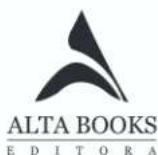


**O'REILLY®**

# Mãos à Obra: Aprendizado de Máquina com Scikit-Learn & TensorFlow

CONCEITOS, FERRAMENTAS E  
TÉCNICAS PARA A CONSTRUÇÃO  
DE SISTEMAS INTELIGENTES



Aurélien Géron

---

# Mãos à Obra: Aprendizado de Máquina com Scikit-Learn & TensorFlow

*Conceitos, Ferramentas e Técnicas para a  
Construção de Sistemas Inteligentes*

*Aurélien Géron*



**Mãos à Obra Aprendizado Máquina com Scikit-Learn e TensorFlow**

Copyright © 2019 da Starlin Alta Editora e Consultoria Eireli. ISBN: 978-85-508-0902-1 (PDF)

*Translated from original Hands-On Machine Learning With Scikit-Learn and TensorFlow© 2017 by Aurélien Géron. All rights reserved. ISBN 978-1-491-96229-9. This translation is published and sold by permission of O'Reilly Media, Inc the owner of all rights to publish and sell the same. PORTUGUESE language edition published by Starlin Alta Editora e Consultoria Eireli, Copyright © 2019 by Starlin Alta Editora e Consultoria Eireli.*

Todos os direitos estão reservados e protegidos por Lei. Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida. A violação dos Direitos Autorais é crime estabelecido na Lei nº 9.610/98 e com punição de acordo com o artigo 184 do Código Penal.

A editora não se responsabiliza pelo conteúdo da obra, formulada exclusivamente pelo(s) autor(es).

**Marcas Registradas:** Todos os termos mencionados e reconhecidos como Marca Registrada e/ou Comercial são de responsabilidade de seus proprietários. A editora informa não estar associada a nenhum produto e/ou fornecedor apresentado no livro.

Edição revisada conforme o Acordo Ortográfico da Língua Portuguesa de 2009.

Publique seu livro com a Alta Books. Para mais informações envie um e-mail para [autoria@altabooks.com.br](mailto:autoria@altabooks.com.br)

Obra disponível para venda corporativa e/ou personalizada. Para mais informações, fale com [projetos@altabooks.com.br](mailto:projetos@altabooks.com.br)

<b>Produção Editorial</b> Editora Alta Books	<b>Produtor Editorial</b> Juliana de Oliveira Thiê Alves	<b>Marketing Editorial</b> <a href="mailto:marketing@altabooks.com.br">marketing@altabooks.com.br</a>	<b>Vendas Atacado e Varejo</b> Danielle Fonseca Viviane Paiva <a href="mailto:comercial@altabooks.com.br">comercial@altabooks.com.br</a>	<b>Ouvidoria</b> <a href="mailto:ouvidoria@altabooks.com.br">ouvidoria@altabooks.com.br</a>
<b>Gerência Editorial</b> Anderson Vieira	<b>Assistente Editorial</b> Adriano Barros	<b>Editor de Aquisição</b> José Rugeri <a href="mailto:j.rugeri@altabooks.com.br">j.rugeri@altabooks.com.br</a>		
<b>Equipe Editorial</b>	Bianca Teodoro Ian Verçosa Illysabelle Trajano	Kelry Oliveira Keyciane Botelho Maria de Lourdes Borges	Paulo Gomes Thales Silva Thauan Gomes	
<b>Tradução</b> Rafael Contatori	<b>Copidesque</b> Amanda Meirinho	<b>Revisão Gramatical</b> Vivian Sbravatti Fernanda Lutfi	<b>Revisão Técnica</b> Gabriel Campos Engenheiro Eletrônico formado pelo Instituto Militar de Engenharia (IME)	<b>Diagramação</b> Daniel Vargas

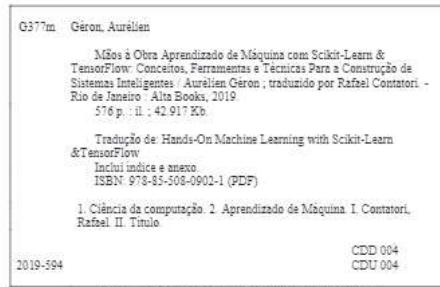
**Erratas e arquivos de apoio:** No site da editora relatamos, com a devida correção, qualquer erro encontrado em nossos livros, bem como disponibilizamos arquivos de apoio se aplicáveis à obra em questão.

Acesse o site [www.altabooks.com.br](http://www.altabooks.com.br) e procure pelo título do livro desejado para ter acesso às erratas, aos arquivos de apoio e/ou a outros conteúdos aplicáveis à obra.

**Suporte Técnico:** A obra é comercializada na forma em que está, sem direito a suporte técnico ou orientação pessoal/exclusiva ao leitor.

A editora não se responsabiliza pela manutenção, atualização e idioma dos sites referidos pelos autores nesta obra.

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBN



Rua Viúva Cláudio, 291 — Bairro Industrial do Jacaré  
CEP: 20.970-031 — Rio de Janeiro (RJ)  
Tels.: (21) 3278-8069 / 3278-8419  
[www.altabooks.com.br](http://www.altabooks.com.br) — [altabooks@altabooks.com.br](mailto:altabooks@altabooks.com.br)  
[www.facebook.com/altabooks](http://www.facebook.com/altabooks) — [www.instagram.com/altabooks](http://www.instagram.com/altabooks)

ASSOCIADO



# Sumário

---

<b>Parte I. Os Fundamentos do Aprendizado de Máquina</b>	<b>1</b>
1. O Cenário do Aprendizado de Máquinas .....	3
O que é o Aprendizado de Máquina?	4
Por que Utilizar o Aprendizado de Máquina?	5
Tipos de Sistemas do Aprendizado de Máquina	8
Aprendizado Supervisionado/Não Supervisionado	8
Aprendizado Supervisionado	8
Aprendizado Não Supervisionado	10
Aprendizado Semi-supervisionado	13
Aprendizado por Reforço	14
Aprendizado Online e Em Lote	15
Aprendizado baseado em Instância Versus	
Aprendizado baseado no Modelo	18
Principais Desafios do Aprendizado de Máquina	23
Quantidade Insuficiente de Dados de Treinamento	23
Dados de Treinamento Não Representativos	25
Dados de Baixa Qualidade	26
Características Irrelevantes	27
Sobreajustando os Dados de Treinamento	27
Subajustando os Dados de Treinamento	29
Voltando Atrás	30
Testando e Validando	30
Exercícios	32

---

<b>2. Projeto de Aprendizado de Máquina de Ponta a Ponta.....</b>	<b>35</b>
Trabalhando com Dados Reais	35
Um Olhar no Quadro Geral	37
Enquadre o Problema	37
Selecione uma Medida de Desempenho	39
Verifique as Hipóteses	42
Obtenha os Dados	42
Crie o Espaço de Trabalho	42
Baixe os Dados	45
Uma Rápida Olhada na Estrutura dos Dados	47
Crie um Conjunto de Testes	51
Descubra e Visualize os Dados para Obter Informações	55
Visualizando Dados Geográficos	55
Buscando Correlações	58
Experimentando com Combinações de Atributo	60
Prepare os Dados para Algoritmos do Aprendizado de Máquina	62
Limpeza dos Dados	62
Manipulando Texto e Atributos Categóricos	65
Customize Transformadores	67
Escalonamento das Características	68
Pipelines de Transformação	69
Selecionar e Treine um Modelo	71
Treinando e Avaliando no Conjunto de Treinamento	72
Avaliando Melhor com a Utilização da Validação Cruzada	73
Ajuste Seu Modelo	76
Grid Search	76
Randomized Search	78
Métodos de Ensemble	78
Analise os Melhores Modelos e Seus Erros	79
Avalie Seu Sistema no Conjunto de Testes	80
Lance, Monitore e Mantenha seu Sistema	80
Experimente!	81
Exercícios	82
<b>3. Classificação .....</b>	<b>83</b>
MNIST	83
Treinando um Classificador Binário	86

Medições de Desempenho	86
Medição da Acurácia com a Utilização da Validação Cruzada	87
Matriz de Confusão	88
Precisão e Revocação	90
Compensação da Precisão/Revocação	91
A Curva ROC	95
Classificação Multiclasse	97
Análise de Erro	100
Classificação Multilabel	104
Classificação Multioutput	105
Exercícios	107
<b>4. Treinando Modelos .....</b>	<b>109</b>
Regressão Linear	110
Método dos Mínimos Quadrados	112
Complexidade Computacional	114
Gradiente Descendente	115
Gradiente Descendente em Lote	118
Gradiente Descendente Estocástico	121
Gradiente Descendente em Minilotes	124
Regressão Polinomial	125
Curvas de Aprendizado	127
Modelos Lineares Regularizados	132
Regressão de Ridge	132
Regressão Lasso	135
Elastic Net	137
Parada Antecipada	138
Regressão Logística	139
Estimando Probabilidades	140
Treinamento e Função de Custo	141
Limites de Decisão	142
Regressão Softmax	144
Exercícios	147
<b>5. Máquinas de Vetores de Suporte (SVM) .....</b>	<b>149</b>
Classificação Linear dos SVM	149
Classificação de Margem Suave	150

Classificação SVM Não Linear	153
Kernel Polinomial	154
Adicionando Características de Similaridade	155
Kernel RBF Gaussiano	156
Complexidade Computacional	158
Regressão SVM	158
Nos Bastidores	160
Função de Decisão e Previsões	161
Objetivo do Treinamento	162
Programação Quadrática	163
O Problema Dual	164
SVM Kernelizado	165
SVM Online	167
Exercícios	169
<b>6. Árvores de Decisão.....</b>	<b>171</b>
Treinando e Visualizando uma Árvore de Decisão	171
Fazendo Previsões	173
Estimando as Probabilidades de Classes	175
O Algoritmo de Treinamento CART	175
Complexidade Computacional	176
Coeficiente de Gini ou Entropia?	177
Hiperparâmetros de Regularização	177
Regressão	179
Instabilidade	181
Exercícios	182
<b>7. Ensemble Learning e Florestas Aleatórias .....</b>	<b>185</b>
Classificadores de Votação	186
Bagging e Pasting	189
Bagging e Pasting no Scikit-Learn	190
Avaliação Out-of-Bag	191
Patches e Subespaços Aleatórios	192
Florestas Aleatórias	193
Árvores-Extras	194
Importância da Característica	194

Boosting	196
AdaBoost	196
Gradient Boosting	199
Stacking	204
Exercícios	207
<b>8. Redução da Dimensionalidade.....</b>	<b>209</b>
A Maldição da Dimensionalidade	210
Principais Abordagens para a Redução da Dimensionalidade	211
Projeção	211
Manifold Learning	213
PCA	215
Preservando a Variância	215
Componentes Principais	216
Projetando para d Dimensões	217
Utilizando o Scikit-Learn	218
Taxa de Variância Explicada	218
Escolhendo o Número Certo de Dimensões	219
PCA para a Compressão	220
PCA Incremental	221
PCA Randomizado	222
Kernel PCA	222
Selecionando um Kernel e Ajustando Hiperparâmetros	223
LLE	225
Outras Técnicas de Redução da Dimensionalidade	227
Exercícios	228
<b>Parte II. Redes Neurais e Aprendizado Profundo</b>	<b>231</b>
<b>9. Em Pleno Funcionamento com o TensorFlow .....</b>	<b>233</b>
Instalação	236
Criando Seu Primeiro Grafo e o Executando em uma Sessão	237
Gerenciando Grafos	238
Ciclo de Vida de um Valor do Nô	239
Regressão Linear com o TensorFlow	240

Implementando o Gradiente Descendente	241
Calculando Manualmente os Gradientes	241
Utilizando o autodiff	242
Utilizando um Otimizador	244
Fornecendo Dados ao Algoritmo de Treinamento	244
Salvando e Restaurando Modelos	245
Visualização do Grafo e Curvas de Treinamento com o TensorBoard	247
Escopos do Nome	250
Modularidade	251
Compartilhando Variáveis	253
Exercícios	256
<b>10. Introdução às Redes Neurais Artificiais .....</b>	<b>259</b>
De Neurônios Biológicos a Neurônios Artificiais	260
Neurônios Biológicos	261
Cálculos Lógicos com Neurônios	262
O Perceptron	263
Perceptron Multicamada e Retropropagação	267
Treinando um MLP com a API de Alto Nível do TensorFlow	270
Treinando um DNN Utilizando um TensorFlow Regular	271
Fase de Construção	271
Fase de Execução	275
Utilizando a Rede Neural	276
Ajustando os Hiperparâmetros da Rede Neural	277
Número de Camadas Ocultas	277
Número de Neurônios por Camada Oculta	278
Funções de Ativação	279
Exercícios	280
<b>11. Treinando Redes Neurais Profundas.....</b>	<b>283</b>
Problemas dos Gradientes: Vanishing/Exploding	283
Inicialização Xavier e Inicialização He	285
Funções de Ativação Não Saturadas	287
Normalização em Lote	290
Implementando a Normalização em Lote com o TensorFlow	292
Gradient Clipping	294

Reutilizando Camadas Pré-Treinadas	295
Reutilizando um Modelo do TensorFlow	296
Reutilizando Modelos de Outras Estruturas	298
Congelando as Camadas Inferiores	298
Armazenamento em Cache das Camadas Congeladas	299
Ajustando, Descartando ou Substituindo as Camadas Superiores	300
Zoológicos de Modelos	301
Pré-treinamento Não Supervisionado	301
Pré-treinamento em uma Tarefa Auxiliar	302
Otimizadores Velozes	303
Otimização Momentum	303
Gradiente Acelerado de Nesterov	305
AdaGrad	306
RMSProp	307
Otimização Adam	308
Evitando o Sobreajuste Por Meio da Regularização	312
Parada Antecipada	313
Regularização $\ell_1$ e $\ell_2$	313
Dropout	314
Regularização Max-Norm	317
Data Augmentation	319
Diretrizes Práticas	320
Exercícios	321
<b>12. Distribuindo o TensorFlow Por Dispositivos e Servidores .....</b>	<b>325</b>
Múltiplos Dispositivos em uma Única Máquina	326
Instalação	326
Gerenciando a RAM do GPU	329
Colocando Operações em Dispositivos	331
Posicionamento Simples	331
Registro dos Posicionamentos	332
Função de Posicionamento Dinâmico	333
Operações e Kernels	333
Posicionamento Suave	334
Execução em Paralelo	334
Dependências de Controle	335
Vários Dispositivos em Vários Servidores	336
Abrindo uma Sessão	338
Os Serviços Master e Worker	338

Fixando Operações em Tarefas	339
Particionando Variáveis em Múltiplos	
Servidores de Parâmetros	340
Compartilhando Estado entre Sessões com a Utilização	
de Contêiner de Recursos	341
Comunicação Assíncrona com a Utilização de Filas	
do TensorFlow	343
Enfileirando Dados	344
Desenfileirando os Dados	344
Filas de Tuplas	345
Fechando uma Fila	346
RandomShuffleQueue	346
PaddingFifoQueue	347
Carregando Dados Diretamente do Grafo	348
Pré-carregue os Dados em uma Variável	348
Lendo os Dados de Treinamento Diretamente do Grafo	349
Leitores Multithreaded Utilizando as Classes	
Coordinator e QueueRunner	352
Outras Funções de Conveniências	354
Paralelizando Redes Neurais em um Cluster do TensorFlow	356
Uma Rede Neural por Dispositivo	356
Replicação em Grafo Versus Replicação Entre Grafos	357
Paralelismo do Modelo	359
Paralelismo de Dados	361
Exercícios	366
<b>13. Redes Neurais Convolucionais (CNN) .....</b>	<b>369</b>
A Arquitetura do CórTEX Visual	370
Camada Convolucional	371
Filtros	373
Empilhando Múltiplos Mapas de Características	374
Implementação do TensorFlow	376
Requisitos de Memória	379
Camada Pooling	380
Arquiteturas CNN	381
LeNet-5	382
AlexNet	384
GoogLeNet	385
ResNet	389
Exercícios	393

<b>14. Redes Neurais Recorrentes (RNN).....</b>	<b>397</b>
Neurônios Recorrentes	398
Células de Memória	400
Sequências de Entrada e Saída	401
RNNs Básicas no TensorFlow	402
Desenrolamento Estático Através do Tempo	403
Desenrolamento Dinâmico Através do Tempo	405
Manipulando Sequências de Entrada de Comprimento Variável	406
Manipulando Sequências de Saída de Comprimento Variável	407
Treinando RNNs	407
Treinando um Classificador de Sequência	408
Treinando para Prever Séries Temporais	410
RNN Criativa	415
RNNs Profundas	415
Distribuindo uma RNN Profunda Através de Múltiplas GPUs	416
Aplicando o Dropout	418
A Dificuldade de Treinar sobre Muitos Intervalos de Tempo	419
Célula LSTM	420
Conexões Peephole	422
Célula GRU	423
Processamento de Linguagem Natural	424
Word Embeddings	424
Uma Rede Codificador-Decodificador para	
Tradução de Máquina	426
Exercícios	429
<b>15. Autoencoders .....</b>	<b>431</b>
Representações Eficientes de Dados	432
Executando o PCA com um Autoencoder Linear Incompleto	433
Autoencoders Empilhados	435
Implementação do TensorFlow	436
Amarrando Pesos	437
Treinando um Autoencoder Por Vez	438
Visualizando as Reconstruções	441
Visualizando as Características	441
Pré-treinamento Não Supervisionado Utilizando	
Autoencoders Empilhados	443

Autoencoders de Remoção de Ruídos	444
Implementando o TensorFlow	445
Autoencoders Esparsos	446
Implementando o TensorFlow	448
Autoencoders Variacionais	449
Gerando Dígitos	452
Outros Autoencoders	453
Exercícios	454
<b>16. Aprendizado por Reforço.....</b>	<b>457</b>
Aprendendo a Otimizar Recompensas	458
Pesquisa de Políticas	459
Introdução ao OpenAI Gym	461
Políticas de Rede Neural	464
Avaliação das Ações: O Problema de Atribuição de Crédito	467
Gradientes de Política	468
Processos de Decisão de Markov	473
Aprendizado de Diferenças Temporais e Q-Learning	477
Políticas de Exploração	479
Q-Learning Aproximado e Deep Q-Learning	480
Aprendendo a Jogar Ms. Pac-Man com a	
Utilização do Algoritmo DQN	482
Exercícios	489
Obrigado!	491
<b>A. Soluções dos Exercícios.....</b>	<b>493</b>
<b>B. Lista de Verificação do Projeto de Aprendizado de Máquina .....</b>	<b>521</b>
<b>C. Problema SVM Dual .....</b>	<b>527</b>
<b>D. Autodiff .....</b>	<b>531</b>
<b>E. Outras Arquiteturas Populares RNA.....</b>	<b>539</b>
<b>Índice .....</b>	<b>549</b>

---

## Prefácio

### O Tsunami do Aprendizado de Máquina

Em 2006, Geoffrey Hinton *et al.* publicou um artigo<sup>1</sup> mostrando como treinar uma rede neural profunda capaz de reconhecer dígitos manuscritos com uma precisão de última geração ( $> 98\%$ ). A técnica foi rotulada como “Aprendizado Profundo” [Deep Learning]. Treinar uma rede neural profunda era considerado uma tarefa impossível na época<sup>2</sup> e, desde os anos 1990, que a maioria dos pesquisadores havia abandonado essa ideia. Este artigo reavivou o interesse da comunidade científica e, em pouco tempo, muitos novos trabalhos demonstraram que o Aprendizado Profundo não só era possível, mas capaz de feitos alucinantes que nenhuma outra técnica do Aprendizado de Máquina (AM) poderia esperar alcançar (com a ajuda do tremendo poder de computação e grandes quantidades de dados). Esse entusiasmo logo se estendeu a muitas outras áreas do Aprendizado de Máquina.

Avançaremos dez anos para ver que o Aprendizado de Máquina conquistou a indústria e, atualmente, está nos holofotes dos produtos de alta tecnologia, classificando os resultados de pesquisas na web, potencializando o reconhecimento de voz do smartphone, recomendando vídeos e superando o campeão mundial no jogo Go. Antes que você perceba, estará dirigindo seu carro.

### O Aprendizado de Máquina em Seus Projetos

Naturalmente você está eufórico com o Aprendizado de Máquina e adoraria participar da festa!

---

1 Disponível em <http://www.cs.toronto.edu/~hinton/> (conteúdo em inglês).

2 As redes neurais convolucionais de aprendizado profundo de Yann Lecun funcionavam bem para o reconhecimento de imagens desde a década de 1990, embora não fossem de propósito geral.

Talvez seu robô caseiro pudesse ganhar um cérebro próprio? Talvez ele pudesse reconhecer rostos ou aprender a andar?

Ou talvez sua companhia tenha toneladas de dados (financeiros, de produção, de sensores de máquinas, logs de usuários, estatísticas de hotline, relatórios de RH, etc.) e, muito provavelmente, você descobriria algumas pepitas de ouro escondidas se soubesse onde procurar:

- Segmentar clientes e encontrar a melhor estratégia de marketing para cada grupo;
- Recomendar produtos para cada cliente com base no que clientes similares compraram;
- Detectar quais transações são susceptíveis de serem fraudulentas;
- Prever a receita do próximo ano;

e mais (<https://www.kaggle.com/wiki/DataScienceUseCases>).<sup>3</sup>

Seja qual for a razão, você decidiu se familiarizar com o Aprendizado de Máquina e implementá-lo em seus projetos. O que é uma grande ideia!

## Objetivo e Abordagem

Este livro pressupõe que você não saiba quase nada sobre Aprendizado de Máquinas. Seu objetivo é fornecer os conceitos, as intuições e as ferramentas necessárias para implementar programas capazes de *aprender com os dados*.

Abordaremos um grande número de técnicas, desde as mais simples às mais comumente utilizadas (como a Regressão Linear) e até algumas das técnicas do Aprendizado Profundo que ganham competições com frequência.

Em vez de implementar nossas próprias versões de cada algoritmo, utilizaremos estruturas Python prontas para produção:

- O Scikit-Learn (<http://scikit-learn.org/>) é muito fácil de usar e implementa muitos algoritmos do AM de maneira eficiente, por isso é uma excelente porta de entrada para o Aprendizado de Máquinas.
- O TensorFlow (<http://tensorflow.org/>) é uma biblioteca complexa que utiliza grafos de fluxo de dados para o cálculo numérico distribuído. Distribuindo os

---

<sup>3</sup> Todo o conteúdo dos websites citados está em inglês. A editora Alta Books não se responsabiliza pela manutenção desse conteúdo.

cálculos entre potencialmente milhares de servidores multi GPU, essa biblioteca torna possível treinar e executar de forma eficiente enormes redes neurais. O TensorFlow foi criado no Google e suporta muitas das aplicações em larga escala do Aprendizado de Máquina. Em novembro de 2015 ele se tornou de código aberto.

O livro favorece uma abordagem prática, desenvolvendo uma compreensão intuitiva de Aprendizado de Máquina por meio de exemplos de trabalho concretos e apenas um pouco de teoria. Embora você possa ler este livro sem pegar seu notebook, é altamente recomendável que você treine com os notebooks Jupyter os exemplos de código disponíveis online em <https://github.com/ageron/handson-ml>.

## Pré-requisitos

Este livro pressupõe que você tenha alguma experiência de programação em Python e que esteja familiarizado com as principais bibliotecas científicas do Python, principalmente NumPy (<http://numpy.org/>), Pandas (<http://pandas.pydata.org/>) e Matplotlib (<http://matplotlib.org/>).

Além disso, você deve ter uma compreensão razoável da matemática em nível superior, (cálculo, álgebra linear, probabilidade e estatística) caso se importe com os bastidores.

Se você ainda não conhece o Python, o <http://learnpython.org/> é um ótimo lugar para começar. O tutorial oficial em [python.org](http://python.org) (<https://docs.python.org/3/tutorial/>) também é muito bom.

Se você nunca usou o Jupyter, o Capítulo 2 o guiará na instalação e no básico: uma ótima ferramenta para ter sempre à mão.

Caso você não esteja familiarizado com as bibliotecas científicas do Python, os notebooks Jupyter incluem alguns tutoriais. Há também um tutorial rápido de matemática para álgebra linear.

## Roteiro

Este livro foi organizado em duas partes.

Parte I, *Os Fundamentos do Aprendizado de Máquina*, cobre os seguintes tópicos:

- O que é Aprendizado de Máquina? Que problemas ele tenta resolver? Quais são as principais categorias e conceitos fundamentais dos sistemas de Aprendizado de Máquina?
- Os principais passos de um típico projeto de Aprendizado de Máquina;

- Aprender ajustando um modelo aos dados;
- Otimizar a função de custo;
- Manipular, limpar e preparar os dados;
- Selecionar e desenvolver recursos;
- Selecionar um modelo com a utilização da validação cruzada e ajustar hiperparâmetros;
- Os principais desafios do Aprendizado de Máquina, especialmente o subajuste e o sobreajuste (a compensação do viés/variância);
- Reduzir a dimensionalidade dos dados de treinamento para combater a maldição da dimensionalidade;
- Os algoritmos de aprendizado mais comuns: Regressão Linear e Polinomial, Regressão Logística, *k-Nearest Neighbors*, Máquinas de Vetores de Suporte [SVM], Árvores de Decisão, Florestas Aleatórias e Métodos de *Ensemble*.

