28 × 28-pixel black and white images. The dataset contains 60,000 training examples and a test set of 10,000 examples.

*Figure 3-2. Samples drawn from the MNIST handwritten digit recognition dataset. (Source: GitHub)*

The MNIST dataset is not particularly challenging as far as machine learning problems go. Decades of research have produced state-of-the-art algorithms that achieve



Copyrighted material

close to 100% test set accuracy on this dataset. As a result, the MNIST dataset is no longer suitable for research work, but it is a good tool for pedagogical purposes.

**Isn't DeepChem Just for the Life Sciences?**

As we mentioned earlier in the chapter, it's entirely feasible to use other deep learning packages for life science applications. Similarly, it's possible to build general machine learning systems using DeepChem. Although building a movie recommendation system in DeepChem might be trickier than it would be with more specialized tools, it would be quite feasible to do so. And for good reason: there have been multiple studies looking into the use of recommendation system algorithms for use in molecular binding prediction. Machine learning architectures used in one field tend to carry over to other fields, so it's important to retain the flexibility needed for innovative work.

## A Convolutional Architecture for MNIST

DeepChem uses the `TensorGraph` class to construct nonstandard deep learning architectures. In this section, we will walk through the code required to construct the convolutional architecture shown in Figure 3-3. It begins with two convolutional layers to identify local features within the image. They are followed by two fully connected layers to predict the digit from those local features.

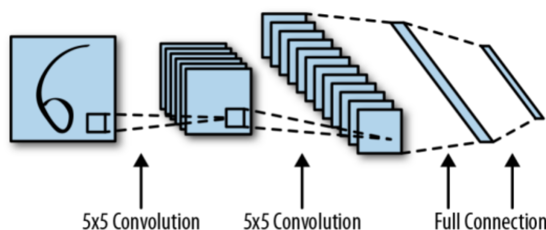



*Figure 3-3. An illustration of the architecture that we will construct in this section for processing the MNIST dataset.*

To begin, execute the following commands to download the raw MNIST data files and store them locally:

```
mkdir MNIST_data
cd MNIST_data
wget http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
wget http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
wget http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
wget http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
cd ..
```



Copyrighted material

Let's now load these datasets:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

We're going to process this raw data into a format suitable for analysis by DeepChem. Let's start with the necessary imports:

```
import deepchem as dc
import tensorflow as tf
import deepchem.models.tensorgraph.layers as layers
```

The submodule `deepchem.models.tensorgraph.layers` contains a collection of "layers." These layers serve as building blocks of deep architectures and can be composed to build new deep learning architectures. We will demonstrate how layer objects are used shortly. Next, we construct `NumpyDataset` objects that wrap the MNIST training and test datasets:

```
train_dataset = dc.data.NumpyDataset(mnist.train.images, mnist.train.labels)
test_dataset = dc.data.NumpyDataset(mnist.test.images, mnist.test.labels)
```

Note that although there wasn't originally a test dataset defined, the `input_data()` function from TensorFlow takes care of separating out a proper test dataset for our use. With the training and test datasets in hand, we can now turn our attention towards defining the architecture for the MNIST convolutional network.

The key concept this is based on is that layer objects can be composed to build new models. As we discussed in the previous chapter, each layer takes input from previous layers and computes an output that can be passed to subsequent layers. At the very start, there are input layers that take in features and labels. At the other end are output layers that return the results of the performed computation. In this example, we will compose a sequence of layers in order to construct an image-processing convolutional network. We start by defining a new`TensorGraph`object:

```
model = dc.models.TensorGraph(model_dir='mnist')
```

The `model_dir` option specifies a directory where the model's parameters should be saved. You can omit this, as we did in the previous example, but then the model will not be saved. As soon as the Python interpreter exits, all your hard work training the model will be thrown out! Specifying a directory allows you to reload the model later and make new predictions with it.