Parte II, *Redes Neurais e Aprendizado Profundo*, cobre os seguintes tópicos:

- O que são redes neurais? Servem para quê?
- Construir e treinar redes neurais com a utilização do TensorFlow;
- As mais importantes arquiteturas de redes neurais: *feedforward*, convolucionais, recorrentes, LSTM (*long short-term memory*) e *autoencoders*;
- Técnicas para treinamento de redes neurais profundas;
- Escalonar redes neurais para grandes conjuntos de dados;
- Aprender por reforço.

A primeira parte é baseada principalmente no Scikit-Learn, enquanto a segunda utiliza o TensorFlow.



Não mergulhe de cabeça logo de primeira: embora o Aprendizado Profundo seja, sem dúvida, uma das áreas mais interessantes do Aprendizado de Máquina, primeiro você deve dominar os fundamentos. Além disso, a maioria dos problemas pode ser resolvida com técnicas mais simples, como os métodos Florestas Aleatórias e *Ensemble* (discutidos na Parte I). O Aprendizado Profundo é mais adequado para problemas complexos, como reconhecimento de imagem e de voz, ou processamento de linguagem natural, desde que você tenha dados, poder de computação e paciência suficientes.

## Outros Recursos

Para saber mais sobre o Aprendizado de Máquina existem muitos recursos disponíveis. O curso de AM de Andrew Ng no Coursera (<https://www.coursera.org/learn/machine-learning/>) e o curso de Geoffrey Hinton sobre redes neurais e Aprendizado Profundo (<https://www.coursera.org/course/neuralnets>) são incríveis, embora ambos demandem um investimento significativo de tempo (estamos falando de meses).

Há também muitos sites interessantes sobre o Aprendizado de Máquina, incluindo, claro, o excepcional Guia do Usuário do Scikit-Learn ([http://scikit-learn.org/stable/user\\_guide.html](http://scikit-learn.org/stable/user_guide.html)). Você também pode desfrutar do Dataquest (<https://www.dataquest.io/>), que oferece tutoriais interativos muito interessantes e blogs de AM, como aqueles listados no Quora (<http://goo.gl/GwtU3A>). Finalmente, o site do Aprendizado Profundo (<http://deeplearning.net/>) tem uma boa lista de recursos para seguirmos aprendendo mais.

Há também muitos outros livros introdutórios sobre o Aprendizado de Máquina:

- *Data Science do Zero*, de Joel Grus (Alta Books). Apresenta os fundamentos do Aprendizado de Máquina e implementa alguns dos principais algoritmos em Python puro (do zero, como o nome sugere);
- *Machine Learning: An Algorithmic Perspective*, de Stephen Marsland (Chapman and Hall). É uma ótima introdução ao Aprendizado de Máquina, cobrindo uma ampla gama de tópicos em profundidade, com exemplos de código em Python (também a partir do zero, mas utilizando o NumPy);
- *Python Machine Learning*, de Sebastian Raschka (Packt Publishing). É também uma ótima introdução ao Aprendizado de Máquina. Aproveita as bibliotecas de código aberto do Python (Pylearn 2 e Theano);
- *Learning from Data*, de Yaser S. Abu-Mostafa, Malik Magdon-Ismail e Hsuan-Tien Lin (MLBook). Uma abordagem bastante teórica para o AM, este livro fornece insights profundos, principalmente sobre a compensação do viés/variância (ver Capítulo 4);
- *Inteligência Artificial, 3ª Edição*, de Stuart Russell e Peter Norvig (Campus). É um ótimo (e enorme) livro que aborda uma quantidade incrível de tópicos, incluindo o Aprendizado de Máquina. Ajudando a colocar o AM em perspectiva.

Finalmente, uma ótima maneira de aprender é entrar em sites de competição como o Kaggle.com, que lhe permite praticar suas habilidades com problemas do mundo real com a ajuda e insights de alguns dos melhores profissionais de AM que existem por aí.

## Convenções Utilizadas neste Livro

As seguintes convenções tipográficas são usadas neste livro:

### *Itálicos*

Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e extensões de arquivos.

### **Fonte monoespaciada**

Usada para listagens de programas, bem como em parágrafos sobre elementos de programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, declarações e palavras-chave.

### **Fonte monoespaciada em negrito**

Mostra comandos ou outro texto que deve ser literalmente digitado pelo usuário.

### *Fonte monoespaciada em itálico*

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este significa uma nota geral.



Este indica alerta ou cautela.

## Utilizando Exemplos de Código

O material suplementar (exemplos de códigos, exercícios, etc.) está disponível para download em <https://github.com/ageron/handson-ml> e também no site da editora Alta Books. Entre em [www.altabooks.com.br](http://www.altabooks.com.br) e procure pelo nome do livro.

Este livro está aqui para ajudá-lo a fazer o que precisa ser feito. Em geral, se um código de exemplo for oferecido, você poderá utilizá-lo em seus programas e documentações. Não é necessário entrar em contato conosco para obter permissão de uso, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que utiliza vários trechos de código deste livro não requer permissão. Vender ou distribuir um CD-ROM com exemplos dos livros O'Reilly exigirá permissão. Responder uma pergunta citando este livro e citando um exemplo de código não requer permissão. Mas, a incorporação de uma quantidade significativa de exemplos de código deste livro na documentação do seu produto requer sim permissão.

Agradecemos, mas não exigimos atribuições. Uma atribuição geralmente inclui o título, o autor, o editor e o ISBN. Por exemplo: “*Mãos à Obra: Aprendizado de Máquina com Scikit-Learn & TensorFlow* por Aurélien Géron (AltaBooks). Copyright 2019 de Aurélien Géron, 978-85-508-0381-4.”

## Agradecimentos

Gostaria de agradecer aos meus colegas do Google, em especial à equipe de classificação de vídeos do YouTube, por me ensinarem muito sobre o Aprendizado de Máquina. Nunca teria iniciado este projeto sem eles. Agradecimentos especiais aos meus gurus pessoais do AM: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington e todos do YouTube de Paris.

Sou incrivelmente grato a todas as pessoas maravilhosas que tiraram um tempo de suas vidas ocupadas para revisar meu livro com tantos detalhes. Agradeço a Pete Warden por responder todas as minhas perguntas sobre o TensorFlow, revisar a Parte II, fornecer muitos insights interessantes e, claro, por fazer parte da equipe principal do TensorFlow. Aconselho você a definitivamente acompanhar o blog dele (<https://petewarden.com/>)! Muito obrigado a Lukas Biewald por sua revisão completa da Parte II: ele não deixou pedra sobre pedra, testou todo o código (e pegou alguns erros), fez ótimas sugestões, e seu entusiasmo foi contagiante. Acompanhe o blog dele (<https://lukasbiewald.com/>) e seus incríveis robôs (<https://goo.gl/Eu5u28>)! Agradecimentos a Justin Francis, que também analisou muito detalhadamente a Parte II, pegando erros e fornecendo importantes insights, principalmente no Capítulo 16. Confira suas postagens (<https://goo.gl/28ve8z>) sobre o TensorFlow!

Muito obrigado também a David Andrzejewski, que revisou a Parte I identificando seções pouco claras e sugerindo como melhorá-las e forneceu um feedback incrivelmente útil. Confira o site dele (<http://www.david-andrzejewski.com/>)! Agradecimentos a Grégoire Mesnil, que revisou a Parte II e contribuiu com conselhos práticos muito interessantes sobre o treinamento de redes neurais. Agradeço também a Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn, Iain Smears e Vincent Guilbeau por revisarem a Parte I e por fazerem muitas sugestões úteis. Também gostaria de agradecer ao meu sogro, Michel Tessier, ex-professor de matemática e agora um grande tradutor de Anton Tchekhov, por me ajudar a resolver algumas das matemáticas e anotações deste livro e rever o notebook do Jupyter de álgebra linear.

E, claro, um “obrigado” gigantesco ao meu querido irmão Sylvain, que revisou cada capítulo, testou todas as linhas de código, forneceu feedback sobre praticamente todas as seções e me encorajou desde a primeira até a última linha. Amo você, irmãozinho!

Muito obrigado também à fantástica equipe da O'Reilly, principalmente Nicole Tache, que me deu um feedback perspicaz, e foi sempre alegre, encorajadora e prestativa. Agradeço também a Marie Beaugureau, Ben Lorica, Mike Loukides e Laurel Ruma por acreditarem neste projeto e me ajudarem a definir seu escopo. Obrigado a Matt Hacker e a toda a equipe da Atlas por responder todas as minhas questões técnicas sobre formatação, *asciidoc* e *LaTeX*, e a Rachel Monaghan, Nick Adams e toda a equipe de produção pela revisão final e centenas de correções.

Por último, mas não menos importante, sou infinitamente grato a minha amada esposa, Emmanuelle, e aos nossos três maravilhosos filhos, Alexandre, Rémi e Gabrielle, por me encorajarem a trabalhar muito neste livro, fazerem muitas perguntas (quem disse que não é possível ensinar redes neurais para uma criança de sete anos?) e até trazerem biscoitos e café. O que mais se pode sonhar?

## Aviso

Para melhor entendimento as figuras coloridas estão disponíveis no site da editora Alta Books. Acesse: [www.altabooks.com.br](http://www.altabooks.com.br) e procure pelo nome do livro ou ISBN.

## Parte I

# Os Fundamentos do Aprendizado de Máquina



## Capítulo 1

# O Cenário do Aprendizado de Máquina

A maioria das pessoas pensa em um robô quando ouve “Aprendizado de Máquina”: um mordomo confiável ou um Exterminador do Futuro, dependendo para quem você perguntar. Mas o Aprendizado de Máquina não é apenas uma fantasia futurista; ele já está entre nós. Na verdade, há algumas décadas o AM foi introduzido como *Reconhecimento Ótico de Caracteres* (OCR) em algumas publicações especializadas, mas o primeiro aplicativo que realmente ficou famoso e conquistou o mundo na década de 1990, melhorando a vida de centenas de milhões de pessoas, foi o *filtro de spam*. Não era exatamente um robô autoconsciente da Skynet, mas pode ser tecnicamente qualificado como Aprendizado de Máquina (uma máquina que aprendeu tão bem que raramente é necessário marcar um e-mail como spam). O filtro de spam foi seguido por centenas de aplicativos AM que agora, silenciosamente, fornecem centenas de produtos e recursos, de recomendações a buscas por voz, que você utiliza regularmente.

Onde começa e onde termina o Aprendizado de Máquina? O que significa exatamente que uma máquina *aprende* alguma coisa? Se eu baixar uma cópia da Wikipédia, meu computador realmente “aprenderá” algo? Será que de repente ele fica mais esperto? Neste capítulo, começaremos esclarecendo o que é o Aprendizado de Máquina e por que você vai querer utilizá-lo.

Então, antes de iniciarmos a exploração do mundo do Aprendizado de Máquina, analisaremos seu mapa e conheceremos suas principais regiões e os cenários mais notáveis: aprendizado supervisionado versus não supervisionado, aprendizado online versus aprendizado em lote, aprendizado baseado em instâncias versus aprendizado baseado em modelo. Em seguida, analisaremos o fluxo de trabalho de um típico projeto de AM, discutiremos os principais desafios que você poderá enfrentar e mostraremos como avaliar e ajustar um sistema de Aprendizado de Máquina.

Este capítulo apresenta muitos conceitos fundamentais (e jargões) que todo cientista de dados deve saber de cor. Será uma visão geral de alto nível (o único capítulo sem muito

código), tudo bastante simples, mas você deve se certificar de que entendeu tudo antes de continuar. Então pegue um café e mãos à obra!



Se você já conhece todos os conceitos básicos do Aprendizado de Máquina, pule diretamente para o Capítulo 2. Se não tiver certeza, tente responder a todas as perguntas listadas no final do capítulo antes de seguir em frente.

## O que É o Aprendizado de Máquina?

Aprendizado de Máquina é a ciência (e a arte) da programação de computadores para que eles possam *aprender com os dados*.

Veja uma definição um pouco mais abrangente:

[Aprendizado de Máquina é o] campo de estudo que dá aos computadores a habilidade de aprender sem ser explicitamente programado.

—Arthur Samuel, 1959

E uma mais direcionada aos engenheiros:

Diz-se que um programa de computador aprende pela experiência E em relação a algum tipo de tarefa T e alguma medida de desempenho P se o seu desempenho em T, conforme medido por P, melhora com a experiência E.

—Tom Mitchell, 1997

Por exemplo, seu filtro de spam é um programa de Aprendizado de Máquina que pode aprender a assinalar e-mails como spam (por exemplo, os marcados pelos usuários) e como regulares (não spam, também chamados de “ham”). Os exemplos utilizados pelo sistema para o aprendizado são chamados de *conjuntos de treinamentos*. Cada exemplo de treinamento é chamado de *instância de treinamento* (ou *amostra*). Nesse caso, assinalar novos e-mails como spam é a tarefa T, a experiência E é o *dado de treinamento* e a medida de desempenho P precisa ser definida; por exemplo, você pode utilizar a média de e-mails classificados corretamente. Esta medida de desempenho particular é chamada de *acurácia* e é utilizada frequentemente em tarefas de classificação.

Se você baixou uma cópia da Wikipédia, seu computador terá muito mais dados, mas não terá repentinamente um melhor desempenho em nenhuma tarefa. Portanto, não é Aprendizado de Máquina.

## Por que Utilizar o Aprendizado de Máquina?

Considere como você escreveria um filtro de spam utilizando técnicas de programação tradicionais (Figura 1-1):

1. Primeiro, identificaria as características do spam. Você nota que algumas palavras ou frases (termos em inglês como “4U”, “credit card”, “free” e “amazing”) tendem a aparecer muito no campo do assunto. Talvez você note outros padrões no nome do remetente, no corpo do e-mail, e assim por diante.
2. Segundo, escreveria um algoritmo de detecção para cada um dos padrões observados, e, se fossem detectados, seu programa marcaria esses e-mails como spam.
3. Por último, você testaria seu programa, e repetiria os passos 1 e 2 até que esteja satisfatório.

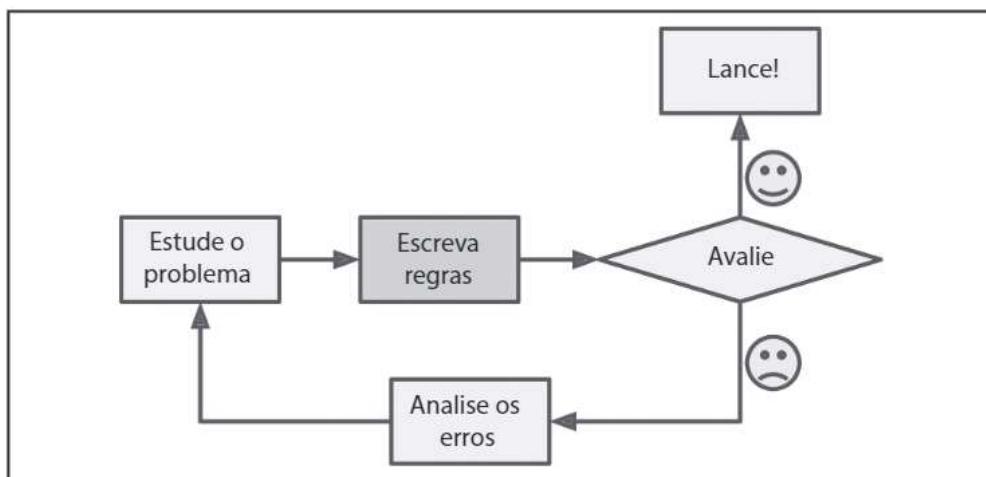


Figura 1-1. A abordagem tradicional

Como o problema não é trivial, seu programa provavelmente se tornará uma longa lista de regras complexas — com uma manutenção muito difícil.

Em contrapartida, um filtro de spam baseado em técnicas de Aprendizado de Máquina aprende automaticamente quais palavras e frases são bons indicadores de spam, detectando padrões de palavras estranhamente frequentes em exemplos de spam se comparados aos exemplos dos e-mails “não spam” (Figura 1-2). O programa é muito menor, de mais fácil manutenção e, provavelmente, mais preciso.

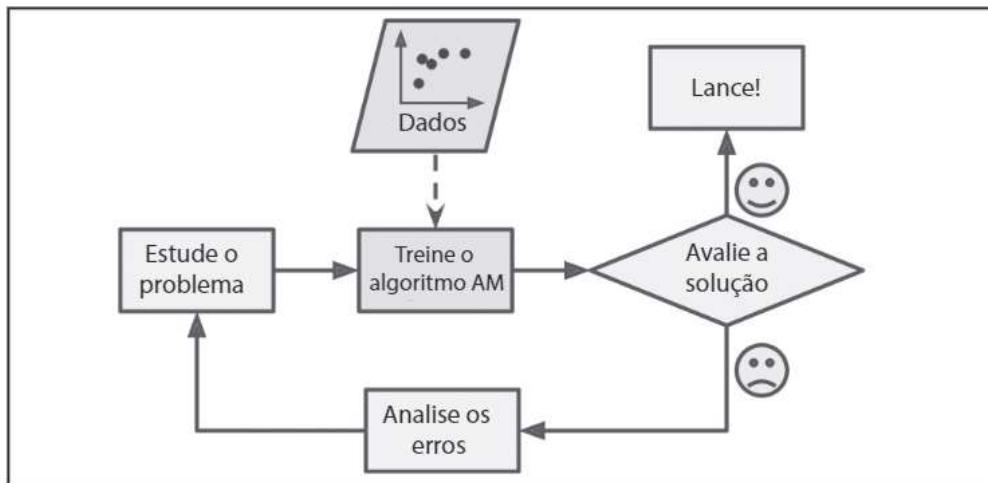


Figura 1-2. Abordagem do Aprendizado de Máquina

Além disso, se os *spammers* perceberem que todos os seus e-mails contendo “4U” são bloqueados, poderão começar a escrever “For U”. Um filtro de *spam* que utiliza técnicas de programação tradicionais precisaria ser atualizado para marcar os e-mails “For U”. Se os *spammers* continuam contornando seu filtro de *spam*, será preciso escrever novas regras para sempre.

Em contrapartida, um filtro de *spam* baseado em técnicas de Aprendizado de Máquina percebe automaticamente que “For U” tornou-se frequente no *spam* marcado pelos usuários e começa a marcá-los sem a sua intervenção (Figura 1-3).

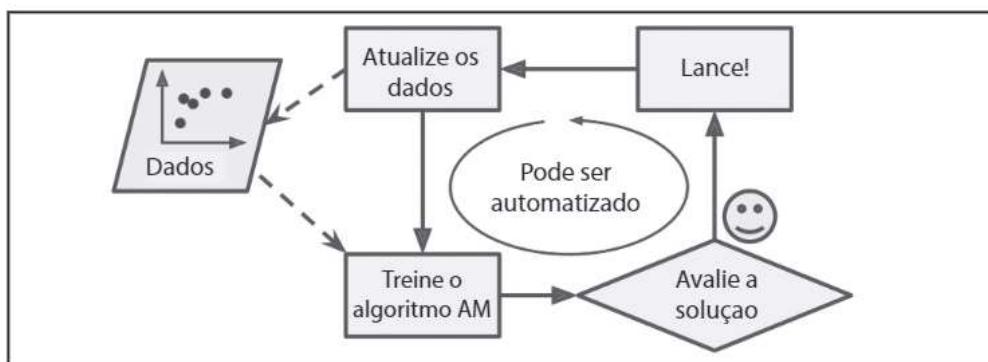


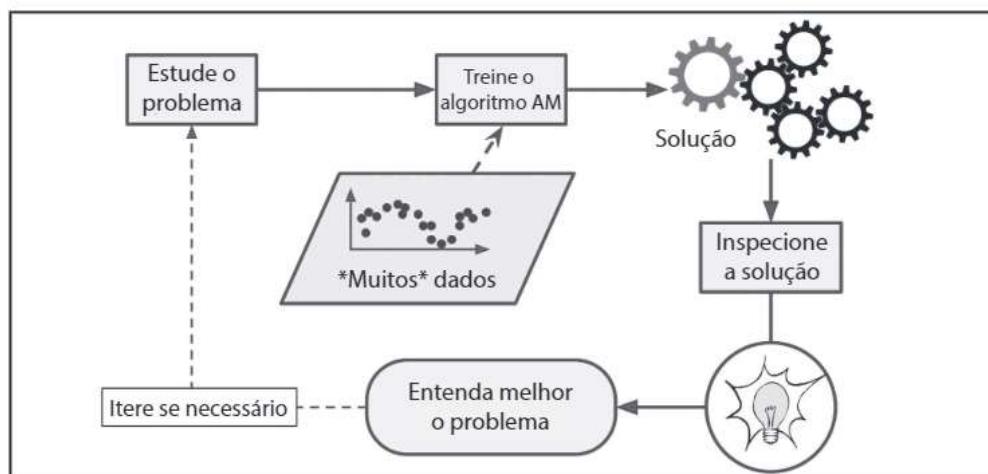
Figura 1-3. Adaptando-se automaticamente à mudança

Outra área na qual o Aprendizado de Máquina se destaca é nos problemas muito complexos para abordagens tradicionais ou que não possuem um algoritmo conhecido. Por exemplo, considere o reconhecimento da fala: digamos que você deseja começar com o básico e escreve um programa capaz de distinguir as palavras “one” e “two”. Você deve

perceber que a palavra “Two” começa com um som agudo (“T”), então você codifica um algoritmo que mede a intensidade do som agudo e o utiliza para distinguir “One” e “Two”. Obviamente, esta técnica não se estenderá a milhares de palavras faladas por milhões de pessoas muito diferentes em ambientes confusos e em dezenas de idiomas. A melhor solução (pelo menos hoje) seria escrever um algoritmo que aprenda por si só por meio de muitas gravações de exemplos para cada palavra.

Finalmente, o Aprendizado de Máquina pode ajudar os seres humanos a aprender (Figura 1-4): os algoritmos do AM podem ser inspecionados para que vejamos o que eles aprenderam (embora para alguns algoritmos isso possa ser complicado). Por exemplo, uma vez que o filtro foi treinado para o spam, ele pode facilmente ser inspecionado e revelar uma lista de palavras e combinações previstas que ele acredita serem as mais prováveis. Às vezes, isso revelará correlações não esperadas ou novas tendências, levando a uma melhor compreensão do problema.

Aplicar técnicas do AM para se aprofundar em grandes quantidades de dados pode ajudar na descoberta de padrões que não eram aparentes. Isto é chamado de mineração de dados.



*Figura 1-4. O Aprendizado de Máquina pode ajudar no ensino dos humanos*

Resumindo, o Aprendizado de Máquina é ótimo para:

- Problemas para os quais as soluções existentes exigem muita configuração manual ou longas listas de regras: um algoritmo de Aprendizado de Máquina geralmente simplifica e melhora o código;
- Problemas complexos para os quais não existe uma boa solução quando utilizamos uma abordagem tradicional: as melhores técnicas de Aprendizado de Máquina podem encontrar uma solução;

- Ambientes flutuantes: um sistema de Aprendizado de Máquina pode se adaptar a novos dados;
- Compreensão de problemas complexos e grandes quantidades de dados.

## Tipos de Sistemas do Aprendizado de Máquina

Existem tantos tipos diferentes de sistemas de Aprendizado de Máquina que será útil classificá-los em extensas categorias com base em:

- Serem ou não treinados com supervisão humana (supervisionado, não supervisionado, semissupervisionado e aprendizado por reforço);
- Se podem ou não aprender rapidamente, de forma incremental (aprendizado online versus aprendizado por lotes);
- Se funcionam simplesmente comparando novos pontos de dados com pontos de dados conhecidos, ou se detectam padrões em dados de treinamento e criam um modelo preditivo, como os cientistas (aprendizado baseado em instâncias versus aprendizado baseado em modelo).

Esses critérios não são exclusivos; você pode combiná-los da maneira que quiser. Por exemplo, um filtro de spam de última geração pode aprender rapidamente com a utilização de um modelo de rede neural profundo treinado com exemplos de spam e não spam, fazendo deste um sistema de aprendizado supervisionado online, baseado em modelos.

Vejamos cada um desses critérios um pouco mais de perto.

## Aprendizado Supervisionado/Não Supervisionado

Os sistemas de Aprendizado de Máquina podem ser classificados de acordo com a quantidade e o tipo de supervisão que recebem durante o treinamento. Existem quatro categorias principais de aprendizado: supervisionado, não supervisionado, semissupervisionado e por reforço.

### Aprendizado Supervisionado

No *aprendizado supervisionado*, os dados de treinamento que você fornece ao algoritmo incluem as soluções desejadas, chamadas de *rótulos* (Figura 1-5).



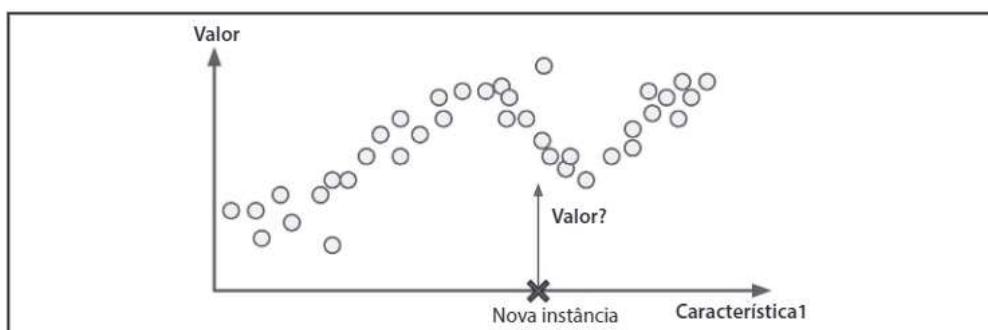
*Figura 1-5. Um conjunto de treinamento rotulado para aprendizado supervisionado (por exemplo, classificação de spam)*

A *classificação* é uma tarefa típica do aprendizado supervisionado. O filtro de spam é um bom exemplo disso: ele é treinado com muitos exemplos de e-mails junto às *classes* (spam ou não spam) e deve aprender a classificar novos e-mails.

Prever um *alvo* de valor numérico é outra tarefa típica, como o preço de um carro a partir de um conjunto de *características* (quilometragem, idade, marca, etc.) denominadas *previsores*. Esse tipo de tarefa é chamada de *regressão* (Figura 1-6)<sup>1</sup>. Para treinar o sistema, você precisa fornecer muitos exemplos de carros incluindo seus previsores e seus *labels* (ou seja, seus preços).



No Aprendizado de Máquina, um *atributo* é um tipo de dado (por exemplo, “Quilometragem”), enquanto uma *característica* possui vários significados, dependendo do contexto, geralmente significando um atributo mais o seu valor (por exemplo, “Quilometragem = 15000”). Embora muitas pessoas utilizem as palavras *atributo* e *característica* intercambiavelmente.



*Figura 1-6. Regressão*

<sup>1</sup> Curiosidade: este nome estranho é um termo de estatística introduzido por Francis Galton, enquanto estudava o fato de que os filhos de pessoas altas tendem a ser mais baixos do que os pais. Como as crianças eram mais baixas, ele chamou essa alteração de *regressão à média*. Este nome foi aplicado aos métodos utilizados por ele para analisar as correlações entre variáveis.

Observe que alguns algoritmos de regressão também podem ser utilizados para classificação, e vice-versa. Por exemplo, a *Regressão Logística* é comumente utilizada para classificação pois pode produzir um valor que corresponde à probabilidade de pertencer a uma determinada classe (por exemplo, 20% de chances de ser spam).

Estes são alguns dos algoritmos mais importantes do aprendizado supervisionado (abordados neste livro):

- k-Nearest Neighbours
- Regressão Linear
- Regressão Logística
- Máquinas de Vetores de Suporte (SVM)
- Árvores de Decisão e Florestas Aleatórias
- Redes Neurais<sup>2</sup>

### Aprendizado Não Supervisionado

No *aprendizado não supervisionado*, como você pode imaginar, os dados de treinamento não são rotulados (Figura 1-7). O sistema tenta aprender sem um professor.

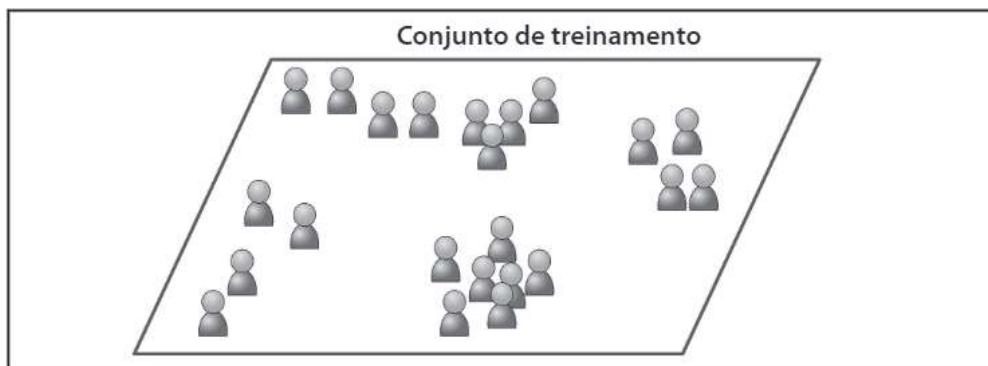


Figura 1-7. Conjunto de treinamento não rotulado para aprendizado não supervisionado

Eis alguns dos mais importantes algoritmos de aprendizado não supervisionado (falarímos sobre a redução da dimensionalidade no Capítulo 8):

<sup>2</sup> Algumas arquiteturas de redes neurais podem ser não supervisionadas, como *autoencoders* e máquinas de Boltzmann restritas. Elas também podem ser semi-supervisionadas, como redes de crenças profundas e pré-treino sem supervisão.

- Clustering
  - k-Means
  - Clustering Hierárquico [HCA, do inglês]
  - Maximização da Expectativa
- Visualização e redução da dimensionalidade
  - Análise de Componentes Principais [PCA, do inglês]
  - Kernel PCA
  - Locally-Linear Embedding (LLE)
  - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Aprendizado da regra da associação
  - Apriori
  - Eclat

Por exemplo, digamos que você tenha muitos dados sobre os visitantes do seu blog. Você quer executar um algoritmo de *clustering* para tentar detectar grupos de visitantes semelhantes (Figura 1-8). Em nenhum momento você diz ao algoritmo a qual grupo o visitante pertence: ele encontrará essas conexões sem sua ajuda. Por exemplo, ele pode notar que 40% dos seus visitantes são homens que adoram histórias em quadrinhos e geralmente leem seu blog à noite, enquanto 20% são jovens amantes de ficção científica e o visitam durante os finais de semana, e assim por diante. Se você utilizar um algoritmo de *clustering hierárquico*, ele também poderá subdividir cada grupo em grupos menores. O que pode ajudá-lo a segmentar suas postagens para cada um deles.

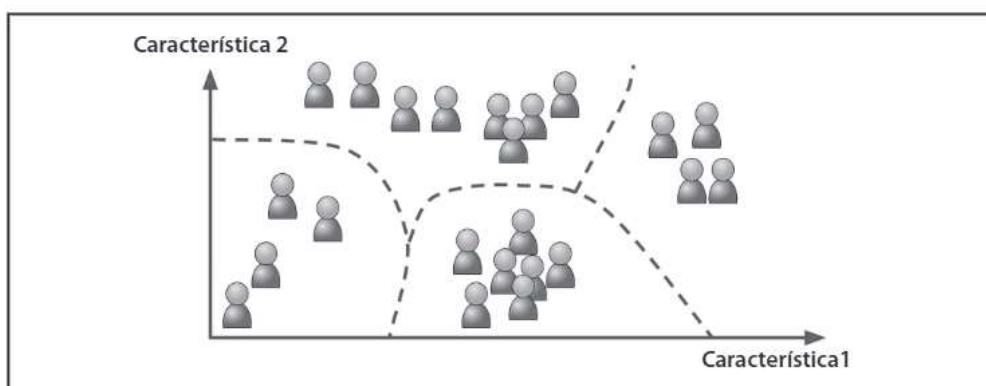


Figura 1-8. Clustering

Os algoritmos de *visualização* também são bons exemplos de algoritmos de aprendizado não supervisionado: você os alimenta com muitos dados complexos e não rotulados e eles

exibem uma representação 2D ou 3D de seus dados que podem ser facilmente plotados (Figura 1-9). Esses algoritmos tentam preservar o máximo da estrutura (por exemplo, tentam fazer com que os clusters no espaço de entrada que estão separados não se sobreponham na visualização) para que você possa entender como os dados estão organizados e talvez identificar padrões ignorados.

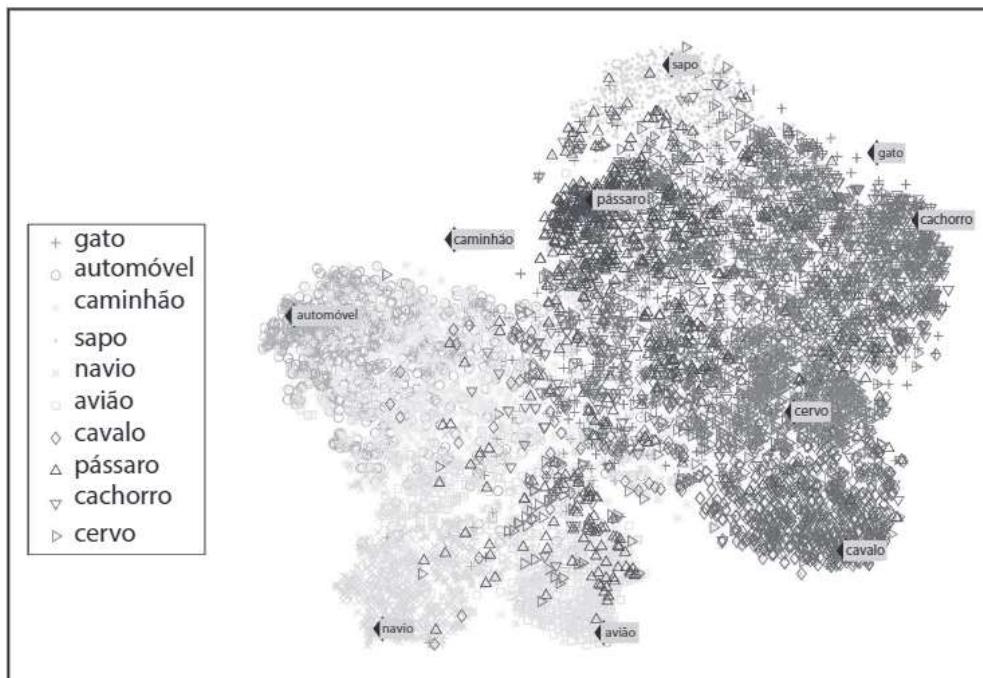


Figura 1-9. Exemplo de uma visualização t-SNE destacando grupos semânticos<sup>3</sup>

A *redução da dimensionalidade* é uma tarefa relacionada na qual o objetivo é simplificar os dados sem perder muita informação. Uma maneira de fazer isso é mesclar várias características correlacionadas em uma. Por exemplo, a quilometragem de um carro pode estar muito correlacionada com seu tempo de uso, de modo que o algoritmo da redução de dimensionalidade irá mesclá-los em uma característica que representa o desgaste do carro. Isso é chamado de *extração de características*.

---

<sup>3</sup> Observe como os animais estão muito bem separados dos veículos, como os cavalos estão próximos aos cervos, mas longe dos pássaros, e assim por diante. Imagem reproduzida com permissão de Socher, Ganjoo, Manning e Ng (2013), “Visualização T-SNE do espaço da palavra semântica”.



É uma boa ideia tentar reduzir a dimensão dos dados de treinamento com a utilização de um algoritmo de redução de dimensionalidade antes de fornecê-lo a outro algoritmo do Aprendizado de Máquina (como um algoritmo de aprendizado supervisionado). Este algoritmo será executado muito mais rapidamente, os dados ocuparão menos espaço em disco e na memória e, em alguns casos, podem rodar melhor também.

Outra importante tarefa não supervisionada é a *detecção de anomalias* — por exemplo, a detecção de transações incomuns em cartões de crédito para evitar fraudes, detectar defeitos de fabricação ou remover automaticamente outliers de um conjunto de dados antes de fornecê-lo a outro algoritmo de aprendizado. O sistema é treinado com instâncias normais e, quando vê uma nova instância, pode dizer se ela parece normal ou se é uma provável anomalia (veja a Figura 1-10).

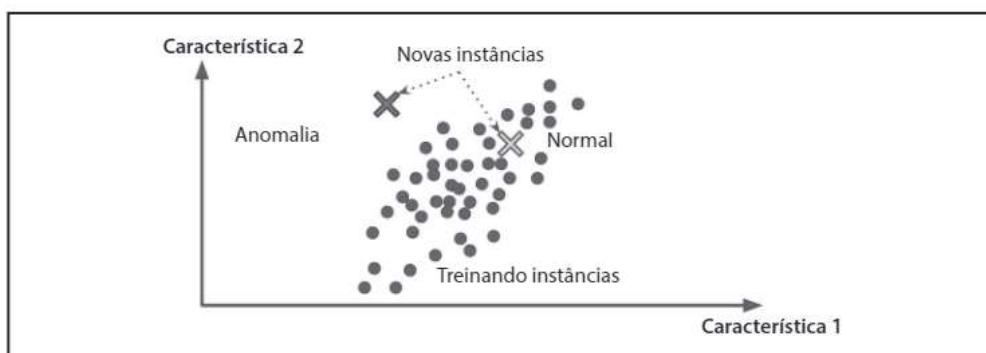


Figura 1-10. Detecção da anomalia

Finalmente, outra tarefa comum não supervisionada é o *aprendizado de regras de associação*, cujo objetivo é se aprofundar em grandes quantidades de dados e descobrir relações interessantes entre atributos. Por exemplo, suponha que você possua um supermercado. Executar uma regra de associação em seus registros de vendas pode revelar que as pessoas que compram molho de churrasco e batatas fritas também tendem a comprar carnes. Desta forma, você vai querer colocar esses itens próximos uns dos outros.

### Aprendizado Semi-supervisionado

Alguns algoritmos podem lidar com dados de treinamento parcialmente rotulados, uma grande quantidade de dados não rotulados e um pouco de dados rotulados. Isso é chamado de *aprendizado semi-supervisionado* (Figura 1-11).

Alguns serviços de hospedagem de fotos, como o Google Fotos, são bons exemplos disso. Ao carregar todas as suas fotos de família, o aplicativo reconhecerá automaticamente que a mesma pessoa (A) aparece nas fotos 1, 5 e 11 enquanto outra pessoa (B) aparece nas fotos 2, 5 e 7. Esta é a parte não supervisionada do algoritmo (agrupamento). Agora, o sistema apenas precisa que você diga quem são essas pessoas. Com apenas um *rótulo* por pessoa<sup>4</sup> ele será capaz de nomear todos, o que é útil para pesquisar fotos.

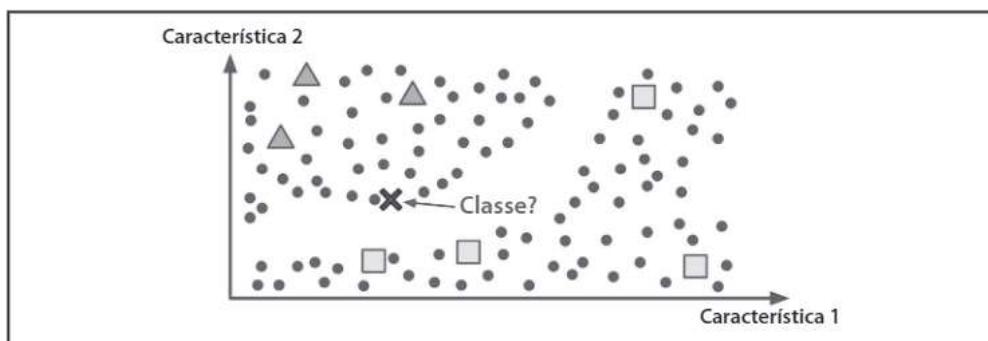


Figura 1-11. Aprendizado Semissupervisionado

A maior parte dos algoritmos de aprendizado semissupervisionado é de combinações de algoritmos supervisionados e não supervisionados. Por exemplo, as *redes neurais de crenças profundas* [DBNs, do inglês] são baseadas em componentes não supervisionados, chamados *máquinas restritas de Boltzmann* [RBMs, do inglês], empilhados uns em cima dos outros. As RBMs são treinadas sequencialmente de forma não supervisionada, e então todo o sistema é ajustado utilizando-se técnicas de aprendizado supervisionado.

### Aprendizado por Reforço

O *aprendizado por reforço* é um bicho muito diferente. O sistema de aprendizado, chamado de *agente* nesse contexto, pode observar o ambiente, selecionar e executar ações e obter *recompensas* em troca — ou *penalidades* na forma de recompensas negativas (Figura 1-12). Ele deve aprender por si só qual é a melhor estratégia, chamada de *política*, para obter o maior número de recompensas ao longo do tempo. Uma política define qual ação o agente deve escolher quando está em determinada situação.

---

<sup>4</sup> Isso quando o sistema funciona perfeitamente. Na prática, muitas vezes ele cria alguns agrupamentos por pessoa e, às vezes, mistura duas pessoas parecidas, portanto, é necessário fornecer alguns rótulos por pessoa e limpar manualmente alguns clusters.

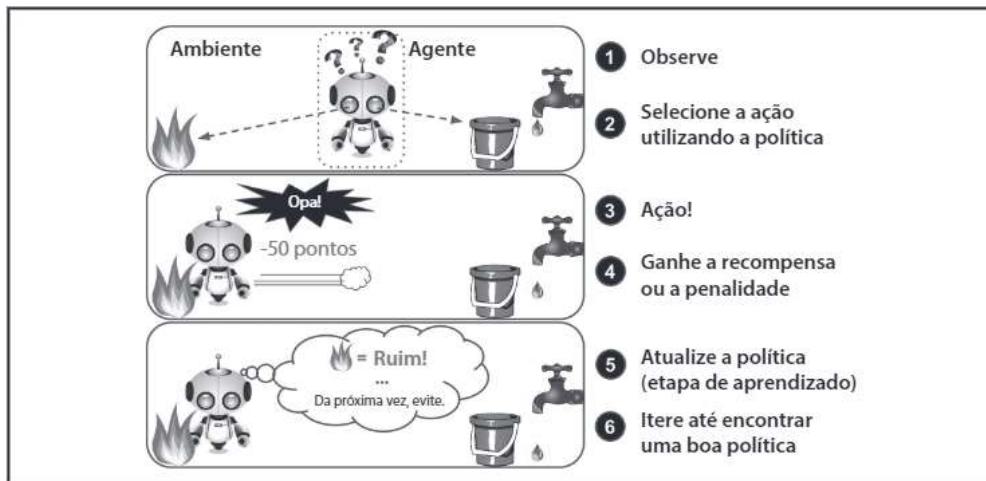


Figura 1-12. Aprendizado por reforço

Por exemplo, muitos robôs implementam algoritmos de aprendizado por reforço para aprender a andar. O programa AlphaGo da DeepMind também é um bom exemplo de aprendizado por reforço: ele apareceu nas manchetes em maio de 2017 quando venceu o campeão mundial Ke Jie no jogo Go. Ele desenvolveu sua política vencedora analisando milhões de jogos e depois praticando muito contra si mesmo. Note que o aprendizado foi desativado durante os jogos contra o campeão; o AlphaGo apenas aplicou a política que aprendeu.

## Aprendizado Online e em Lote

Outro critério utilizado para classificar os sistemas de Aprendizado de Máquina é se o sistema pode ou não aprender de forma incremental a partir de um fluxo de dados recebido.

### Aprendizado em lote

No *aprendizado em lote*, o sistema é incapaz de aprender de forma incremental: ele deve ser treinado com a utilização de todos os dados disponíveis. Isso geralmente demandará muito tempo e recursos de computação, portanto, normalmente é feito offline. Primeiro, o sistema é treinado, em seguida, é lançado em produção, e roda sem aprender mais nada; apenas aplicando o que aprendeu. Isso é chamado de *aprendizado offline*.

Se você quiser que um sistema de aprendizado em lote conheça novos dados (como um novo tipo de spam), é preciso treinar uma nova versão do sistema a partir do zero no conjunto completo de dados (não apenas os novos dados, mas também os antigos), então parar o sistema antigo e substituí-lo pelo novo.

Felizmente, todo o processo de treinamento, avaliação e lançamento de um sistema de Aprendizado de Máquina pode ser automatizado facilmente (como mostrado na Figura 1-3), então mesmo um sistema de aprendizado em lote pode se adaptar às mudanças. Basta atualizar os dados e treinar a partir do zero uma nova versão do sistema sempre que necessário.

Esta solução é simples e geralmente funciona bem, mas, com a utilização do conjunto completo de dados, o treinamento pode demorar muitas horas, então você normalmente treinará um novo sistema apenas a cada 24 horas ou mesmo semanalmente. Se seu sistema precisa se adaptar a dados que mudam rapidamente (por exemplo, prever os preços das ações), você precisa de uma solução mais reativa.

Além disso, o treinamento no conjunto completo de dados requer muitos recursos de computação (CPU, espaço de memória, espaço em disco, E/S do disco, E/S de rede, etc.). Se você tem muitos dados e seu sistema é automatizado para treinar todos os dias a partir do zero, esse processo ficará muito caro. Se a quantidade de dados for enorme, talvez seja impossível utilizar um algoritmo de aprendizado em lote.

Finalmente, se o seu sistema precisa ser capaz de aprender de forma autônoma e tem recursos limitados (por exemplo, um aplicativo de smartphone ou um rover em Marte), então, seria um grave erro carregar grandes quantidades de dados de treinamento e usar inúmeros recursos para treinar por horas todos os dias.

Felizmente, uma opção melhor em todos esses casos seria utilizar algoritmos capazes de aprender de forma incremental.

### Aprendizado online

No *aprendizado online*, você treina o sistema de forma incremental, alimentando sequencialmente as instâncias de dados individualmente ou em pequenos grupos, chamados de *minilotes*. Cada etapa do aprendizado é rápida e barata, então o sistema pode aprender rapidamente sobre os novos dados assim que eles chegam (veja Figura 1-13).

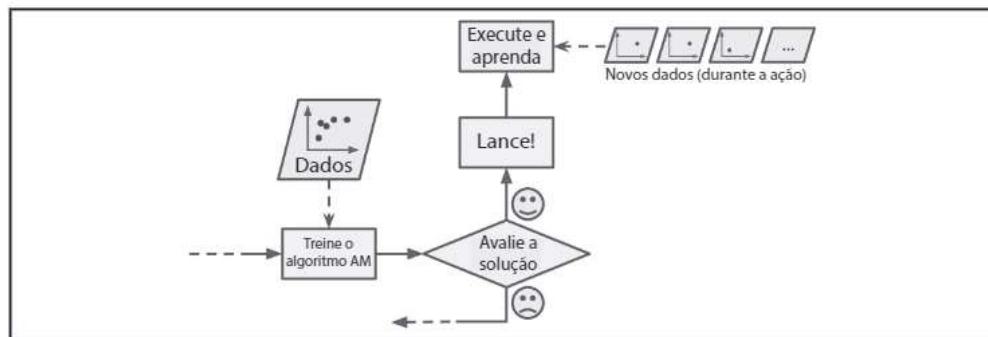


Figura 1-13. Aprendizado online

O aprendizado online é excelente para sistemas que recebem dados como um fluxo contínuo (por exemplo, preços das ações) e precisam se adaptar às mudanças rápida ou autonomamente. Também é uma boa opção se seus recursos de computação são limitados: uma vez que um sistema de aprendizado online aprendeu sobre as novas instâncias de dados, ele não precisa mais delas, então você pode descartá-las (a menos que queira reverter para um estágio anterior e “reproduzir” os dados). O que pode economizar muito espaço.

Os algoritmos de aprendizado online também podem ser utilizados para treinar sistemas em grandes conjuntos de dados que não cabem na memória principal de uma máquina (isto é chamado de *out-of-core learning*). O algoritmo carrega parte dos dados, executa uma etapa do treinamento nesses dados e repete o processo até que ele tenha sido executado em todos os dados (veja Figura 1-14).



Todo esse processo geralmente é feito offline (ou seja, não no sistema ao vivo), então *aprendizado online* pode ser um nome confuso. Pense nisso como *aprendizado incremental*.