Note that since `TensorGraph` inherits from `Model`, this object is an instance of `dc.models.Model` and supports the same `fit()` and `evaluate()` functions we saw previously:

```
In : isinstance(model, dc.models.Model)
Out: True
```



Copyrighted material

We haven't added anything to `model` yet, so our model isn't likely to be very interesting. Let's start by adding some inputs for features and labels by using the `Feature` and `Label` classes:

```
feature = layers.Feature(shape=(None, 784))
label = layers.Label(shape=(None, 10))
```

MNIST contains images of size 28 × 28. When flattened, these form feature vectors of length 784. The labels have a second dimension of 10 since there are 10 possible digit values, and the vector is one-hot encoded. Note that `None` is used as an input dimension. In systems that build on TensorFlow, the value `None` often encodes the ability for a given layer to accept inputs that have any size in that dimension. Put another way, our object `feature` is capable of accepting inputs of shape `(20, 784)` and `(97, 784)` with equal facility. In this case, the first dimension corresponds to the batch size, so our model will be able to accept batches with any number of samples.

### One-Hot Encoding

The MNIST dataset is categorical. That is, objects belong to one of a finite list of potential categories. In this case, these categories are the digits 0 through 9. How can we feed these categories into a machine learning system? One obvious answer would be to simply feed in a single number that takes values from 0 through 9. However, for a variety of technical reasons, this encoding often doesn't seem to work well. The alternative that people commonly use is to *one-hot encode*. Each label for MNIST is a vector of length 10 in which a single element is set to 1, and all others are set to 0. If the nonzero value is at the 0th index, then the label corresponds to the digit 0. If the nonzero value is at the 9th index, then the label corresponds to the digit 9.

In order to apply convolutional layers to our input, we need to convert our flat feature vectors into matrices of shape `(28, 28)`. To do this, we will use a `Reshape` layer:

```
make_image = layers.Reshape(shape=(None, 28, 28), in_layers=feature)
```

Here again the value `None` indicates that arbitrary batch sizes can be handled. Note that we have a keyword argument `in_layers=feature`. This indicates that the `Reshape` layer takes our previous `Feature` layer, `feature`, as input. Now that we have successfully reshaped the input, we can pass it through to the convolutional layers:

```
conv2d_1 = layers.Conv2D(num_outputs=32, activation_fn=tf.nn.relu,
                         in_layers=make_image)
conv2d_2 = layers.Conv2D(num_outputs=64, activation_fn=tf.nn.relu,
                         in_layers=conv2d_1)
```



Copyrighted material

Here, the Conv2D class applies a 2D convolution to each sample of its input, then passes it through a rectified linear unit (ReLU) activation function. Note how in_layers is used to pass along previous layers as inputs to succeeding layers. We want to end by applying Dense (fully connected) layers to the outputs of the convolutional layer. However, the output of Conv2D layers is 2D, so we will first need to apply a Flatten layer to flatten our input to one dimension (more precisely, the Conv2D layer produces a 2D output *for each sample*, so its output has three dimensions; the Flatten layer collapses this to a single dimension per sample, or two dimensions in total):

```
flatten = layers.Flatten(in_layers=conv2d_2)
dense1 = layers.Dense(out_channels=1024, activation_fn=tf.nn.relu,
                                          in_layers=flatten)
dense2 = layers.Dense(out_channels=10, activation_fn=None, in_layers=dense1)
```

The out_channels argument in a Dense layer specifies the width of the layer. The first layer outputs 1,024 values per sample, but the second layer outputs 10 values, corresponding to our 10 possible digit values. We now want to hook this output up to a loss function, so we can train the output to accurately predict classes. We will use the SoftMaxCrossEntropy loss to perform this form of training:

```
smce = layers.SoftMaxCrossEntropy(in_layers=[label, dense2])
loss = layers.ReduceMean(in_layers=smce)
model.set_loss(loss)
```

Note that the SoftMaxCrossEntropy layer accepts both the labels and the output of the last Dense layer as inputs. It computes the value of the loss function for every sample, so we then need to average over all samples to obtain the final loss. This is done with the ReduceMean layer, which we set as our model's loss function by calling model.set_loss().

## SoftMax and SoftMaxCrossEntropy

You often want a model to output a probability distribution. For MNIST, we want to output the probability that a given sample represents each of the 10 digits. Every output must be positive, and they must sum to 1. An easy way to achieve this is to let the model compute arbitrary numbers, then pass them through the confusingly named *softmax* function:

$$\sigma_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The exponential in the numerator ensures that all values are positive, and the sum in the denominator ensures they add up to 1. If one element of $x$ is much larger than the others,



Copyrighted material

the corresponding output element is very close to 1 and all the other outputs are very close to 0.