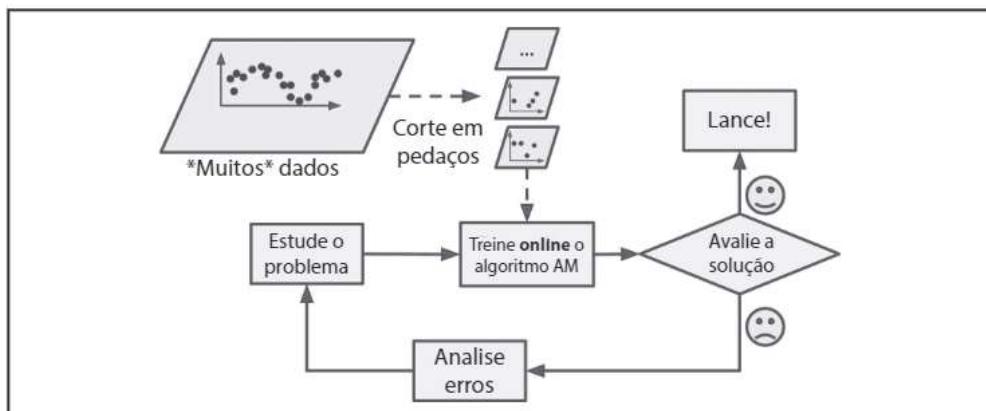


Figura 1-14. Utilizando aprendizado online para lidar com grandes conjuntos de dados

Um parâmetro importante dos sistemas de aprendizado online é a rapidez com que eles devem se adaptar às mudanças dos dados: isto é chamado de *taxa de aprendizado*. Se você definir uma alta taxa de aprendizado, seu sistema se adaptará rapidamente aos novos dados, mas também tenderá a se esquecer rapidamente os antigos (você não quer que um filtro de spam sinalize apenas os tipos mais recentes de spam). Por outro lado, se você definir uma baixa taxa de aprendizado, o sistema terá mais inércia; isto é, aprenderá mais devagar, mas também será menos sensível ao apontar novos dados ou sequências de pontos de dados não representativos.

Um grande desafio no aprendizado online é que, se incluirmos dados ruins no sistema, seu desempenho diminuirá gradualmente. Se estamos falando de um sistema ao vivo, seus clientes perceberão. Por exemplo, dados ruins podem vir de um sensor com mau funcionamento em um robô, ou de alguém que envia um spam a um mecanismo de pesquisa para tentar se posicionar no topo. Para reduzir esse risco, você precisa monitorar de perto o sistema e desligar o aprendizado rapidamente se detectar uma queda no desempenho (e possivelmente reverter para um estágio anterior de trabalho). Você também poderá monitorar os dados de entrada e reagir a dados anormais (por exemplo, com a utilização de um algoritmo de detecção de anomalias).

## Aprendizado Baseado em Instância Versus Aprendizado Baseado em Modelo

Mais uma forma de categorizar os sistemas de Aprendizado de Máquina é por meio da *generalização*. A maioria das tarefas de Aprendizado de Máquina faz previsões. Isso significa que, dada uma série de exemplos de treinamento, o sistema precisa ser capaz de generalizar em exemplos que nunca viu antes. Ter uma boa medida do desempenho nos dados de treinamento é bom, mas insuficiente; o verdadeiro objetivo é ter um bom desempenho em novas instâncias.

Existem duas abordagens principais para a generalização: aprendizado baseado em instâncias e aprendizado baseado em modelo.

### Aprendizado baseado em instância

Possivelmente, a forma mais trivial de aprendizado é simplesmente decorar. Se você fosse criar um filtro de *spam* desta forma, ele apenas marcaria todos os e-mails que são idênticos em relação aos e-mails que já foram marcados pelos usuários — não seria a pior solução, mas certamente não é a melhor.

Em vez de marcar apenas e-mails que são idênticos aos e-mails de spam conhecidos, seu filtro de spam pode ser programado para marcar também e-mails que são muito semelhantes aos e-mails conhecidos de spam. Isso requer uma *medida de similaridade* entre dois e-mails. Uma medida de similaridade (muito básica) entre dois e-mails poderia ser contar o número de palavras que eles têm em comum. O sistema marcaria um e-mail como spam se tivesse muitas palavras em comum com um e-mail de spam conhecido.

Isso é chamado de *aprendizado baseado em instância*: o sistema aprende os exemplos por meio da memorização e, em seguida, generaliza para novos casos utilizando uma medida de similaridade (Figura 1-15).

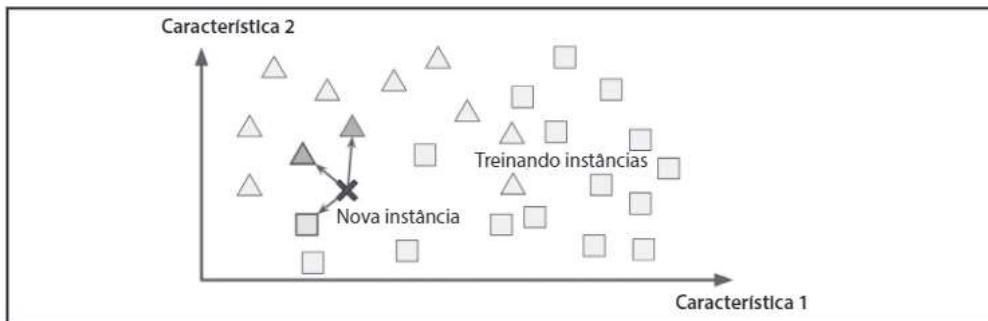


Figura 1-15. Aprendizado baseado em instância

### Aprendizado baseado em modelo

Outra maneira de generalizar a partir de um conjunto de exemplos seria construir um modelo desses exemplos e utilizar esse modelo para fazer *previsões*. Isso é chamado de *aprendizado baseado em modelo* (Figura 1-16).

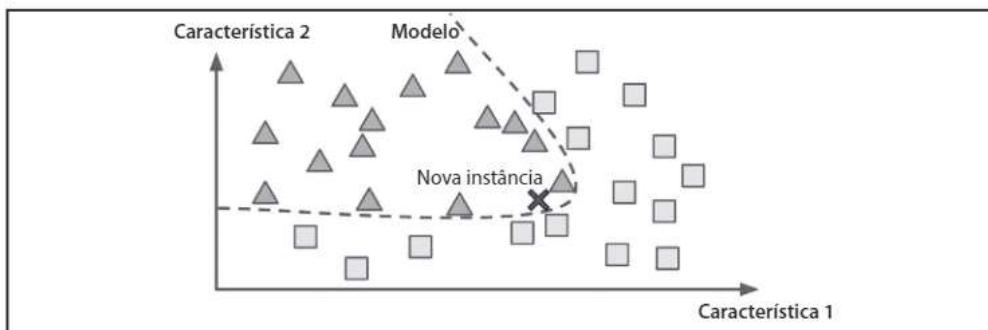


Figura 1-16. Aprendizado baseado em modelo

Por exemplo, suponha que você queira saber se o dinheiro faz as pessoas felizes; então você baixa os dados do *Better Life Index* no site da OCDE (<https://goo.gl/0Eht9W>), bem como as estatísticas sobre o PIB per capita no site do FMI (<http://goo.gl/j1MSKe>). Logo, você junta as tabelas e classifica pelo PIB per capita. A Tabela 1-1 exemplifica uma amostra do que você obtém.

Tabela 1-1. O dinheiro torna as pessoas mais felizes?

País	PIB per capita (USD)	Satisfação de vida
Hungria	12.240	4,9
Coreia	27.195	5,8
França	37.675	6,5
Austrália	50.962	7,3
Estados Unidos	55.805	7,2

Vamos plotar os dados para alguns países aleatórios (Figura 1-17).

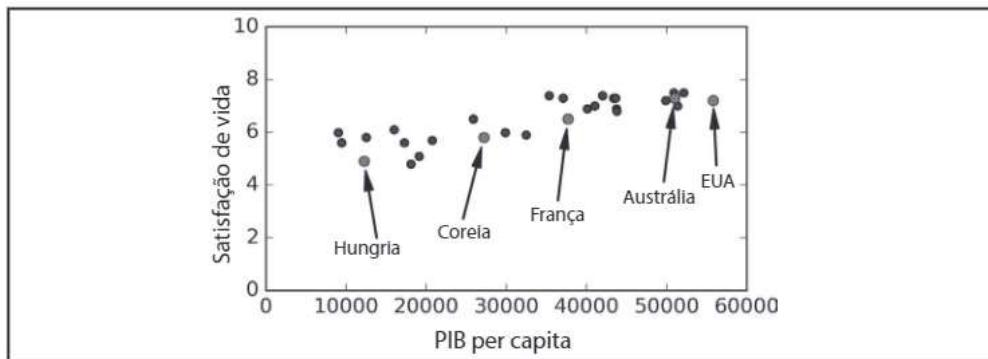


Figura 1-17. Você vê uma tendência aqui?

Parece haver uma tendência aqui! Embora os dados estejam *ruidosos* (ou seja, parcialmente aleatórios), parece que a satisfação de vida aumenta de forma mais ou menos linear à medida que aumenta o PIB per capita do país. Então você decide modelar a satisfação de vida [*life\_satisfaction*] como uma função linear do PIB per capita [*GDP\_per\_capita*]. Esta etapa é chamada de *seleção do modelo*: você selecionou um *modelo linear* de satisfação de vida com apenas um atributo, o PIB per capita (Equação 1-1).

*Equação 1-1. Um simples modelo linear*

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

Este modelo tem dois *parâmetros do modelo*,  $\theta_0$  e  $\theta_1$ .<sup>5</sup> Ao ajustar esses parâmetros, você pode fazer com que seu modelo represente qualquer função linear, como mostrado na Figura 1-18.

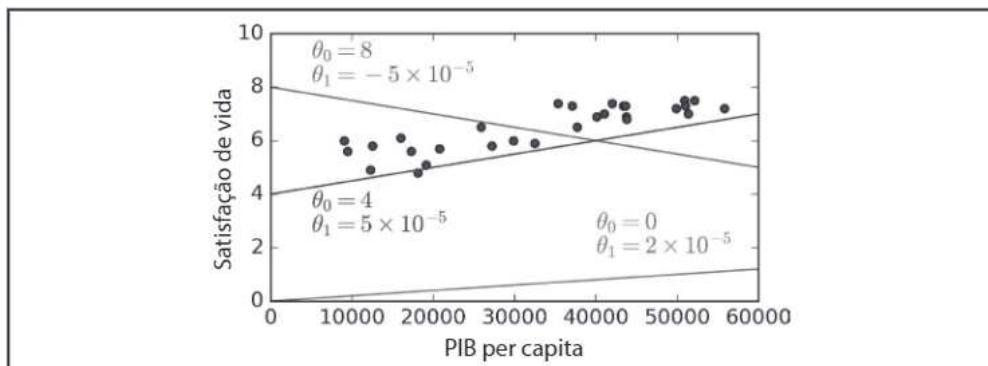


Figura 1-18. Alguns modelos lineares possíveis

<sup>5</sup> Por convenção, a letra grega  $\theta$  (*theta*) é frequentemente utilizada para representar os parâmetros do modelo.

Antes de poder utilizar seu modelo, você precisa definir os valores dos parâmetros  $\theta_0$  e  $\theta_1$ . Como você pode saber quais valores farão seu modelo funcionar melhor? Para responder a esta pergunta, você precisa especificar uma medida de desempenho. Você pode definir uma *função de utilidade* (ou *função fitness*) que mede o quanto *bom* seu modelo é, ou uma *função de custo*, que mede o quanto *ruim* ele é. Para problemas de regressão linear, as pessoas geralmente utilizam uma função de custo que mede a distância entre as previsões do modelo linear e os exemplos de treinamento; o objetivo é minimizar essa distância.

É aqui que o algoritmo de regressão linear entra: você o alimenta com seus exemplos de treinamento e ele encontra os parâmetros que tornam o modelo linear mais adequado aos seus dados. Isso é chamado de *treinar* o modelo. No nosso caso, o algoritmo descobre que os valores dos parâmetros otimizados são  $\theta_0 = 4,85$  e  $\theta_1 = 4,91 \times 10^{-5}$ .

Agora, o modelo se ajusta o mais próximo possível dos dados do treinamento (para um modelo linear), como você pode ver na Figura 1-19.

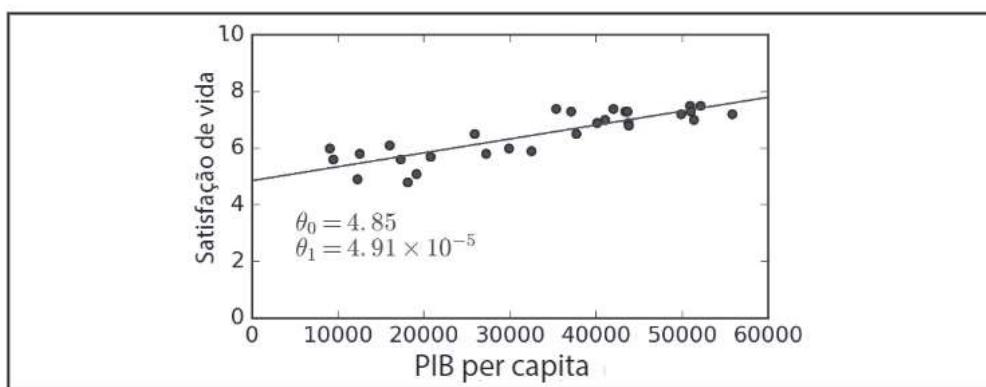


Figura 1-19. O modelo linear que melhor se ajusta aos dados de treinamento

Você finalmente está pronto para executar o modelo e fazer previsões. Por exemplo, digamos que quer saber o nível de felicidade dos cipriotas, e os dados da OCDE não têm a resposta. Felizmente, você pode utilizar o seu modelo para fazer uma boa previsão: você olha para o PIB per capita do Chipre, encontra US\$22.587 e, em seguida, aplica seu modelo e verifica que a satisfação de vida estará provavelmente ao redor de  $4,85 + 22.587 \times 4,91 \times 10^{-5} = 5,96$ .

Para estimular o seu apetite, o Exemplo 1-1 mostra o código Python que carrega os dados, os prepara,<sup>6</sup> cria um diagrama de dispersão para visualização e, em seguida, treina um modelo linear e faz uma previsão.<sup>7</sup>

<sup>6</sup> O código assume que `prepare_country_stats()` já está definido: ele mescla os dados de PIB e de satisfação de vida em um único *dataframe* do Pandas.

<sup>7</sup> Tudo bem se você não entender todo o código ainda; apresentaremos o Scikit-Learn nos próximos capítulos.

*Exemplo 1-1. Treinando e executando um modelo linear utilizando o Scikit-Learn*

```

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# Carregue os dados
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv", thousands=',', delimiter='\t',
                             encoding='latin1', na_values="n/a")

# Prepare os dados
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Visualize os dados
country_stats.plot(kind='scatter', x="GDP per capita", y='Life satisfaction')
plt.show()

# Selecione um modelo linear
model = sklearn.linear_model.LinearRegression()

# Treine o modelo
model.fit(X, y)

# Faça uma predição para o Chipre
X_new = [[22587]] # Chipre' GDP per capita
print(model.predict(X_new)) # outputs [[ 5.96242338]]

```



Se você tivesse optado por utilizar um algoritmo de aprendizado baseado em instâncias, teria visto que a Eslovênia tem o PIB per capita mais próximo do Chipre (US\$20.732), e, como os dados da OCDE nos dizem que a satisfação da vida dos eslovenos é 5,7, você teria previsto uma satisfação de vida de 5,7 para o Chipre. Se você reduzir um pouco e observar os dois países mais próximos, encontrará Portugal e Espanha com satisfações de vida de 5,1 e 6,5, respectivamente. Fazendo a média desses três valores, obtém-se 5,77, o que é bastante próximo da sua previsão baseada em modelo. Este algoritmo simples é chamado de regressão *k-nearest neighbors* (neste exemplo, k = 3).

Para substituir o modelo de regressão linear com a regressão *k-nearest neighbors* no código anterior, simplesmente substitua esta linha:

```
model = sklearn.linear_model.LinearRegression()
```

por esta:

```
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

Se tudo correu bem, seu modelo fará boas previsões. Caso contrário, talvez seja necessário utilizar mais atributos (taxa de emprego, saúde, poluição do ar, etc.), obter mais ou melhor qualidade dos dados de treinamento, ou talvez selecionar um modelo mais poderoso (por exemplo, um modelo de Regressão Polinomial).

Em resumo:

- Você estudou os dados;
- Selecioneou um modelo;
- Treinou-o com os dados de treinamento (ou seja, o algoritmo de aprendizado procurou os valores dos parâmetros do modelo que minimizam uma função de custo);
- Finalmente, você aplicou o modelo para fazer previsões em novos casos (isto é chamado de *Inferência*), na expectativa de que esse modelo generalize bem.

Um projeto típico de Aprendizado de Máquina é assim. No Capítulo 2, você experimentará isso em primeira mão, de ponta a ponta em um projeto.

Já caminhamos muito até aqui: agora você já sabe o que é realmente o Aprendizado de Máquina, por que ele é útil, quais são algumas das categorias mais comuns de sistemas do AM e como é um fluxo de trabalho típico do projeto. Agora, veremos o que pode dar errado no aprendizado e impedir que você faça previsões precisas.

## Principais Desafios do Aprendizado de Máquina

Em suma, uma vez que a sua tarefa principal é selecionar um algoritmo de aprendizado e treiná-lo em alguns dados, as duas coisas que podem dar errado são: “algoritmos ruins” e “dados ruins”. Comecemos com exemplos de dados ruins.

### Quantidade Insuficiente de Dados de Treinamento

Para que uma criança aprenda o que é uma maçã, é preciso que você aponte para uma maçã e diga “maçã” (possivelmente repetindo algumas vezes esse procedimento). Agora, a criança consegue reconhecer maçãs em todos os tipos de cores e formas. Genial.

O Aprendizado de Máquina ainda não está lá; é preciso uma grande quantidade de dados para que a maioria dos algoritmos de Aprendizado de Máquina funcione corretamente. Você precisará de milhares de exemplos mesmo para problemas muito simples, e para problemas complexos, como reconhecimento de imagem ou da fala, precisará de milhões de exemplos (a menos que possa reutilizar partes de um modelo existente).

## A Eficácia Não Razoável dos Dados

Em um artigo famoso (<http://goo.gl/R5enIE>) publicado em 2001, os pesquisadores da Microsoft Michele Banko e Eric Brill mostraram que, uma vez que dados suficientes foram fornecidos, algoritmos do aprendizado de máquina muito diferentes, mesmo os mais simples, tiveram um desempenho quase idêntico em um problema complexo de desambiguação<sup>8</sup> (como você pode ver na Figura 1-20).

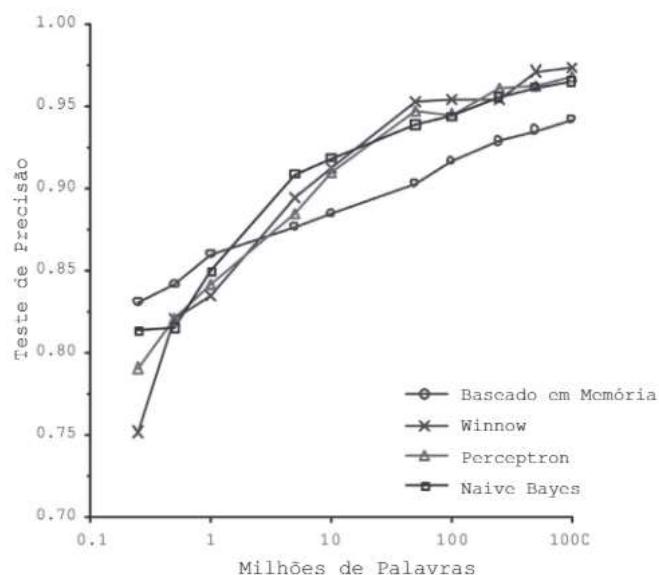


Figura 1-20. A importância dos dados versus algoritmos<sup>9</sup>

Como os autores afirmam: “esses resultados sugerem que talvez possamos reconsiderar o custo-benefício entre gastar tempo e dinheiro no desenvolvimento de algoritmos ou gastá-los no desenvolvimento de *corpus*.”

A ideia de que, para problemas complexos, os dados são mais importantes do que algoritmos foi popularizada por Peter Norvig *et al.* em um artigo intitulado “The Unreasonable Effectiveness of Data” [“A Eficácia Não Razoável dos Dados”, em tradução livre] (<http://goo.gl/q6LaZ8>) publicado em outubro de 2009.<sup>10</sup> No entanto, vale notar que os conjuntos de dados de pequenas e médias dimensões ainda são muito comuns e nem sempre é fácil ou barato obter dados extras de treinamento, então não abandone os algoritmos ainda.

<sup>8</sup> Por exemplo, em inglês, saber quando escrever “to”, “two”, ou “too” dependendo do contexto.

<sup>9</sup> Figura reproduzida com permissão de Banko e Brill (2001), “Learning Curves for Confusion Set Disambiguation”.

<sup>10</sup> “The Unreasonable Effectiveness of Data” Peter Norvig *et al.* (2009).

## Dados de Treinamento Não Representativos

A fim de generalizar bem, é crucial que seus dados de treinamento sejam representativos dos novos casos para os quais você deseja generalizar, não importa se você utiliza o aprendizado baseado em instâncias ou o aprendizado baseado em modelo.

Por exemplo, o conjunto de países que utilizamos anteriormente para treinar o modelo linear não era perfeitamente representativo; faltavam alguns países. A Figura 1-21 mostra como ficam os dados quando os países que faltam são adicionados.

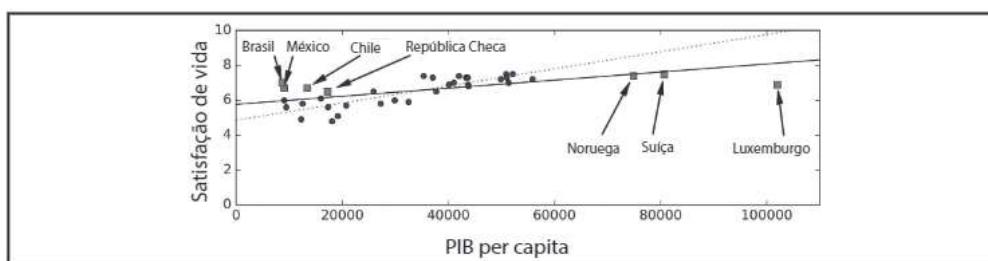


Figura 1-21. Uma amostra de treinamento mais representativa

Se você treinar um modelo linear nesses dados, obterá a linha sólida, enquanto o modelo antigo está representado pela linha pontilhada. Como pode ser observado, adicionar alguns países faltantes não só altera significativamente o modelo, como deixa claro que um modelo linear tão simples provavelmente nunca funcionará bem. Parece que países muito ricos não são mais felizes do que países moderadamente ricos (na verdade, parecem mais infelizes) e, inversamente, alguns países pobres parecem ser mais felizes do que muitos países ricos.

Ao utilizar um conjunto de treinamento não representativo, treinamos um modelo que dificilmente fará previsões precisas, especialmente para países muito pobres e muito ricos.

É crucial utilizar um conjunto de treinamento representativo nos casos em que desejamos generalizar. Isso pode ser mais difícil do que parece: se a amostra for muito pequena, existirá um *ruído de amostragem* (ou seja, dados não representativos como resultado do acaso), mas mesmo as amostras muito grandes podem não ser representativas se o método de amostragem for falho. Isso é chamado de viés de amostragem.

### Um exemplo famoso de Viés de Amostragem

Talvez o exemplo mais famoso de viés de amostragem tenha acontecido durante as eleições presidenciais dos EUA em 1936, na disputa de Landon contra Roosevelt: a *Literary Digest* conduziu uma grande pesquisa enviando cartas pelo correio para cerca de 10 milhões de pessoas. Obteve um retorno de 2,4 milhões de respostas e previu com grande confiança que Landon deveria obter 57% dos votos.

Mas, na verdade, Roosevelt venceu as eleições com 62% dos votos. A falha aconteceu no método de amostragem da *Literary Digest*:

- Em primeiro lugar, ao obter os endereços para o envio das pesquisas, a *Literary Digest* utilizou listas telefônicas, de assinantes, de membros de clubes, dentre outras. Todas essas listas tendem a favorecer pessoas mais ricas, mais propensas a votar em republicanos (daí Landon).
- Em segundo lugar, menos de 25% das pessoas responderam à enquete. Novamente, ao descartar pessoas que não se importam muito com a política, pessoas que não gostam da *Literary Digest* e outros grupos-chave, é introduzido um viés de amostragem, chamado de *viés de falta de resposta*.

Aqui temos outro exemplo: digamos que você deseja criar um sistema para o reconhecimento de vídeos de música funk. Uma forma de construí-lo seria pesquisar “música funk” no YouTube e utilizar os vídeos resultantes. Mas isso pressupõe que o mecanismo de pesquisa do YouTube retornará um conjunto de vídeos que representa todos os vídeos de música no YouTube. Provavelmente, os artistas populares terão resultados tendenciosos na pesquisa (e, se você mora no Brasil, receberá muitos vídeos de “funk carioca” que não se parecem nada com James Brown). Por outro lado, de que outra forma você consegue obter um grande conjunto de treinamento?

## Dados de Baixa Qualidade

Obviamente, se seus dados de treinamento estiverem cheios de erros, outliers e ruídos (por exemplo, devido a medições de baixa qualidade), o sistema terá mais dificuldade para detectar os padrões subjacentes, portanto, menos propício a um bom funcionamento. Muitas vezes vale a pena perder um tempo limpando seus dados de treinamento. A verdade é que a maioria dos cientistas de dados gasta uma parte significativa de seu tempo fazendo exatamente isso. Por exemplo:

- Se algumas instâncias são claramente outliers, isso pode ajudar a descartá-las ou tentar manualmente a correção dos erros;
- Se faltam algumas características para algumas instâncias (por exemplo, 5% dos seus clientes não especificaram sua idade), você deve decidir se deseja ignorar completamente esse atributo, se deseja ignorar essas instâncias, preencher os valores ausentes (por exemplo, com a média da idade), ou treinar um modelo com a característica e um modelo sem ela, e assim por diante.

## Características Irrelevantes

Como diz o ditado: entra lixo, sai lixo. Seu sistema só será capaz de aprender se os dados de treinamento contiverem características relevantes suficientes e poucas características irrelevantes. Uma parte crítica do sucesso de um projeto de Aprendizado de Máquina é criar um bom conjunto de características para o treinamento, processo chamado de *feature engineering* que envolve:

- *Seleção das características*: selecionar as mais úteis para treinar entre as existentes;
- *Extração das características*: combinar características existentes para produzir uma mais útil (como vimos anteriormente, os algoritmos de redução de dimensionalidade podem ajudar);
- Criação de novas características ao coletar novos dados.

Agora que examinamos vários exemplos ruins de dados, vejamos alguns exemplos ruins de algoritmos.

## Sobreajustando os dados de treinamento

Digamos que você está visitando um país estrangeiro e o taxista o rouba. Você é tentado a dizer que *todos* os motoristas de táxi nesse país são ladrões. Generalizar é algo que nós humanos fazemos com muita frequência e, infelizmente, as máquinas podem cair na mesma armadilha se não tivermos cuidado. No Aprendizado de Máquina, isso é chamado de *sobreajuste*: significa que o modelo funciona bem nos dados de treinamento, mas não generaliza tão bem.

A Figura 1-22 mostra um exemplo de um modelo polinomial de satisfação de vida de alto grau que se sobreajusta fortemente nos dados de treinamento. Você realmente confiaria em suas previsões, se o seu desempenho é muito melhor em dados de treinamento do que no modelo linear simples?

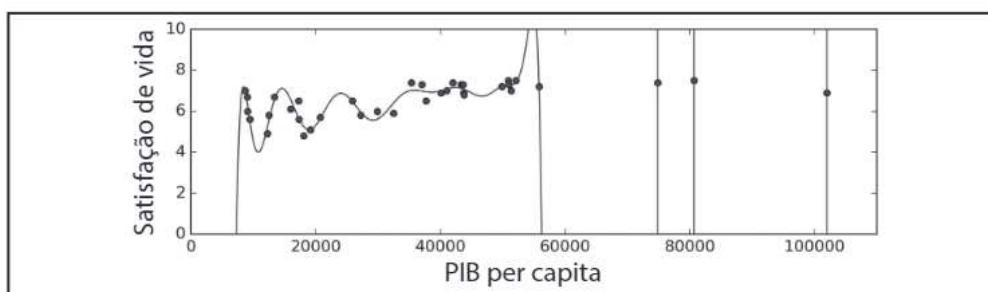


Figura 1-22. Sobreajustando nos dados de treinamento

Modelos complexos como redes neurais profundas podem detectar padrões sutis nos dados, mas se o conjunto de treinamento é ruidoso ou se é muito pequeno (o que introduz ruído na amostragem), então o modelo provavelmente detectará padrões no próprio ruído. É óbvio que esses padrões não serão generalizados para novas instâncias. Por exemplo, digamos que você alimenta o seu modelo de satisfação de vida com muitos outros atributos, incluindo alguns não informativos, como o nome do país. Nesse caso, um modelo complexo poderia detectar padrões como o fato de que todos os países com um W em seu nome em inglês têm uma satisfação de vida superior a 7: New Zealand (7,3), Norway (7,4), Sweden (7,2) e Switzerland (7,5). Você confiaria nesta regra que também generalizará para países como Rwanda ou Zimbabwe? É evidente que esse padrão nos dados ocorreu por acaso, mas o modelo não tem como saber se um padrão é real ou simplesmente resultado de ruído nos dados.



O sobreajuste acontece quando o modelo é muito complexo em relação à quantidade e ao ruído dos dados de treinamento. As possíveis soluções são:

- Simplificar o modelo ao selecionar um com menos parâmetros (por exemplo, um modelo linear em vez de um modelo polinomial de alto grau), reduzindo o número de atributos nos dados de treinamento ou restringindo o modelo;
- Coletar mais dados de treinamento;
- Reduzir o ruído nos dados de treinamento (por exemplo, corrigir erros de dados e remover outliers).

Chamamos de *regularização* quando restringimos um modelo para simplificar e reduzir o risco de sobreajuste. Por exemplo, o modelo linear que definimos anteriormente possui dois parâmetros,  $\theta_0$  e  $\theta_1$ . Isso dá ao algoritmo de aprendizado *dois graus* de liberdade para adaptar o modelo aos dados de treinamento: ele pode ajustar tanto a altura ( $\theta_0$ ) quanto a inclinação ( $\theta_1$ ) da linha. Se forçássemos  $\theta_1 = 0$  o algoritmo teria apenas um grau de liberdade e teria muito mais dificuldade para ajustar os dados corretamente: ele apenas poderia mover a linha para cima ou para baixo para chegar o mais próximo possível das instâncias de treinamento e finalizar em torno da média. Este é um modelo muito simples! Mas, se permitirmos que o algoritmo modifique  $\theta_1$  e forçarmos para que ele se mantenha reduzido, então o algoritmo efetivamente terá algo entre um e dois graus de liberdade. Ele produzirá um modelo mais simples do que aquele com dois graus de liberdade, mas mais complexo do que o modelo com apenas um. Queremos encontrar o equilíbrio certo entre o ajuste perfeito dos dados e a necessidade de manter o modelo simples o suficiente para garantir que ele generalize bem.

A Figura 1-23 mostra três modelos: a linha pontilhada representa o modelo original que foi treinado com poucos países, a linha tracejada é o nosso segundo modelo treinado com todos os países, e a linha sólida é um modelo linear treinado com os mesmos dados do primeiro modelo, mas com uma restrição de regularização. Veja que a regularização forçou o modelo a ter uma inclinação menor, o que faz com que tenha um ajuste menor dos dados no modelo treinado mas, na verdade, permite generalizar melhor em novos exemplos.

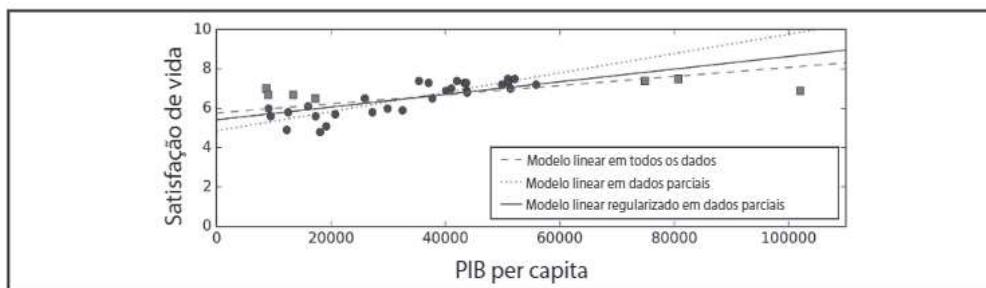


Figura 1-23. A regularização reduz o risco de sobreajustes

A quantidade de regularização a ser aplicada por um *hiperparâmetro* durante o aprendizado pode ser controlada. Um hiperparâmetro é um parâmetro de um algoritmo de aprendizado (não do modelo). Como tal, não é afetado pelo próprio algoritmo de aprendizado; deve ser definido antes do treinamento e permanecer constante durante ele. Se você configurar o hiperparâmetro de regularização para um valor muito alto, obterá um modelo quase plano (uma inclinação próxima a zero); o algoritmo de aprendizado certamente não se sobreajustará nos dados de treinamento, mas terá menos chances de encontrar uma boa solução. O ajuste dos hiperparâmetros é uma parte importante da construção de um sistema de Aprendizado de Máquina (você verá um exemplo detalhado no próximo capítulo).

## Subajustando os dados de Treinamento

Como você pode ver, *subajuste* é o oposto de sobreajuste: ocorre quando seu modelo é muito simples para o aprendizado da estrutura subjacente dos dados. Por exemplo, um modelo linear de satisfação de vida é propenso a ser subajustado; a realidade é mais complexa do que o modelo, por isso suas previsões tendem a ser imprecisas mesmo nos exemplos de treinamento.

As principais opções para resolver esses problemas são:

- Selecionar um modelo mais poderoso, com mais parâmetros;

- Alimentar o algoritmo de aprendizado com melhores características (*feature engineering*);
- Reduzir as restrições no modelo (por exemplo, reduzindo o hiperparâmetro de regularização).

## Voltando Atrás

Agora você já sabe muito sobre o Aprendizado de Máquina. No entanto, passamos por tantos conceitos que você pode estar se sentindo um pouco perdido, então vamos dar um passo atrás e olhar para o quadro geral:

- Aprendizado de Máquina é fazer com que as máquinas evoluam em algumas tarefas aprendendo com os dados, em vez de ter que programar as regras explicitamente;
- Existem muitos tipos diferentes de sistemas do AM: supervisionados ou não, em lote ou online, baseados em instâncias ou em modelos, e assim por diante;
- Em um projeto do AM, você coleta dados em um conjunto de treinamento e os fornece para um algoritmo de aprendizado. Se o algoritmo estiver baseado em modelo, ele ajusta alguns parâmetros para adequar o modelo ao conjunto de treinamento (ou seja, para fazer boas previsões no próprio conjunto de treinamento), e então, se tudo der certo, também poderá fazer boas previsões em novos casos. Se o algoritmo for baseado em instância, ele simplesmente decora os exemplos e utiliza uma medida de similaridade generalizando para novas instâncias;
- O sistema não terá um bom funcionamento se o seu conjunto de treinamento for muito pequeno ou se os dados não forem representativos, ruidosos ou poluídos com características irrelevantes (entra lixo, sai lixo). Por último, o seu modelo não precisa ser nem simples demais (subajustado) nem muito complexo (superajustado).

Abordaremos um último tópico importante: uma vez treinado um modelo, você não vai querer apenas “torcer” para que ele generalize para novos casos. Você vai querer avaliar e ajustar se necessário. Vejamos como.

## Testando e Validando

A única maneira de saber o quanto bem um modelo generalizará em novos casos é de fato testá-lo em novos casos. Uma forma de pôr isso em prática é colocar seu modelo em produção e monitorar a qualidade do seu desempenho. Isso funciona bem, mas, se seu modelo for muito ruim, seus usuários se queixarão — e esta não é a melhor ideia.

Uma opção melhor seria dividir seus dados em dois conjuntos: o *conjunto de treinamento* e o *conjunto de teste*. Como esses nomes implicam, você treina seu modelo utilizando o conjunto de treinamento e o testa utilizando o conjunto de teste. A taxa de erro em novos casos é chamada de *erro de generalização* (ou *erro fora da amostra*) e, ao avaliar seu modelo no conjunto de teste, você obtém uma estimativa desse erro. O valor indica se o seu modelo terá um bom funcionamento em instâncias inéditas.

Se o erro de treinamento for baixo (ou seja, seu modelo comete alguns erros no conjunto de treinamento), mas seu erro de generalização é alto, isso significa que seu modelo está se sobreajustando aos dados de treinamento.



É comum utilizar 80% dos dados para treinamento e *reservar* 20% para o teste.

É muito simples avaliar um modelo: basta utilizar um conjunto de teste. Agora, suponha que você hesite entre dois modelos (digamos um modelo linear e um modelo polinomial): como decidir? Uma opção será treinar ambos utilizando o conjunto de teste e comparar se eles generalizam bem.

Agora, suponha que o modelo linear generalize melhor, mas você deseja aplicar alguma regularização para evitar o sobreajuste. A questão é: como você escolhe o valor do hiperparâmetro de regularização? Uma opção seria treinar 100 modelos diferentes utilizando 100 valores diferentes para este hiperparâmetro. Suponha que você encontre o melhor valor de hiperparâmetro que produza um modelo com o menor erro de generalização, digamos, apenas 5% de erro.

Então, você coloca esse modelo em produção, mas infelizmente ele não funciona tão bem quanto o esperado e produz 15% de erros. O que acabou de acontecer?

O problema é que você mediou o erro de generalização por várias vezes no conjunto de teste e adaptou o modelo e os hiperparâmetros para produzir o melhor modelo *para esse conjunto*. Isso significa que o modelo provavelmente não funcionará tão bem com novos dados.

Uma solução comum para este problema é ter um segundo conjunto de retenção chamado *conjunto de validação*. Você treina vários modelos com vários hiperparâmetros utilizando o conjunto de treinamento, seleciona o modelo e os hiperparâmetros que atuam melhor no conjunto de validação, e, quando estiver contente com seu modelo, executa um único teste final no conjunto de teste para obter uma estimativa do erro de generalização.

Para evitar “desperdiçar” muitos dados de treinamento em conjuntos de validação, uma técnica comum é utilizar a *validação cruzada*: o conjunto de treinamento é dividido em subconjuntos complementares e cada modelo é treinado com uma combinação diferente desses subconjuntos e validado em relação às partes restantes. Uma vez selecionados o tipo de modelo e os hiperparâmetros, um modelo final é treinado com a utilização desses hiperparâmetros no conjunto completo de treinamento e o erro generalizado é medido no conjunto de testes.

### Teorema do Não Existe Almoço Grátis

Um modelo é uma versão simplificada das observações. As simplificações destinam-se a descartar os detalhes supérfluos que provavelmente não serão generalizados em novas instâncias. No entanto, para decidir quais dados descartar e quais manter, você deve fazer suposições. Por exemplo, um modelo linear supõe que os dados são fundamentalmente lineares e que a distância entre as instâncias e a linha reta é apenas o ruído, que pode ser ignorado com segurança.

Em um famoso artigo de 1996 (<https://goo.gl/dzp946>),<sup>11</sup> David Wolpert demonstrou que, se você não fizer suposição alguma sobre os dados, então não há motivo para preferir um ou outro modelo. Isso é chamado de Teorema do *Não existe almoço grátis* (NFL, do inglês "no free lunch"). Para alguns conjuntos de dados, o melhor modelo é um modelo linear, enquanto para outros conjuntos será uma rede neural. Não existe um modelo que tenha a garantia de funcionar melhor a priori (daí o nome do teorema). A única maneira de saber com certeza qual seria o melhor modelo é a avaliação de todos. Como isso não é possível, na prática você parte de alguns pressupostos razoáveis sobre os dados e avalia apenas alguns modelos razoáveis. Por exemplo, para tarefas simples, você pode avaliar modelos lineares com vários níveis de regularização e, para um problema complexo, você pode avaliar várias redes neurais.

## Exercícios

Neste capítulo, abordamos alguns dos conceitos mais importantes do Aprendizado de Máquina. Nos próximos, vamos mergulhar mais fundo e escrever mais código, mas, antes disso, certifique-se de responder as seguintes perguntas:

1. Como você definiria o Aprendizado de Máquina?
2. Você saberia nomear quatro tipos de problemas nos quais o AM se destaca?
3. O que é um conjunto de treinamento rotulado?
4. Quais são as duas tarefas supervisionadas mais comuns?
5. Você consegue nomear quatro tarefas comuns sem supervisão?

<sup>11</sup> “The Lack of A Priori Distinctions Between Learning Algorithms”, D. Wolpert (1996).

6. Que tipo de algoritmo de Aprendizado de Máquina você utilizaria para permitir que um robô ande em vários terrenos desconhecidos?
7. Que tipo de algoritmo você utilizaria para segmentar seus clientes em vários grupos?
8. Você enquadraria o problema da detecção de spam como um problema de aprendizado supervisionado ou sem supervisão?
9. O que é um sistema de aprendizado online?
10. O que é o *out-of-core learning*?
11. Que tipo de algoritmo de aprendizado depende de uma medida de similaridade para fazer previsões?
12. Qual a diferença entre um parâmetro de modelo e o hiperparâmetro do algoritmo de aprendizado?
13. O que os algoritmos de aprendizado baseados em modelos procuram? Qual é a estratégia mais comum que eles utilizam para ter sucesso? Como fazem previsões?
14. Você pode citar quatro dos principais desafios no Aprendizado de Máquina?
15. Se o seu modelo é ótimo nos dados de treinamento, mas generaliza mal para novas instâncias, o que está acontecendo? Você pode nomear três soluções possíveis?
16. O que é um conjunto de testes e por que você o utilizaria?
17. Qual é o propósito de um conjunto de validação?
18. O que pode dar errado se você ajustar os hiperparâmetros utilizando o conjunto de teste?
19. O que é validação cruzada e por que você preferiria usá-la ao invés de um conjunto de validação?

As soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 2

# Projeto de Aprendizado de Máquina de Ponta a Ponta

Neste capítulo você verá um exemplo de projeto de ponta a ponta na pele de um cientista de dados recentemente contratado por uma empresa do mercado imobiliário.<sup>1</sup> Você seguirá estes principais passos:

1. Olhar para o quadro geral;
2. Obter os dados;
3. Descobrir e visualizar os dados para obter informações;
4. Preparar os dados para os algoritmos do Aprendizado de Máquina;
5. Selecionar e treinar um modelo;
6. Ajustar o seu modelo;
7. Apresentar sua solução;
8. Lançar, monitorar e manter seu sistema.

## Trabalhando com Dados Reais

É melhor experimentar com dados do mundo real e não apenas conjuntos de dados artificiais ao estudar Aprendizado de Máquina. Felizmente, existem milhares de conjuntos de dados disponíveis a sua escolha, variando em todos os tipos de domínios. Você pode procurar em muitos lugares, tais quais:

- Reppositórios Populares de *open data*:
  - UC Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>)

<sup>1</sup> Este exemplo de projeto é completamente fictício. O objetivo é apenas ilustrar os níveis principais de um projeto de Aprendizado de Máquina, não aprender sobre o mercado imobiliário.

- Conjunto de dados no Kaggle (<https://www.kaggle.com/datasets>)
- Conjunto de Dados no AWS da Amazon (<http://aws.amazon.com/fr/datasets/>)
- Meta portais (eles listam repositórios *open data*):
  - <http://dataportals.org/>
  - <http://opendatamonitor.eu/>
  - <http://quandl.com/>
- Outras páginas que listam muitos repositórios populares de *open data*:
  - Lista de conjuntos de dados de Aprendizado de Máquina do Wikipedia (<https://goo.gl/SJHN2k>)
  - Pergunta no Quora.com (<http://goo.gl/zDR78y>)
  - Conjuntos de dados no Reddit (<https://www.reddit.com/r/datasets>)

Neste capítulo, escolhemos o conjunto de dados do repositório StatLib referente a preços do setor imobiliário na Califórnia<sup>2</sup> (veja a Figura 2-1). Este conjunto de dados foi baseado no censo de 1990 na Califórnia. Não são dados recentes (naquela época, ainda era possível comprar uma casa agradável perto da Baía de São Francisco), mas possuem muitas qualidades para o aprendizado, então vamos fingir que são dados recentes. Também adicionamos um atributo categórico e removemos algumas características para fins de ensino.

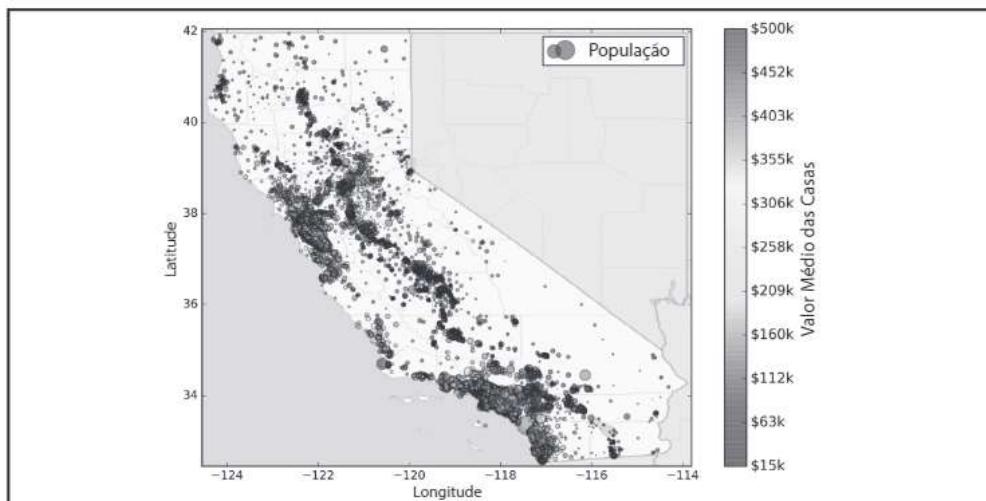


Figura 2-1. Preços de casas na Califórnia

<sup>2</sup> O conjunto de dados original figurou em R. Kelley Pace Ronald Barry, "Sparse Spatial Autoregressions", Statistics & Probability Letters 33, nº 3 (1997): 291–297.

## Um Olhar no Quadro Geral

Bem-vindo à Machine Learning Housing Corporation! A primeira tarefa que você executará será construir um modelo de preços do setor imobiliário utilizando os dados do censo da Califórnia. Esses dados têm métricas como população, renda média, preço médio do setor imobiliário e assim por diante para cada grupo de bairros. Os grupos de bairros são a menor unidade geográfica para a qual o *US Census Bureau* publica dados de amostra (um grupo de bairros geralmente tem uma população de 600 a 3 mil pessoas). Para abreviar, os chamaremos de “bairros”.

Seu modelo deve aprender com esses dados e ser capaz de prever o preço médio em qualquer bairro, considerando todas as outras métricas.



Como um cientista de dados bem organizado, a primeira coisa a fazer é obter sua lista de verificação do projeto. Você pode começar com a lista do Apêndice B, que deve funcionar razoavelmente bem para a maioria dos projetos de Aprendizado de Máquina, mas assegure-se de adaptá-la às suas necessidades. Neste capítulo, passaremos por muitos itens da lista de verificação, mas também ignoraremos alguns porque eles são autoexplicativos ou porque serão discutidos em capítulos posteriores.

## Enquadre o Problema

A primeira pergunta a ser feita ao seu chefe é: qual é exatamente o objetivo comercial? Provavelmente construir um modelo não será o objetivo final. Como a empresa espera usar e se beneficiar deste modelo? Isso é importante porque determinará como você enquadra o problema, quais algoritmos selecionará, que medida de desempenho utilizará para avaliar seu modelo e quanto esforço você deve colocar nos ajustes.

Seu chefe responde que o resultado do seu modelo (uma previsão do preço médio do setor imobiliário no bairro) será enviado para outro sistema de Aprendizado de Máquina (veja a Figura 2-2), juntamente com muitos outros *sinais*.<sup>3</sup> Esse sistema de downstream determinará se vale a pena investir em uma determinada área ou não. É fundamental acertar nessa etapa, uma vez que afetará diretamente a receita.

---

<sup>3</sup> Dados fornecidos para um sistema de Aprendizado de Máquina são chamados de sinais em referência à teoria da informação de Shannon: você quer que a proporção sinal/ruído seja alta.

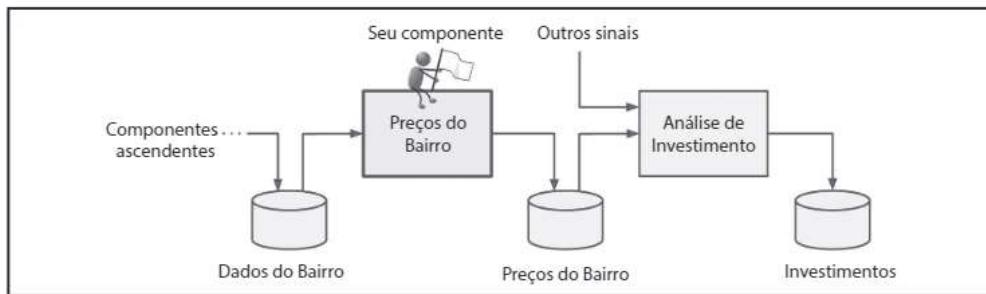


Figura 2-2. Um pipeline de Aprendizado de Máquina para investimentos imobiliários

## Pipelines

Uma sequência de *componentes* de processamento de dados é chamada de *pipeline* de dados. Os pipelines são muito comuns em sistemas do Aprendizado de Máquina, uma vez que existem muitos dados para manipular e muitas transformações para aplicar neles.

Os componentes normalmente rodam de forma assíncrona. Cada componente puxa uma grande quantidade de dados, os processa e envia o resultado para outro armazenador dados, e então, algum tempo depois, o próximo componente no pipeline puxa esses dados e envia sua própria saída, e assim por diante. Cada componente é bastante autônomo: a interface entre os componentes é simplesmente o armazenamento de dados. Isso faz com que entender o sistema seja bastante simples (com a ajuda de um grafo do fluxo de dados), e diferentes equipes podem se concentrar em diferentes componentes. Além disso, se um componente se romper, os componentes de downstream geralmente continuam a funcionar normalmente (pelo menos por um tempo), utilizando apenas a última saída do componente rompido. Isso torna a arquitetura bastante robusta.

Por outro lado, se um monitoramento adequado não for implementado, um componente rompido pode passar despercebido por algum tempo. Os dados ficarão obsoletos e o desempenho geral do sistema decairá.

A próxima pergunta a ser feita é: qual é a solução no momento (se houver)? Muitas vezes a resposta lhe dará um desempenho de referência, bem como informações sobre como resolver o problema. Seu chefe responde que os preços das casas nos bairros são estimados manualmente por especialistas: uma equipe reúne informações atualizadas sobre um bairro e, quando não conseguem obter o preço médio, o estimam utilizando regras complexas.

Este processo é caro e demorado, e suas estimativas não são boas; nos casos em que conseguem descobrir o preço médio real do setor imobiliário, geralmente percebem que suas estimativas ficaram mais de 10% abaixo do valor. É por isso que a empresa acha útil treinar um modelo a partir de outros dados sobre esse bairro para prever o preço médio do setor imobiliário. Os dados do recenseamento

parecem um ótimo conjunto de dados para serem explorados com este propósito, pois incluem os preços médios de milhares de bairros, bem como outros dados.

Bem, com toda essa informação você já está pronto para começar a projetar seu sistema. Primeiro, você precisa enquadrar o problema: será supervisionado, sem supervisão ou um Aprendizado por Reforço? Será uma tarefa de classificação, de regressão ou outra coisa? Você deve utilizar técnicas de aprendizado em lote ou online? Antes de ler, pause e tente responder para si estas perguntas.

Você encontrou as respostas? Vamos ver: claramente temos uma tarefa típica de aprendizado supervisionado, uma vez que você recebe exemplos *rotulados* de treinamento (cada instância vem com o resultado esperado, ou seja, o preço médio do setor imobiliário do bairro). Além disso, também é uma tarefa típica de regressão, já que você é solicitado a prever um valor. Mais especificamente, trata-se de um problema de *regressão multivariada*, uma vez que o sistema utilizará múltiplas características para fazer uma previsão (ele usará a população do bairro, a renda média, etc.). No primeiro capítulo, você previu níveis de satisfação de vida com base em apenas uma característica, o PIB per capita, de modo que era um problema de *regressão univariante*. Finalmente, não há um fluxo contínuo de dados entrando no sistema, não há uma necessidade em especial de se ajustar rapidamente à mudança de dados, e os dados são pequenos o suficiente para caber na memória, de modo que o aprendizado simples em lote deve funcionar bem.



Se os dados fossem enormes, você poderia dividir seu trabalho de aprendizado em lotes em vários servidores (utilizando a técnica *MapReduce*, como veremos mais adiante), ou poderia utilizar uma técnica de *aprendizado online*.

## Selecionar uma Medida de Desempenho

Seu próximo passo é selecionar uma medida de desempenho. Uma medida típica de desempenho para problemas de regressão é a *Raiz do Erro Quadrático Médio* (sigla RMSE, em inglês). Ela dá uma ideia da quantidade de erros gerados pelo sistema em suas previsões, com um peso maior para grandes erros. A Equação 2-1 mostra a fórmula matemática para calcular a RMSE.

Equação 2-1. Raiz do Erro Quadrático Médio (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

## Notações

Esta equação apresenta vários apontamentos muito comuns do Aprendizado de Máquina que usaremos ao longo deste livro:

- $m$  é o número de instâncias no conjunto de dados no qual você está medindo o RMSE.
  - Por exemplo, se estiver avaliando o RMSE em um conjunto de validação de 2 mil bairros, então  $m = 2$  mil.
- $\mathbf{x}^{(i)}$  é um vetor de todos os valores da característica (excluindo o *rótulo*) da  $i$ -ésima instância no conjunto de dados, e  $y^{(i)}$  é seu *rótulo* (o valor desejado da saída para aquela instância).
  - Por exemplo, se o primeiro bairro no conjunto de dados estiver localizado na longitude  $-118,29^\circ$ , latitude  $33,91^\circ$ , e ele tem 1.416 habitantes com uma renda média de US\$ 38.372, e o valor médio da casa é de US\$ 156.400 (ignorando as outras características por ora), então,

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118,29 \\ 33,91 \\ 1416 \\ 38372 \end{pmatrix}$$

e

$$y^{(1)} = 156.400$$

- $\mathbf{X}$  é uma matriz que contém todos os valores da característica (excluindo rótulos) de todas as instâncias no conjunto de dados. Existe uma linha por instância e a  $i$ -ésima linha é igual a transposição de  $\mathbf{x}^{(i)}$ , notado por  $(\mathbf{x}^{(i)})^T$ .<sup>4</sup>
  - Por exemplo, se o primeiro bairro for conforme descrito, então a matriz  $\mathbf{X}$  fica assim:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118,29 & 33,91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- $h$  é a função de previsão do seu sistema, também chamada de *hipótese*. Quando seu sistema recebe o vetor  $\mathbf{x}^{(i)}$  da característica de uma instância, ele exibe um valor previsto  $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$  para aquela instância ( $\hat{y}$  se pronuncia “y-chapéu”).

<sup>4</sup> Lembre-se de que o operador de transposição vira um vetor de coluna em um vetor de linha (e vice-versa).

- Por exemplo, se o seu sistema prevê que o preço médio no primeiro bairro é de US\$ 158.400, então  $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158.400$ . A previsão de erro para este bairro é  $\hat{y}^{(1)} - y^{(1)} = 2.000$ .
- RMSE( $\mathbf{X}, h$ ) é a função de custo medida no conjunto de exemplos utilizando sua hipótese  $h$ .

Usamos a fonte em itálico e caixa baixa para valores escalares (como  $m$  ou  $y^{(i)}$ ) e nomes de funções (como  $h$ ), fonte em negrito e caixa baixa para vetores (como  $\mathbf{x}^{(i)}$ ) e em negrito e caixa alta para matrizes (como  $\mathbf{X}$ ).

Embora a RMSE seja geralmente a medida de desempenho preferencial para tarefas de regressão, em alguns contextos você pode preferir utilizar outra função. Por exemplo, suponha que existam vários bairros outliers. Nesse caso, você pode considerar usar o *Erro Médio Absoluto* (sigla MAE, em inglês) (também chamado de Desvio Médio Absoluto; ver Equação 2-2):

#### *Equação 2-2. Erro Médio Absoluto*

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

Tanto a RMSE quanto o MAE são formas de medir a distância entre dois vetores: o vetor das previsões e o vetor dos valores alvo. Várias medidas de distância, ou *normas*, são possíveis:

- Calcular a raiz de uma soma de quadrados (RMSE) corresponde à *norma euclidiana*: é a noção de distância que você conhece. Também chamado de *norma  $\ell_2$* , notado por  $\|\cdot\|_2$  (ou apenas  $\|\cdot\|$ ).
- Calcular a soma de absolutos (MAE) corresponde à *norma  $\ell_1$* , notado por  $\|\cdot\|_1$ . Às vezes é chamado de *norma Manhattan* porque mede a distância entre dois pontos em uma cidade, se você só puder viajar ao longo de bairros ortogonais da cidade.
- Mais genericamente, a norma  $\ell_k$  de um vetor  $\mathbf{v}$  que contém  $n$  elementos é definida como  $\|\mathbf{v}\|_k = (\|v_0\|^k + \|v_1\|^k + \dots + \|v_n\|^k)^{\frac{1}{k}}$ .  $\ell_0$  apenas dá o número de elementos diferentes de zero no vetor, e  $\ell_\infty$  dá o valor máximo absoluto no vetor.
- Quanto maior o índice da norma, mais ela se concentra em grandes valores e negligencia os pequenos. É por isso que a RMSE é mais sensível a outliers do que o MAE. Mas, quando os outliers são exponencialmente raros (como em uma curva em forma de sino), a RMSE funciona muito bem, e geralmente é a preferida.

## Verifique as Hipóteses

Por último, uma boa prática é listar e verificar as hipóteses que foram feitas até agora (por você ou outros); fica mais fácil pegar problemas sérios logo no início. Por exemplo, os preços dos bairros mostrados pelo seu sistema serão alimentados em um sistema downstream do Aprendizado de Máquina, e assumiremos que esses preços serão usados como tal. Mas, e se o sistema downstream converter os preços em categorias (por exemplo, “barato”, “médio” ou “caro”) e utilizar essas categorias em vez dos próprios preços? Neste caso, não é importante saber o preço exato se o seu sistema só precisa obter a categoria certa. Se for assim, então o problema deveria ter sido enquadrado como uma tarefa de classificação, não uma tarefa de regressão. Você não vai querer descobrir isso após trabalhar por meses em um sistema de regressão.

Felizmente, depois de conversar com a equipe responsável pelo sistema downstream, você está confiante de que realmente precisa dos preços reais e não apenas das categorias. Ótimo! Você está pronto, as luzes estão verdes e você pode começar a programar agora!

## Obtenha os Dados

É hora de colocar as mãos na massa. Não hesite em pegar seu laptop e acompanhar os seguintes exemplos de código em um notebook do Jupyter, que está disponível na íntegra em <https://github.com/ageron/handson-ml>.

## Crie o Espaço de Trabalho

Primeiro, você precisará ter o Python instalado. Provavelmente ele já está instalado em seu sistema. Caso contrário, você pode baixá-lo em <https://www.python.org/>.<sup>5</sup>

Em seguida, crie um diretório de trabalho para seu código e conjuntos de dados do Aprendizado de Máquina. Abra um terminal e digite os seguintes comandos (após o prompts \$):