`SoftMaxCrossEntropy` first uses a softmax function to convert the outputs to probabilities, then computes the cross entropy of those probabilities with the labels. Remember that the labels are one-hot encoded: 1 for the correct class, 0 for all others. You can think of that as a probability distribution! The loss is minimized when the predicted probability of the correct class is as close to 1 as possible. These two operations (softmax followed by cross entropy) often appear together, and computing them as a single step turns out to be more numerically stable than performing them separately.

For numerical stability, layers like `SoftMaxCrossEntropy` compute in log probabilities. We'll need to transform the output with a `SoftMax` layer to obtain per-class output probabilities. We'll add this output to `model` with `model.add_output()`:

```
output = layers.SoftMax(in_layers=dense2)
model.add_output(output)
```

We can now train the model using the same `fit()` function we called in the previous section:

```
model.fit(train_dataset, nb_epoch=10)
```

Note that this method call might take some time to execute on a standard laptop! If the function is not executing quickly enough, try using `nb_epoch=1`. The results will be worse, but you will be able to complete the rest of this chapter more quickly.

Let's define our metric this time to be accuracy, the fraction of labels that are correctly predicted:

```
metric = dc.metrics.Metric(dc.metrics.accuracy_score)
```

We can then compute the accuracy using the same computation as before:

```
train_scores = model.evaluate(train_dataset, [metric])
test_scores = model.evaluate(test_dataset, [metric])
```

This produces excellent performance: the accuracy is 0.999 on the training set, and 0.991 on the test set. Our model identifies more than 99% of the test set samples correctly.



Copyrighted material

**Try to Get Access to a GPU**

As you saw in this chapter, deep learning code can run pretty slowly! Training a convolutional neural network on a good laptop can take more than an hour to complete. This is because this code depends on a large number of linear algebraic operations on image data. Most CPUs are not well equipped to perform these types of computations.

If possible, try to get access to a modern graphics processing unit. These cards were originally developed for gaming, but are now used for many types of numeric computations. Most modern deep learning workloads will run much faster on GPUs. The examples you'll see in this book will be easier to complete with GPUs as well.

If it's not feasible to get access to a GPU, don't worry. You'll still be able to complete the exercises in this book—they might just take a little longer (you might have to grab a coffee or read a book while you wait for the code to finish running).

## Conclusion

In this chapter, you've learned how to use the DeepChem library to implement some simple machine learning systems. In the remainder of this book, we will continue to use DeepChem as our library of choice, so don't worry if you don't have a strong grasp of the fundamentals of the library yet. There will be plenty more examples coming.

In subsequent chapters, we will begin to introduce the basic concepts needed to do effective machine learning on life science datasets. In the next chapter, we will introduce you to machine learning on molecules.



Copyrighted material

Copyrighted material

CHAPTER 4

# Machine Learning for Molecules

This chapter covers the basics of performing machine learning on molecular data. Before we dive into the chapter, it might help for us to briefly discuss why molecular machine learning can be a fruitful subject of study. Much of modern materials science and chemistry is driven by the need to design new molecules that have desired properties. While significant scientific work has gone into new design strategies, much random search is sometimes still needed to construct interesting molecules. The dream of molecular machine learning is to replace such random experimentation with guided search, where machine-learned predictors can propose which new molecules might have desired properties. Such accurate predictors could enable the creation of radically new materials and chemicals with useful properties.

This dream is compelling, but how can we get started on this path? The first step is to construct technical methods for transforming molecules into vectors of numbers that can then be passed to learning algorithms. Such methods are called *molecular featurizations*. We will cover a number of them in this chapter, and more in the next chapter.Molecules are complex entities, and researchers have developed a host of different techniques for featurizing them. These representations include chemical descriptor vectors, 2D graph representations, 3D electrostatic grid representations, orbital basis function representations, and more.

Once featurized, a molecule still needs to be learned from. We will review some algorithms for learning functions on molecules, including simple fully connected networks as well as more sophisticated techniques like graph convolutions. We'll also describe some of the limitations of graph convolutional techniques, and what we should and should not expect from them. We'll end the chapter with a molecular machine learning case study on an interesting dataset.



Copyrighted material