```
$ export ML_PATH="$HOME/ml"      # Mude o caminho se preferir  
$ mkdir -p $ML_PATH
```

Você precisará de vários módulos Python: Jupyter, NumPy, Pandas, Matplotlib e Scikit-Learn. Se você já tem o Jupyter rodando com todos esses módulos instalados, pode pular com segurança para “Baixe os dados” na página 45. Se você ainda não tem, há várias

<sup>5</sup> A versão mais recente do Python 3 é recomendada. O Python 2.7+ também funcionará bem, mas está obsoleto. Se usar o Python 2, você deve adicionar `from_fUTURE_import division, print_function, unicode_literals` no início do seu código.

maneiras de instalá-los (e suas dependências). Você pode utilizar o sistema de empacotamento do seu sistema (por exemplo, apt-get no Ubuntu, ou MacPorts ou HomeBrew no macOS), instale uma distribuição científica do Python, como o Anaconda, e utilize seu sistema de empacotamento, ou utilize o sistema de empacotamento do Python, pip, que está incluído por padrão com os instaladores binários Python (desde a versão Python 2.7.9).<sup>6</sup> Você pode verificar se o pip está instalado digitando o seguinte comando:

```
$ pip3 --version
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

Você deve se certificar de que tenha uma versão recente do pip instalada, pelo menos > 1.4 para suportar a instalação do módulo binário (também conhecida como wheels). Para atualizar o módulo do pip, digite:<sup>7</sup>

```
$ pip3 install --upgrade pip
Collecting pip
[...]
Successfully installed pip-9.0.1
```

## Criando um Ambiente Isolado

Se você quiser trabalhar em um ambiente isolado (o que é fortemente recomendado para que se possa trabalhar em projetos diferentes sem ter versões conflitantes de biblioteca), instale o virtualenv executando o seguinte comando pip:

```
$ pip3 install --user --upgrade virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Agora você pode criar um ambiente Python isolado digitando:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

Agora, sempre que quiser ativar esse ambiente, basta abrir um terminal e digitar:

```
$ cd $ML_PATH
$ source env/bin/activate
```

<sup>6</sup> Mostraremos a seguir os passos para a instalação usando pip em uma *bash shell* no Linux ou no macOS. Talvez seja preciso adaptar estes comandos ao seu próprio sistema. Para o Windows, recomendamos a instalação do Anaconda.

<sup>7</sup> Talvez seja necessário ter direitos de administrador para usar este comando; se for o caso, tente prefixá-lo com *sudo*.

Qualquer pacote que você instalar utilizando pip será instalado neste ambiente isolado enquanto o ambiente estiver ativo, e o Python somente terá acesso a esses pacotes (se você também quiser acesso aos pacotes do sistema no site, deverá criar o ambiente utilizando a opção `--system-site-packages` do virtualenv). Confira a documentação do virtualenv para obter mais informações.

Agora você pode instalar todos os módulos necessários e suas dependências utilizando este simples comando pip (se você não estiver utilizando um virtualenv, precisará de direitos de administrador ou adicionar a opção `--user`):

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

Para verificar sua instalação, tente importar todos os módulos assim:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

Não deve haver nenhuma saída e nenhum erro. Agora você pode iniciar o Jupyter digitando:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut down all
kernels (twice to skip confirmation).
```

Um servidor Jupyter está sendo executado agora pela porta 8888 em seu terminal. Você pode visitar este servidor abrindo seu navegador em `http://localhost:8888/` (isso geralmente acontece automaticamente quando o servidor é iniciado). Você deve ver seu diretório de trabalho vazio (contendo apenas o diretório `env` se você seguiu as instruções anteriores do virtualenv).

Agora, crie um novo notebook Python clicando no botão New e selecionando a versão adequada do Python<sup>8</sup> (veja a Figura 2-3).

Isto faz três coisas: primeiro, cria um novo arquivo notebook chamado `Untitled.ipynb` em seu espaço de trabalho; segundo, inicia um *Jupyter Python kernel* para rodar este *notebook*; e, terceiro, abre este notebook em uma nova guia. Você deve começar clicando em Untitled e digitando o novo nome, renomeando este notebook para “Housing” (isso mudará o arquivo automaticamente para `Housing.ipynb`).

<sup>8</sup> Note que o Jupyter pode lidar com várias versões do Python, e até muitas outras linguagens, como R ou Octave.

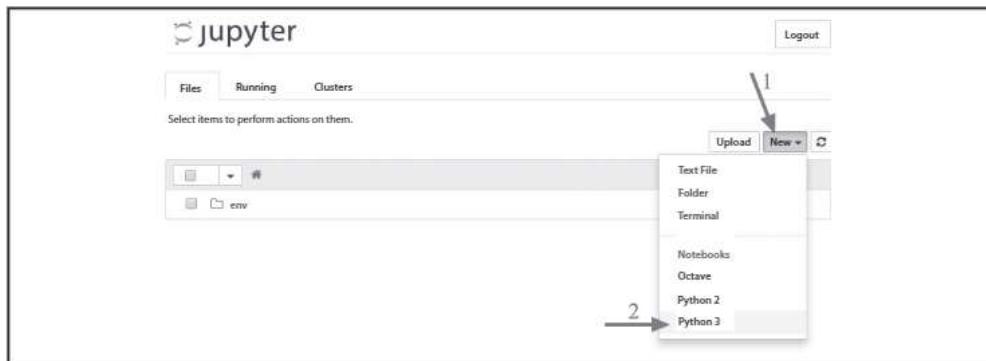


Figura 2-3. Seu espaço de trabalho no Jupyter

Um notebook contém uma lista de células. Cada célula pode conter código executável ou texto formatado. No momento, o notebook contém apenas uma célula de código vazio, rotulada “In [1]”. Tente digitar `print("Hello world!")` na célula, e clique no botão Reproduzir (veja a Figura 2-4) ou pressione Shift-Enter. Isso envia a célula atual para o *kernel* Python deste notebook, que o executa e retorna a saída. O resultado é exibido abaixo da célula, e, já que chegamos ao final do notebook, uma nova célula é automaticamente criada. Confira o User Interface Tour no menu de Ajuda do Jupyter para aprender o básico.

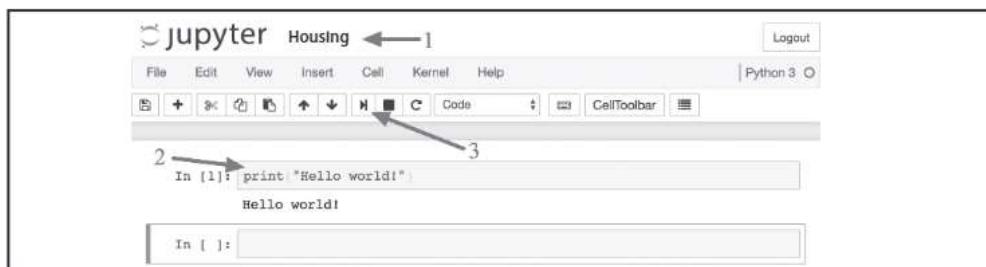


Figura 2-4. Notebook Python Hello world

## Baixe os Dados

Em ambientes típicos, seus dados estariam disponíveis em um banco de dados relacional (ou algum outro armazenamento comum de dados) e espalhados por várias tabelas/ documentos/arquivos. Para acessá-lo, primeiro você precisaria obter suas credenciais e autorizações<sup>9</sup> de acesso e familiarizar-se com o esquema de dados. Neste projeto, no entanto, as coisas são muito mais simples: você só vai baixar um único arquivo compactado,

<sup>9</sup> Você também pode precisar verificar restrições legais, como campos privados que nunca devem ser copiados para armazenamentos inseguros de dados.

*housing.tgz*, que contém um arquivo com valores separados por vírgulas (CSV) chamado *housing.csv* com todos os dados.

Você poderia usar o navegador da Web para baixá-lo e executar o `tar xzf housing.tgz` para descompactá-lo e extrair o arquivo CSV, mas é preferível criar uma pequena função para fazer isso. Ela será útil, principalmente, se os dados mudarem regularmente, pois permite que você escreva um pequeno script que poderá ser executado sempre que precisar para buscar os dados mais recentes (ou poderá configurar um trabalho agendado para fazer isso automaticamente em intervalos regulares). Também será útil automatizar o processo de busca caso você precise instalar o conjunto de dados em várias máquinas.

Esta é a função para buscar os dados:<sup>10</sup>

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")

    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Agora, quando chamamos `fetch_housing_data()`, ele cria um diretório *datasets/housing* no seu espaço de trabalho, baixa o arquivo *housing.tgz* e extraí o *housing.csv* neste diretório.

Agora, vamos carregar os dados com o Pandas. Mais uma vez você deve escrever uma pequena função para carregá-los:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

Esta função retorna um objeto *DataFrame Pandas* contendo todos os dados.

<sup>10</sup> Em um projeto real, você salvaria este código em um arquivo Python, mas, por enquanto, você pode apenas escrevê-lo no seu notebook do Jupyter.

## Uma Rápida Olhada na Estrutura dos Dados

Utilizando o método `head()` do DataFrame, vejamos as cinco linhas superiores (veja Figura 2-5).

In [5]:	<code>housing = load_housing_data() housing.head()</code>																																										
Out[5]:	<table border="1"> <thead> <tr> <th></th><th>longitude</th><th>latitude</th><th>housing_median_age</th><th>total_rooms</th><th>total_bedrooms</th><th>population</th></tr> </thead> <tbody> <tr> <td>0</td><td>-122.23</td><td>37.88</td><td>41.0</td><td>880.0</td><td>129.0</td><td>322.0</td></tr> <tr> <td>1</td><td>-122.22</td><td>37.86</td><td>21.0</td><td>7099.0</td><td>1106.0</td><td>2401.0</td></tr> <tr> <td>2</td><td>-122.24</td><td>37.85</td><td>52.0</td><td>1467.0</td><td>190.0</td><td>496.0</td></tr> <tr> <td>3</td><td>-122.25</td><td>37.85</td><td>52.0</td><td>1274.0</td><td>235.0</td><td>558.0</td></tr> <tr> <td>4</td><td>-122.25</td><td>37.85</td><td>52.0</td><td>1627.0</td><td>280.0</td><td>565.0</td></tr> </tbody> </table>		longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	0	-122.23	37.88	41.0	880.0	129.0	322.0	1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	2	-122.24	37.85	52.0	1467.0	190.0	496.0	3	-122.25	37.85	52.0	1274.0	235.0	558.0	4	-122.25	37.85	52.0	1627.0	280.0	565.0
	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population																																					
0	-122.23	37.88	41.0	880.0	129.0	322.0																																					
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0																																					
2	-122.24	37.85	52.0	1467.0	190.0	496.0																																					
3	-122.25	37.85	52.0	1274.0	235.0	558.0																																					
4	-122.25	37.85	52.0	1627.0	280.0	565.0																																					

Figura 2-5. As cinco linhas superiores no conjunto de dados

Cada linha representa um bairro. Existem 10 atributos (você pode ver os primeiros seis na captura de tela): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` e `ocean_proximity`.

O método `info()` é útil para a obtenção de uma rápida descrição dos dados, em especial o número total de linhas, o tipo de cada atributo e o número de valores não nulos (veja a Figura 2-6).

In [6]:	<code>housing.info()</code>
	<pre>&lt;class 'pandas.core.frame.DataFrame'&gt; RangeIndex: 20640 entries, 0 to 20639 Data columns (total 10 columns): longitude          20640 non-null float64 latitude           20640 non-null float64 housing_median_age 20640 non-null float64 total_rooms         20640 non-null float64 total_bedrooms      20433 non-null float64 population         20640 non-null float64 households          20640 non-null float64 median_income       20640 non-null float64 median_house_value  20640 non-null float64 ocean_proximity     20640 non-null object dtypes: float64(9), object(1) memory usage: 1.6+ MB</pre>

Figura 2-6. Informações das Casas

Existem 20.640 instâncias no conjunto de dados, o que significa que é muito pequeno para os padrões de Aprendizado de Máquina, mas é perfeito para começar. Repare que o atributo `total_bedrooms` tem apenas 20.433 valores não nulos, significando que 207 bairros não possuem esta característica. Cuidaremos disso mais tarde.

Todos os atributos são numéricos, exceto o campo `ocean_proximity`. Seu tipo é `object`, então ele poderia conter qualquer tipo de objeto Python, mas, como você carregou esses dados de um arquivo CSV, você sabe que ele deve ser um atributo de texto. Quando você olhou para as cinco primeiras linhas, provavelmente percebeu que os valores na coluna `ocean_proximity` eram repetitivos, o que significa que é provavelmente um atributo categórico. Você pode descobrir quais categorias existem e quantos bairros pertencem a cada categoria utilizando o método `value_counts()`:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: ocean_proximity, dtype: int64
```

Vamos olhar para outros campos. O método `describe()` mostra um resumo dos atributos numéricos (Figura 2-7).

In [8]:	housing.describe()					
Out[8]:		longitude	latitude	housing_median_age	total_rooms	total_bedrooms
	<code>count</code>	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
	<code>mean</code>	-119.569704	35.631861	28.639486	2635.763081	537.870553
	<code>std</code>	2.003532	2.135952	12.585558	2181.615252	421.385077
	<code>min</code>	-124.350000	32.540000	1.000000	2.000000	1.000000
	<code>25%</code>	-121.800000	33.930000	18.000000	1447.750000	296.000000
	<code>50%</code>	-118.490000	34.260000	29.000000	2127.000000	435.000000
	<code>75%</code>	-118.010000	37.710000	37.000000	3148.000000	647.000000
	<code>max</code>	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Figura 2-7. Resumo de cada atributo numérico

As linhas `count`, `mean`, `min` e `max` são autoexplicativas. Observe que os valores nulos são ignorados (então, por exemplo, `count` do `total_bedrooms` é 20.433, não 20.640). A linha `std` mostra o *desvio padrão*, que mede a dispersão dos valores.<sup>11</sup> As linhas `25%`, `50%` e `75%` mostram os *percentis* correspondentes: um percentil indica o valor abaixo no qual uma dada porcentagem cai em um grupo de observações. Por exemplo, 25% dos bairros tem uma `housing_median_age` menor que 18, enquanto 50% é menor que 29, e 75% é menor que 37. Estes são frequentemente chamados de 25º percentil (ou 1º quartil), a média e o 75º percentil (ou 3º quartil).

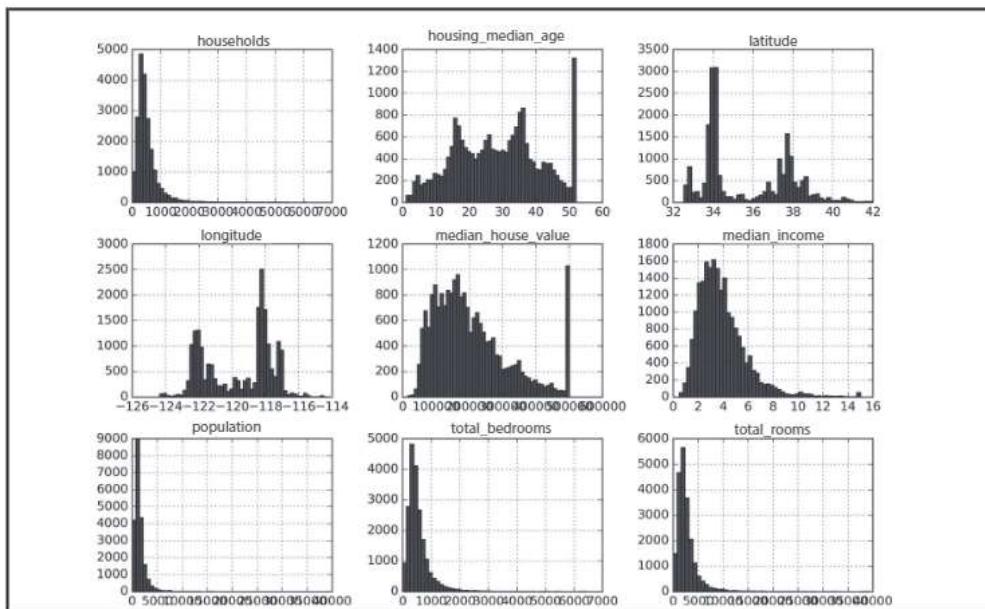
<sup>11</sup> O desvio padrão é geralmente denotado  $\sigma$  (da letra grega sigma) e é a raiz quadrada da *variância*, que é a média do desvio quadrado da média. Quando uma característica tem uma *distribuição normal* em forma de sino (também chamada de *distribuição Gaussiana*), que é muito comum, aplica-se a regra “68-95-99,7”: cerca de 68% dos valores se enquadram em 1 $\sigma$  da média, 95% dentro de 2 $\sigma$  e 99,7% dentro de 3 $\sigma$ .

Outro método rápido de perceber o tipo de dados com o qual você está lidando é traçar um histograma para cada atributo numérico. Um histograma mostra o número de instâncias (no eixo vertical) que possuem um determinado intervalo de valores (no eixo horizontal). Você pode traçar esse único atributo por vez ou pode chamar o método `hist()` em todo o conjunto de dados e traçar um histograma para cada atributo numérico (veja a Figura 2-8). Por exemplo, você pode ver que pouco mais de 800 bairros possuem um `median_house_value` equivalente a mais ou menos US\$ 100 mil.

```
%matplotlib inline # somente no notebook Jupyter
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



O método `hist()` depende do Matplotlib, que por sua vez depende de um backend gráfico especificado pelo usuário para figurar em sua tela. Então, antes que você consiga plotar qualquer coisa, é preciso especificar qual backend o Matplotlib deve usar. A opção mais simples é usar o comando mágico do Jupyter `%matplotlib inline`. Isso leva o Jupyter a configurar o Matplotlib para que ele utilize seu próprio backend. As plotagens são então processadas dentro do próprio notebook. Observe que é opcional em um notebook do Jupyter chamar o `show()`, pois ele exibirá gráficos automaticamente sempre que uma célula for executada.



*Figura 2-8. Um histograma para cada atributo numérico*

Preste atenção em alguns pontos destes histogramas:

1. Primeiro, o atributo da renda média não parece estar expresso em dólares americanos (USD). Depois de verificar com a equipe de coleta de dados, você é informado que os dados foram dimensionados e limitados em 15 (na verdade 15,0001) para a média dos maiores rendimentos, e em 0,5 (na verdade 0,4999) para a média dos rendimentos mais baixos. É comum trabalhar com atributos pré-processados no Aprendizado de Máquina e isto não é necessariamente um problema, mas você deve tentar entender como os dados foram calculados.
2. A idade média e o valor médio da casa também foram limitados. Este último pode ser um problema sério, pois é seu atributo alvo (seus rótulos). Os algoritmos de Aprendizado de Máquina podem aprender que os preços nunca ultrapassam esse limite. Você precisa verificar com a equipe do seu cliente (a equipe que usará a saída do seu sistema) para ver se isso é ou não um problema. Se eles disserem que precisam de previsões precisas mesmo acima de US\$ 500 mil, então você terá duas opções:
  - a. Coletar rótulos adequados para os bairros cujos rótulos foram limitados.
  - b. Remover esses bairros do conjunto de treinamento (e também do conjunto de testes, já que seu sistema não deve ser responsabilizado por prever valores além de US\$ 500 mil).
3. Esses atributos têm escalas muito diferentes. Discutiremos isso mais adiante neste capítulo, quando explorarmos o escalonamento das características.
4. Finalmente, muitos histogramas têm um *rastro alongado*: eles se estendem muito mais à direita da média do que à esquerda. Isso pode dificultar a detecção de padrões em alguns algoritmos do Aprendizado de Máquina. Vamos tentar transformar esses atributos mais tarde para conseguir mais distribuições na forma de sino.

Espero que você tenha agora uma melhor compreensão do tipo de dados com os quais está lidando.



Espere! Antes de analisar ainda mais os dados, você precisa criar um conjunto de teste, colocá-lo de lado e nunca checá-lo.

## Crie um Conjunto de Testes

Pode parecer estranho colocar de lado, voluntariamente, uma parte dos dados nesta fase. Afinal, você só deu uma rápida olhada nos dados e certamente precisa aprender muito mais a respeito antes de decidir quais algoritmos usar, certo? Isso é verdade, mas o seu cérebro é um incrível sistema de detecção de padrões, o que significa que é altamente propenso ao *sobreajuste*: se você olhar para o conjunto de teste, pode tropeçar em algum padrão aparentemente interessante nos dados de teste que o levará a selecionar um tipo particular de modelo do Aprendizado de Máquina. Quando você estima o erro de generalização utilizando o conjunto de teste, sua estimativa será muito otimista e você lançará um sistema que não funcionará tão bem quanto o esperado. Isso é chamado de *data snooping bias*.

Criar um conjunto de testes é, teoricamente, bastante simples: basta escolher aleatoriamente algumas instâncias, geralmente 20% do conjunto de dados, e colocá-las de lado:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Você pode usar essa função assim:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

Bem, isso funciona, mas não é perfeito: ao executar o programa novamente, será gerado um conjunto diferente de testes! Ao longo do tempo, você (ou seus algoritmos do Aprendizado de Máquina) verá todo o conjunto de dados, o que deve ser evitado.

Uma solução seria salvar o conjunto de testes na primeira execução e depois carregá-lo em execuções subsequentes. Outra opção é definir a semente do gerador de números aleatórios (por exemplo, `np.random.seed(42)`)<sup>12</sup> antes de chamar a `np.random.permutation()`, de modo que ele sempre gere os mesmos índices embaralhados.

Mas ambas as soluções serão interrompidas na próxima vez que você buscar um conjunto de dados atualizado. Uma solução comum é utilizar o identificador de cada instância para decidir se ela deve ou não ir no conjunto de teste (supondo que as instâncias tenham um identificador único e imutável). Por exemplo, você pode calcular um hash do identi-

---

<sup>12</sup> Você verá com frequência pessoas colocarem a *random seed* em 42. Esse número não possui propriedade especial alguma além de ser A Resposta para a Vida, para o Universo e Tudo Mais.

fificador de cada instância, manter apenas o último byte do hash e colocar a instância no conjunto de teste se esse valor for menor ou igual a 51 (~20% de 256). Isso garante que o conjunto de teste permanecerá consistente em várias execuções, mesmo ao atualizar o conjunto de dados. O novo conjunto de testes conterá 20% das novas instâncias, mas não conterá nenhuma instância que já estivesse no conjunto de treinamento. Veja uma possível implementação:

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Infelizmente, o conjunto de dados do setor imobiliário não possui uma coluna de identificação. A solução mais simples é utilizar o índice da linha como ID:

```
housing_with_id = housing.reset_index() # adiciona uma coluna 'index'
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

Se você utilizar o índice de linha como um identificador exclusivo, precisa se certificar de que novos dados sejam anexados ao final do conjunto de dados e que nenhuma linha seja excluída. Se isso não for possível, tente utilizar características mais estáveis para criar um identificador exclusivo. Por exemplo, a latitude e a longitude de um bairro serão certamente estáveis por alguns milhões de anos, então você pode combiná-las em um ID dessa forma:<sup>13</sup>

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

O Scikit-Learn fornece algumas funções para dividir conjuntos de dados em vários subconjuntos de diversas maneiras. A função mais simples é `train_test_split`, que faz praticamente o mesmo que a função `split_train_test` definida anteriormente, mas com alguns recursos adicionais. Primeiro, temos um parâmetro `random_state` que permite que você defina a semente do gerador de números aleatórios como explicado anteriormente e, em segundo lugar, você pode passar múltiplos conjuntos de dados com um número idêntico de linhas, e ele os dividirá nos mesmos índices (isso é muito útil se, por exemplo, você tiver um *DataFrame* separado para os rótulos):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

---

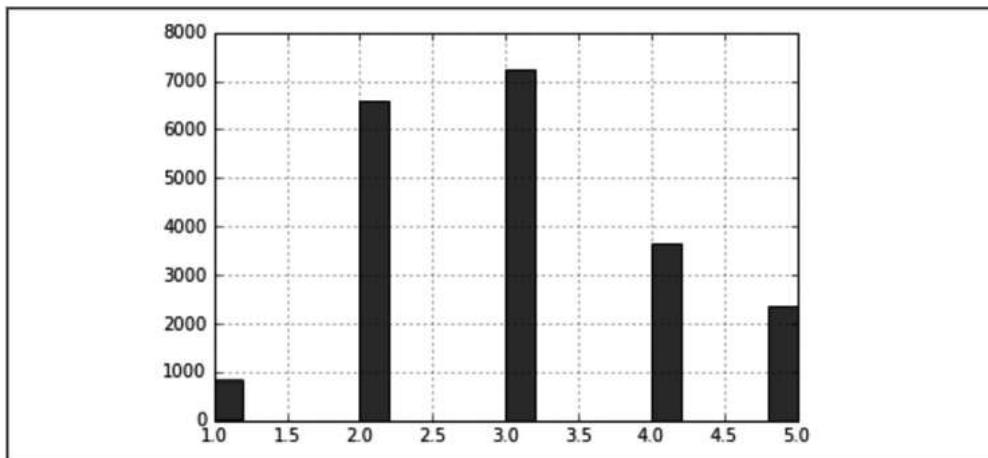
<sup>13</sup> A informação de localização é realmente bastante grosseira e, como resultado, muitos bairros terão exatamente a mesma identificação, acabando no mesmo conjunto (*test* ou *train*). Infelizmente, isso introduz algum viés de amostragem.

Até agora consideramos somente métodos de amostragem puramente aleatórios. Isso geralmente é bom se o seu conjunto de dados for suficientemente grande (especialmente em relação ao número de atributos), mas, se não for, corre o risco de apresentar um viés significativo de amostragem. Quando uma empresa de pesquisa decide ligar para mil pessoas e lhes fazer algumas perguntas, eles não escolhem aleatoriamente mil pessoas em uma lista telefônica. Eles tentam garantir que essas mil pessoas representem toda a população. Por exemplo, a população dos EUA é composta por 51,3% de pessoas do sexo feminino e 48,7% do sexo masculino, de modo que uma pesquisa bem conduzida tentaria manter essa proporção na amostragem: 513 mulheres e 487 homens. Isso é chamado de *amostragem estratificada*: a população é dividida em subgrupos homogêneos, chamados de *estratos*, e o número certo de instâncias de cada estrato é amostrado para garantir que o conjunto de testes seja representativo da população em geral. Se eles utilizassem amostragem puramente aleatória, haveria cerca de 12% de chance de amostrar um conjunto de teste distorcido, tanto com menos de 49% feminino quanto com mais de 54%. De qualquer forma, os resultados da pesquisa seriam significativamente tendenciosos.

Suponha que você tenha conversado com especialistas que lhe disseram que a renda média é um atributo muito importante para estimar os preços médios. Você quer garantir que o conjunto de testes seja representativo das várias categorias de rendimentos em todo o conjunto de dados. Uma vez que a renda média é um atributo numérico contínuo, primeiro você precisa criar um atributo na categoria da renda. Vejamos mais de perto o histograma da renda média (de volta à Figura 2-8): a maioria dos valores médios da renda está agrupada em torno de US\$ 20 mil–US\$ 50 mil mas alguns rendimentos médios ultrapassam os US\$ 60 mil. É importante ter um número suficiente de instâncias para cada estrato em seu conjunto de dados, ou então a estimativa da importância do estrato poderá ser tendenciosa. Isso significa que você não deve ter muitos estratos, e cada estrato deve ser grande o suficiente. O código a seguir cria um atributo da categoria da renda dividindo a renda média por 1,5 (para limitar o número de categorias da renda) e arredondando com a utilização do `ceil` (para ter categorias discretas) e, em seguida, mesclando todas as categorias maiores que 5, na categoria 5:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
```

Essas categorias de renda estão representadas na (Figura 2-9):



*Figura 2-9. Histograma das categorias de renda*

Agora você está pronto para fazer uma amostragem estratificada com base na categoria da renda. Para isso você pode utilizar a classe `StratifiedShuffleSplit` do Scikit-Learn:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Vamos ver se isso funcionou como esperado. Você pode começar pela análise das proporções da categoria de renda no conjunto de testes:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3.0    0.350533
2.0    0.318798
4.0    0.176357
5.0    0.114583
1.0    0.039729
Name: income_cat, dtype: float64
```

Com um código similar, você pode medir as proporções da categoria de renda no conjunto completo de dados. A Figura 2-10 compara as proporções da categoria de renda no conjunto geral de dados, no conjunto de teste gerado com a amostragem estratificada e em um conjunto de testes gerado a partir da amostragem puramente aleatória. Como você pode ver, o conjunto de testes gerado com a utilização da amostragem estratificada tem proporções da categoria de renda quase idênticas às do conjunto completo de dados, enquanto o conjunto de testes gerado com amostragem puramente aleatória é bastante distorcido.

	<b>Overall</b>	<b>Random</b>	<b>Stratified</b>	<b>Rand. %error</b>	<b>Strat. %error</b>
<b>1.0</b>	0.039826	0.040213	0.039738	0.973236	-0.219137
<b>2.0</b>	0.318847	0.324370	0.318876	1.732260	0.009032
<b>3.0</b>	0.350581	0.358527	0.350618	2.266446	0.010408
<b>4.0</b>	0.176308	0.167393	0.176399	-5.056334	0.051717
<b>5.0</b>	0.114438	0.109496	0.114369	-4.318374	-0.060464

Figura 2-10. Comparação de viés de amostragem estratificada versus amostragem aleatória

Agora, você deve remover o atributo `income_cat` para que os dados voltem ao seu estado original:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Ficamos um tempo na geração de conjuntos de testes por uma boa razão: esta parte crítica é muitas vezes negligenciada em um projeto de Aprendizado de Máquina. Além disso, muitas dessas ideias serão úteis mais tarde quando discutirmos a validação cruzada. Agora, é hora de avançar para o próximo estágio: explorar os dados.

## Descubra e Visualize os Dados para Obter Informações

Até aqui você deu apenas uma rápida olhada para ter uma compreensão geral do tipo de dados que está manipulando. Agora, o objetivo é aprofundar-se um pouco mais.

Primeiro, certifique-se de colocar o teste de lado e apenas explorar o conjunto de treinamento. Além disso, se o conjunto de treinamento for muito grande, talvez seja melhor experimentar um conjunto de exploração para fazer manipulações fácil e rapidamente. No nosso caso, o conjunto é muito pequeno, então você pode trabalhar diretamente no conjunto completo. Criaremos uma cópia para que você possa treinar com ela sem prejudicar o conjunto de treinamento:

```
housing = strat_train_set.copy()
```

## Visualizando Dados Geográficos

Como existem informações geográficas (latitude e longitude), é uma boa ideia criar um diagrama de dispersão para visualizar os dados de todos os bairros (Figura 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

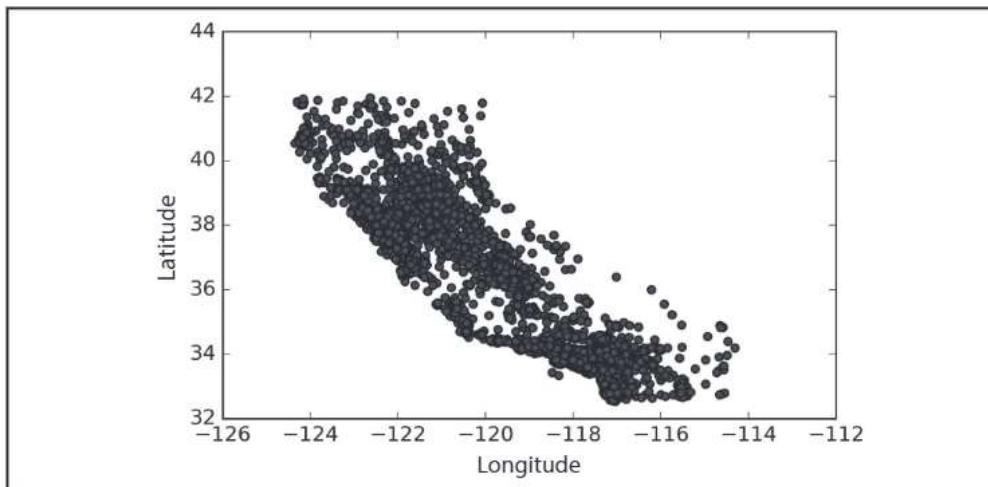


Figura 2-11. Um diagrama de dispersão geográfica dos dados

Isso se parece com a Califórnia, mas, além disso, é difícil ver qualquer padrão específico. Definir a opção `alpha` em `0,1` facilita a visualização dos locais onde existe uma alta densidade de pontos de dados (Figura 2-12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

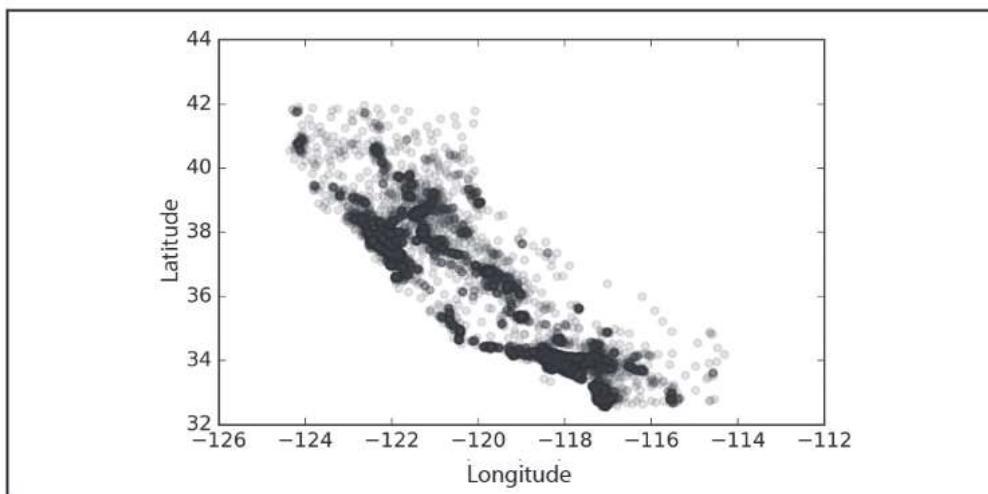


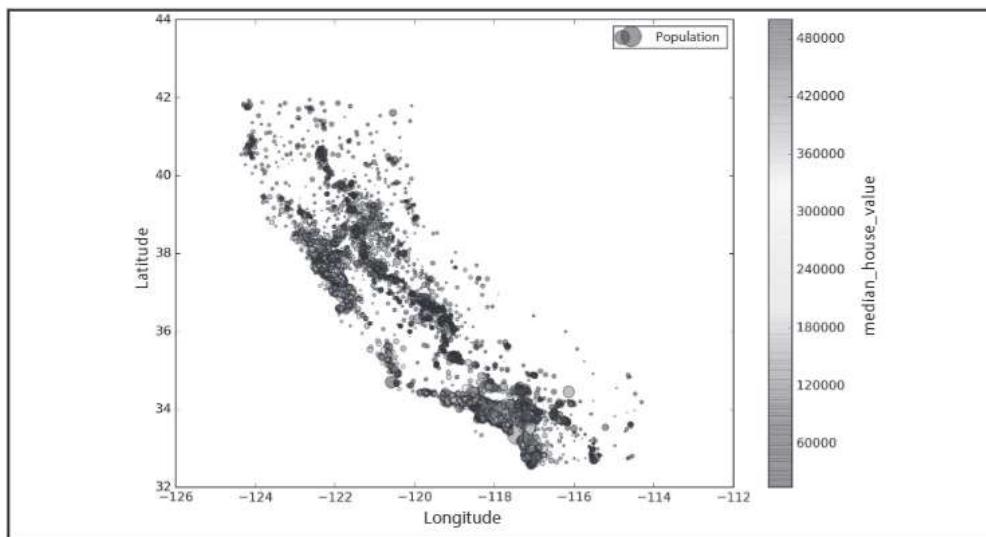
Figura 2-12. Uma melhor visualização destacando áreas de alta densidade

Agora já está melhor: é possível ver claramente as áreas de alta densidade, especificamente a Área da Baía de Los Angeles e San Diego, além de uma longa linha de alta densidade no Vale Central, principalmente ao redor de Sacramento e Fresno.

No geral, nosso cérebro é muito bom em detectar padrões em imagens, mas talvez seja necessário brincar com parâmetros de visualização para que os padrões se destaquem.

Agora, vejamos os preços do setor imobiliário (Figura 2-13). O raio de cada círculo representa a população do bairro (opção `s`) e a cor representa o preço (opção `c`). Usaremos um mapa de cores pré-definido (opção `cmap`) chamado `jet`, que varia do azul (valores baixos) para o vermelho (preços altos):<sup>14</sup>

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
             s=housing["population"]/100, label="population", figsize=(10,7),
             c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,
)
plt.legend()
```



*Figura 2-13. Preços das casas na Califórnia*

Esta imagem informa que os preços do setor imobiliário estão muito relacionados à localização (por exemplo, perto do oceano) e à densidade populacional, como provavelmente você já sabia. Será útil utilizar um algoritmo de agrupamento para detectar os grupos principais e adicionar novas características que medem a proximidade com os centros de agrupamento. O atributo de proximidade do oceano também pode ser útil, embora na costa norte da Califórnia os preços não sejam muito altos, então essa não é uma regra simples.

---

14 Se você estiver lendo em escala de cinza, pegue uma caneta vermelha e rabisque na maior parte do litoral da Área da Baía até San Diego (como esperado). Você também pode acrescentar um trecho amarelo ao redor de Sacramento.

## Buscando Correlações

Uma vez que o conjunto de dados não é muito grande, você pode calcular facilmente o *coeficiente de correlação padrão* (também chamado *r de Pearson*) entre cada par de atributos utilizando o método `corr()`:

```
corr_matrix = housing.corr()
```

Agora, vejamos o quanto cada atributo se correlaciona com o valor médio da habitação:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age   0.114220
households            0.064702
total_bedrooms        0.047865
population             -0.026699
longitude              -0.047279
latitude                -0.142826
Name: median_house_value, dtype: float64
```

O coeficiente de correlação varia de -1 a 1. Quando está próximo de 1, significa que existe uma forte correlação positiva; por exemplo, o valor médio da habitação tende a aumentar quando a renda média aumenta. Quando o coeficiente está próximo de -1, significa que existe uma forte correlação negativa; é possível ver uma pequena correlação negativa entre a latitude e o valor médio da habitação (ou seja, os preços tendem a diminuir quando você vai para o norte). Finalmente, coeficientes próximos de zero significam que não há correlação linear. A Figura 2-14 mostra várias plotagens juntamente com o coeficiente de correlação entre seus eixos horizontal e vertical.

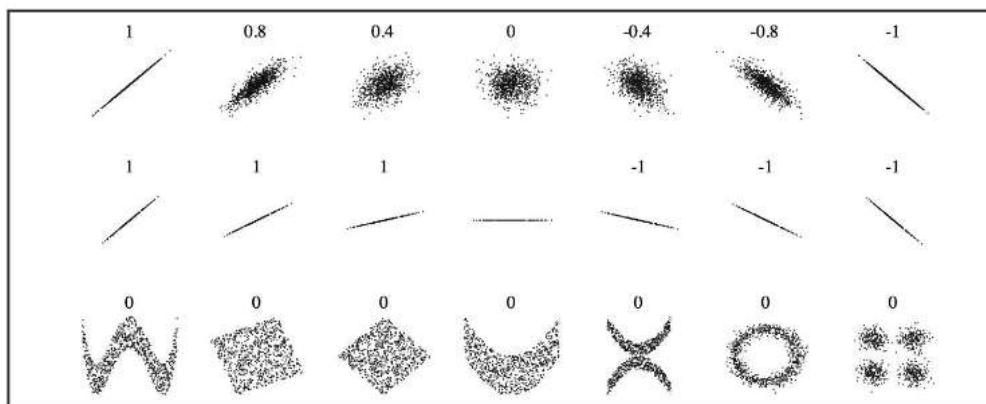


Figura 2-14. Coeficiente de correlação padrão de vários conjuntos de dados (fonte: Wikipedia; imagem de domínio público)



O coeficiente de correlação apenas mede correlações lineares (“se  $x$  sobe, então  $y$  geralmente sobe/desce”). Ele pode perder completamente as relações não lineares (por exemplo, “se  $x$  é próximo a zero, então,  $y$ , geralmente, sobe”). Observe como todas as plotagens da linha inferior têm um coeficiente de correlação igual a zero, apesar do fato de seus eixos claramente não serem independentes: são exemplos de relações não lineares. Além disso, a segunda linha mostra exemplos em que o coeficiente de correlação é igual a 1 ou -1; observe que isso não tem nada a ver com a inclinação. Por exemplo, sua altura em polegadas tem um coeficiente de correlação de 1 com sua altura em pés ou em nanômetros.

Outra maneira de verificar a correlação entre atributos é utilizar a função `scatter_matrix`, do Pandas, que plota cada atributo numérico em relação a qualquer outro atributo numérico. Uma vez que existem 11 atributos numéricos, você obteria  $11^2 = 121$  plotagens, o que não caberia em uma página, então focaremos apenas em alguns atributos promissores que parecem mais correlacionados com o valor médio do setor imobiliário (Figura 2-15):

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
               "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

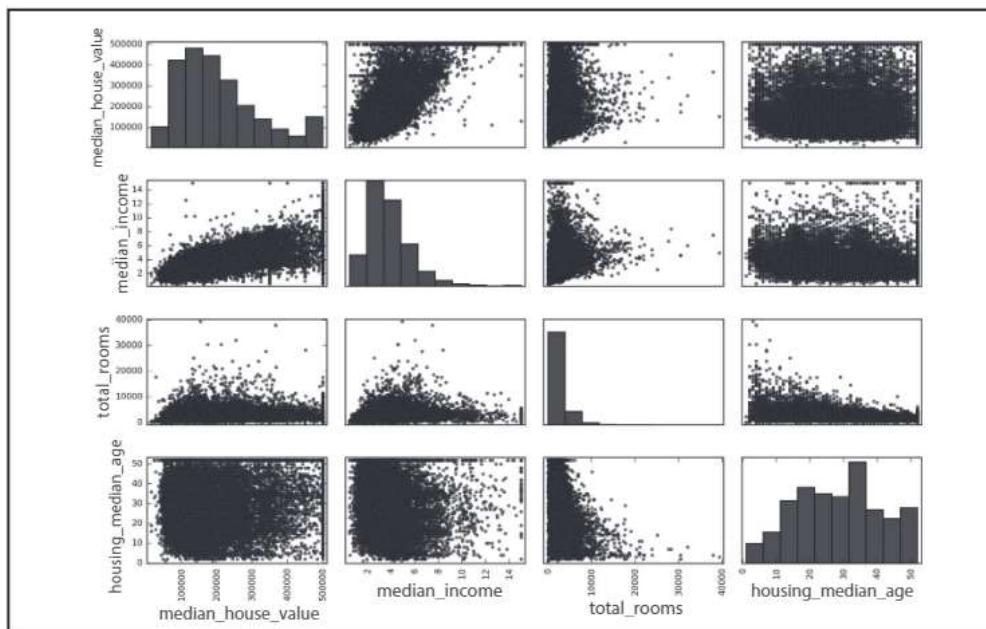


Figura 2-15. Matriz de dispersão

A diagonal principal (superior esquerda até a parte inferior direita) seria cheia de linhas retas se o Pandas plotasse cada variável em relação a si mesma, o que não seria muito útil. Então, em vez disso, o Pandas exibe um histograma para cada atributo (outras opções estão disponíveis, veja a documentação do Pandas para mais detalhes).

O atributo mais promissor para prever o valor médio da habitação é a renda média, então vamos observar o gráfico de dispersão de correlação (Figura 2-16):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",  
alpha=0.1)
```

Esta plotagem revela algumas coisas. Primeiro, a correlação é realmente muito forte; é possível ver claramente a tendência ascendente, e os pontos não estão muito dispersos. Em segundo lugar, o limite de preços que percebemos anteriormente é claramente visível como uma linha horizontal em US\$ 500 mil. Mas esta plotagem revela outras linhas retas menos óbvias: uma horizontal em torno de US\$ 450 mil, outra em torno de US\$ 350 mil, talvez uma em torno de US\$ 280 mil, e mais algumas abaixo disso. Você pode tentar remover os bairros correspondentes para evitar que seus algoritmos aprendam a reproduzir essas peculiaridades dos dados.

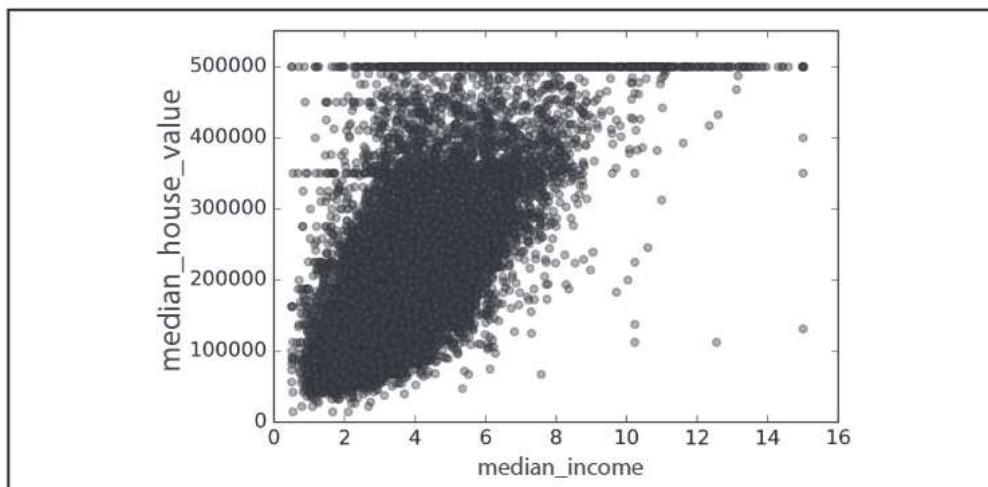


Figura 2-16. Renda média versus valor médio da habitação

## Experimentando com Combinações de Atributo

Esperamos que as seções anteriores tenham dado a você uma ideia de algumas maneiras de explorar os dados e obter informações. Você identificou algumas peculiaridades dos dados que talvez queira limpar antes de fornecê-los a um algoritmo de Aprendizado de Máquina, e também encontrou correlações interessantes entre atributos, especialmente

com o atributo-alvo. Também foi possível perceber que alguns atributos têm um rastro de distribuição prolongado, então pode ser que você queira transformá-los (por exemplo, ao calcular seu logaritmo). Claro, o seu raio de ação variará consideravelmente a cada projeto, mas as ideias gerais são semelhantes.

Uma última coisa que você pode querer fazer é tentar várias combinações de atributos antes de preparar os dados para os algoritmos de Aprendizado de Máquina. Por exemplo, o número total de cômodos em um bairro não terá muita utilidade se você não souber quantos domicílios existem. O que você realmente quer é o número de cômodos por domicílio. Da mesma forma, o número total de quartos por si só não é muito útil: você provavelmente vai querer compará-lo com o número de cômodos. E a população por domicílio também parece uma combinação de atributos interessante. Criaremos esses novos atributos:

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

E, agora, vejamos a matriz de correlação novamente:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income          0.687160
rooms_per_household    0.146285
total_rooms             0.135097
housing_median_age     0.114110
households              0.064506
total_bedrooms          0.047689
population_per_household -0.021985
population              -0.026920
longitude                -0.047432
latitude                  -0.142724
bedrooms_per_room       -0.259984
Name: median_house_value, dtype: float64
```

Ei, nada mal! O novo atributo `bedrooms_per_room` está muito mais correlacionado com o valor médio da habitação do que com o número total de cômodos ou quartos. Aparentemente, habitações com uma baixa relação quarto/cômodo tendem a ser mais caras. O número de cômodos por família também é mais informativo do que o número total de cômodos em um bairro — obviamente, quanto maiores as habitações, mais caras elas serão.

Esta rodada de exploração não precisa ser absolutamente minuciosa; o objetivo é começar com o pé direito e rapidamente obter informações que o ajudarão a produzir um primeiro protótipo razoavelmente bom. Mas este é um processo iterativo: uma vez que você obtiver um protótipo funcional, poderá analisar sua saída para adquirir mais informações e voltar a este passo da exploração.

## Prepare os Dados para Algoritmos do Aprendizado de Máquina

É hora de preparar os dados para seus algoritmos de Aprendizado de Máquina. Em vez de fazer isso manualmente, escreva funções, por vários bons motivos:

- Isso permitirá que você reproduza essas transformações facilmente em qualquer conjunto de dados (por exemplo, na próxima vez que você receber um novo conjunto de dados);
- Você gradualmente construirá uma biblioteca de funções de transformação que poderão ser reutilizadas em projetos futuros;
- Você pode usar essas funções em seu sistema ao vivo para transformar os novos dados antes de fornecê-lo aos seus algoritmos;
- Isso possibilitará que você tente várias transformações facilmente e veja qual combinação funciona melhor.

Mas primeiro vamos reverter para um conjunto de treinamento limpo (copiando `strat_train_set` mais uma vez), e vamos separar os previsores e os rótulos, uma vez que não queremos necessariamente aplicar as mesmas transformações às previsões e aos valores-alvo (observe que `drop()` cria uma cópia dos dados e não afeta `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

## Limpando os Dados

A maioria dos algoritmos de Aprendizado de Máquina não pode funcionar com características faltantes, então criaremos algumas funções para cuidar delas. Você notou anteriormente que o atributo `total_bedrooms` tem alguns valores faltantes, então vamos consertar isso. Há três opções:

- Livrar-se dos bairros correspondentes;
- Livrar-se de todo o atributo;
- Definir valores para algum valor (zero, a média, intermediária, etc.).

Você pode fazer isso facilmente utilizando os métodos `dropna()`, `drop()` e `fillna()` do `DataFrame`:

```
housing.dropna(subset=["total_bedrooms"])      # opção 1
housing.drop("total_bedrooms", axis=1)          # opção 2
median = housing["total_bedrooms"].median()     # opção 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Se você escolher a opção 3, deve calcular o valor médio no conjunto de treinamento e usá-lo para preencher os valores faltantes neste, mas não se esqueça de também salvar o valor médio que você calculou. Você precisará dele mais tarde para substituir os valores faltantes no conjunto de testes quando quiser avaliar seu sistema e também quando o sistema entrar em operação para substituir os valores faltantes nos novos dados.

O Scikit-Learn fornece uma classe acessível para cuidar dos valores faltantes: `Imputer`. Veja como utilizá-la. Primeiro você cria uma instância do `Imputer`, especificando que deseja substituir os valores faltantes de cada atributo pela média desse atributo:

```
from sklearn.preprocessing import Imputer  
  
imputer = Imputer(strategy="median")
```

Uma vez que a média só pode ser calculada em atributos numéricos, precisamos criar uma cópia dos dados sem o atributo de texto `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Agora, você pode ajustar a instância `imputer` aos dados de treinamento utilizando o método `fit()`:

```
imputer.fit(housing_num)
```

O `imputer` simplesmente calculou a média de cada atributo e armazenou o resultado em sua variável da instância `statistics_`. Somente o atributo `total_bedrooms` tinha valores faltantes, mas não podemos ter certeza de que não haverá valores faltantes nos novos dados após o sistema entrar em operação por isso é mais seguro aplicar o `imputer` a todos os atributos numéricos:

```
>>> imputer.statistics_  
array([-118.51, 34.26, 29., 2119.5, 433., 1164., 408., 3.5409])  
>>> housing_num.median().values  
array([-118.51, 34.26, 29., 2119.5, 433., 1164., 408., 3.5409])
```

Agora, você pode utilizar esse `imputer` “treinado” substituindo os valores perdidos pelas médias aprendidas a fim de transformar o conjunto de treinamento:

```
X = imputer.transform(housing_num)
```

O resultado será um array Numpy simples que contém as características transformadas. Se quiser colocá-lo de volta em um `DataFrame` Pandas, é simples:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

## Scikit-Learn Design

A API do Scikit-Learn é muito bem projetada. Os principais princípios de design (<http://goo.gl/wL10sI>) são:<sup>15</sup>

- **Consistência.** Todos os objetos compartilham uma interface simples e consistente:
  - *Estimadores.* Qualquer objeto que possa estimar alguns parâmetros com base em um conjunto de dados é chamado de *estimador* (por exemplo, um `Imputer` é um estimador). A estimativa em si é realizada pelo método `fit()`, e é necessário apenas um conjunto de dados como parâmetro (ou dois para algoritmos de aprendizado supervisionado; o segundo conjunto de dados contém os rótulos). Qualquer outro parâmetro necessário para orientar o processo de estimativa é considerado um hiperparâmetro (como `Imputer's strategy`), e deve ser configurado como uma variável da instância (geralmente por meio de um parâmetro do construtor).
  - *Transformadores.* Alguns estimadores (como um `Imputer`) também podem transformar um conjunto de dados; esses são chamados *transformadores*. Mais uma vez, a API é bem simples: a transformação é realizada pelo método `transform()` para transformar o conjunto de dados como um parâmetro. Ele retorna o conjunto de dados transformado. Esta transformação geralmente depende dos parâmetros aprendidos, como é o caso de um `Imputer`. Todos os transformadores também possuem um método de conveniência chamado `fit_transform()` que é o equivalente a chamar a `fit()` e então `transform()` (mas algumas vezes `fit_transform()` é otimizado e roda muito mais rápido).
  - *Previsores.* Finalmente, a partir de um conjunto de dados, alguns estimadores são capazes de fazer previsões; eles são chamados *previsores*. Por exemplo, o modelo `LinearRegression` do capítulo anterior foi um previsor: previu a satisfação de vida, dado o PIB per capita de um país. Um previsor tem um método `predict()` que pega um conjunto de dados de novas instâncias e retorna um conjunto de dados de previsões correspondentes. Ele também tem um método `score()`, que mede a qualidade das previsões, dado um conjunto de teste (e os rótulos correspondentes no caso de algoritmos de aprendizado supervisionado).<sup>16</sup>
- **Inspeção.** Todos os hiperparâmetros do estimador são diretamente acessíveis por meio de variáveis das instâncias públicas (por exemplo, `Imputer.strategy`), e todos os parâmetros aprendidos do estimador também são acessíveis por variáveis das instâncias públicas com um sufixo de sublinhado (por exemplo, `Imputer.statistics_`).

15 Para mais detalhes em princípios de design, consulte “API design for machine learning software: experiences from the scikit-learn project”, L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Müller, et al. (2013).

16 Alguns previsores também fornecem métodos para medir a confiança de suas previsões.

- **Não proliferação de classes.** Os conjuntos de dados são representados como arrays NumPy ou matrizes esparsas SciPy em vez de classes caseiras. Os hiperparâmetros são apenas strings ou números Python.
- **Composição.** Os blocos de construção existentes são reutilizados tanto quanto possível. Por exemplo, é fácil criar um estimador `Pipeline` seguido por um estimador final a partir de uma sequência arbitrária de transformadores, como veremos mais à frente.
- **Padrões sensíveis.** O Scikit-Learn fornece valores padrão razoáveis para a maioria dos parâmetros, facilitando a criação rápida de um sistema padronizado de trabalho.

## Manipulando Texto e Atributos Categóricos

Anteriormente, excluímos o atributo categórico `ocean_proximity` por ser um atributo de texto, portanto não podemos calcular sua média:

```
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat.head(10)
17606      <1H OCEAN
18632      <1H OCEAN
14650      NEAR OCEAN
3230       INLAND
3555      <1H OCEAN
19480      INLAND
8879      <1H OCEAN
13685      INLAND
4937      <1H OCEAN
4861      <1H OCEAN
Name: ocean_proximity, dtype: object
```

De qualquer modo, a maioria dos algoritmos de Aprendizado de Máquina prefere trabalhar com números, então vamos converter essas categorias de texto para números. Para tanto, podemos utilizar o método `factorize()` do Pandas, que mapeia cada categoria para um número inteiro diferente:

```
>>> housing_cat_encoded, housing_categories = housing_cat.factorize()
>>> housing_cat_encoded[:10]
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])
```

Este é melhor: `housing_cat_encoded` agora é puramente numérico. O método `factorize()` também retorna a lista de categorias (“<1H OCEAN” foi mapeado para 0, “NEAR OCEAN” foi mapeado para 1, etc.):

```
>>> housing_categories
Index(['<1H OCEAN', 'NEAR OCEAN', 'INLAND', 'NEAR BAY', 'ISLAND'], dtype='object')
```

Um problema nesta representação é que os algoritmos de Aprendizado de Máquina assumirão que dois valores próximos são mais parecidos do que dois valores distantes. Obviamente, este não é o caso (por exemplo, as categorias 0 e 4 são mais semelhantes do que as categorias 0 e 2). Para corrigir este problema, uma solução comum seria a criação de um atributo binário por categoria: um atributo igual a 1 quando a categoria for “<1H OCEAN” (e 0 caso contrário), outro atributo igual a 1 quando a categoria for “NEAR OCEAN” (e 0 caso contrário), e assim por diante. Isso é chamado de *one-hot encoding*, porque apenas um atributo será igual a 1 (*hot*), enquanto os outros serão 0 (*cold*).

O Scikit-Learn fornece um codificador `OneHotEncoder` para converter valores categóricos inteiros em vetores *one-hot*. Vamos programar as categorias como vetores *one-hot*:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>
      with 16512 stored elements in Compressed Sparse Row format>
```

Note que `fit_transform()` espera um *array* 2D, mas `housing_cat_encoded` é um *array* 1D, então precisamos remodelá-lo.<sup>17</sup> Além disso, observe que a saída é uma *matriz esparsa* SciPy, não um *array* NumPy. Isso é muito útil quando você possui atributos categóricos com milhares de categorias. Após um *one-hot encoding*, obtemos uma matriz com milhares de colunas e cheia de zeros, exceto por um único 1 por linha. Seria muito desperdício utilizar toneladas de memória para armazenar principalmente zeros, então, em vez disso, uma matriz esparsa armazena apenas a localização dos elementos diferentes de zero. Você pode utilizá-la principalmente como um *array* 2D normal,<sup>18</sup> mas, se realmente quiser convertê-la em um *array* (denso) NumPy, basta chamar o método `toarray()`:

```
>>> housing_cat_1hot.toarray()
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Você pode aplicar ambas as transformações de uma única vez (de categorias de texto a categorias de inteiros, depois de categorias de inteiros para vetores *one-hot*) utilizando a classe `CategoricalEncoder`. Ela não faz parte do Scikit-Learn 0.19.0 e mais antigos, mas

---

<sup>17</sup> A função NumPy `reshape()` permite que uma dimensão seja -1, o que significa “não especificado”: o valor é inferido do comprimento do array e das dimensões restantes.

<sup>18</sup> Veja a documentação do SciPy para mais detalhes.

será adicionada em breve, então talvez já esteja disponível no momento em que você lê este livro. Se não estiver, pode obtê-la no notebook Jupyter para este capítulo (o código foi copiado do Pull Request #9151). Segue abaixo como usá-la:

```
>>> from sklearn.preprocessing import CategoricalEncoder # ou pegue do notebook
>>> cat_encoder = CategoricalEncoder()
>>> housing_cat_reshaped = housing_cat.values.reshape(-1, 1)
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
>>> housing_cat_1hot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>
      with 16512 stored elements in Compressed Sparse Row format>
```

Por padrão, o `CategoricalEncoder` mostra uma matriz esparsa, mas você pode configurar a codificação para “onehot-dense” se preferir uma matriz densa:

```
>>> cat_encoder = CategoricalEncoder(encoding="onehot-dense")
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat_reshaped)
>>> housing_cat_1hot
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Você pode obter a lista de categorias utilizando a variável de instância `categories_`. É uma lista que contém um array de 1D de categorias para cada atributo categórico (neste caso, uma lista contendo um único array, uma vez que existe apenas um atributo categórico):

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'], dtype=object)]
```



Se um atributo categórico tiver um grande número de categorias possíveis (por exemplo, código do país, profissão, espécies, etc.), então uma codificação *one-hot* resultará em um grande número de características de entrada. Isso pode diminuir o treinamento e degradar o desempenho. Se isso acontecer, será necessário produzir representações mais densas chamadas *embeddings*, mas isso requer uma boa compreensão das redes neurais (veja o Capítulo 14 para mais detalhes).

## Customize Transformadores

Embora o Scikit-Learn forneça muitos transformadores úteis, você precisará escrever seus próprios para tarefas como operações de limpeza personalizadas ou combinar atributos específicos. É preciso que o seu transformador funcione perfeitamente

com as funcionalidades do Scikit-Learn (como os *pipelines*) e, como o Scikit-Learn depende da tipagem *duck typing* (não herança), você só precisa criar uma classe e implementar os três métodos: `fit()` (retornando `self`), `transform()` e `fit_transform()`. Você pode obter o último de graça ao simplesmente acrescentar `TransformerMixin` em uma classe base. Além disso, se você adicionar `BaseEstimator` como uma classe base (e evitar `*args` e `**kwargs` em seu construtor) você receberá dois métodos extras (`get_params()` e `set_params()`) que serão úteis para o ajuste automático dos hiperparâmetros. Por exemplo, esta é uma pequena classe `transformer` que adiciona os atributos combinados discutidos anteriormente:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # sem *args ou **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                       bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

Neste exemplo, o transformador tem um hiperparâmetro, `add_bedrooms_per_room`, definido por padrão como `True` (geralmente é útil fornecer padrões sensíveis). Este hiperparâmetro permitirá que você descubra facilmente se a adição deste atributo ajuda ou não os algoritmos de Aprendizado de Máquina. De forma mais geral, você pode adicionar um hiperparâmetro para controlar qualquer etapa da preparação de dados sobre a qual você não tem 100% de certeza. Quanto mais você automatizar essas etapas de preparação de dados, mais combinações poderá experimentar automaticamente, tornando muito mais provável encontrar uma ótima combinação (e economizar muito tempo).

## Escalonamento das Características

Uma das transformações mais importantes que você precisa aplicar aos seus dados é o *escalonamento das características*. Com poucas exceções, os algoritmos de Aprendizado de Máquina não funcionam bem quando atributos numéricos de entrada têm escalas muito diferentes. Este é o caso dos dados do setor imobiliário: o número total de cômodos

varia de 6 a 39.320, enquanto os rendimentos médios variam apenas de 0 a 15. Observe que geralmente não é necessário escalar os valores-alvo.

Existem duas maneiras comuns de todos os atributos obterem a mesma escala: *escala min-max e padronização*.

O escalonamento min-max (muitas pessoas chamam de *normalização*) é bastante simples: os valores são deslocados e redimensionados para que acabem variando de 0 a 1. Fazemos isso subtraindo o valor mínimo e dividindo pelo máximo menos o mínimo. O Scikit-Learn fornece um transformador chamado `MinMaxScaler` para isso. Ele possui um hiperparâmetro `feature_range` que permite alterar o intervalo se você não quiser 0-1 por algum motivo.

A padronização é bem diferente: em primeiro lugar ela subtrai o valor médio (assim os valores padronizados sempre têm média zero) e, em seguida, divide pela variância, de modo que a distribuição resultante tenha variância unitária. Ao contrário do escalonamento min-max, a padronização não vincula valores a um intervalo específico, o que pode ser um problema para alguns algoritmos (por exemplo, as redes neurais geralmente esperam um valor de entrada variando de 0 a 1). No entanto, a padronização é muito menos afetada por outliers. Por exemplo, suponha que um bairro tenha uma renda média igual a 100 (por engano). O escalonamento min-max, em seguida, comprimiria todos os outros valores de 0-15 para 0-0,15, enquanto a padronização não seria muito afetada. O Scikit-Learn fornece um transformador para padronização chamado `StandardScaler`.



Tal como acontece com todas as transformações, é importante encaixar os *escalonadores* apenas nos dados de treinamento e não no conjunto completo de dados (incluindo o conjunto de testes). Só então você pode utilizá-los para transformar o conjunto de treinamento e o conjunto de teste (e novos dados).

## Pipelines de Transformação

Como você pode ver, existem muitas etapas de transformação de dados que precisam ser executadas na ordem correta. Felizmente, o Scikit-Learn fornece a classe `Pipeline` para ajudar com tais sequências de transformações. Eis um pequeno pipeline para os atributos numéricos:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

O construtor `Pipeline` se vale de uma lista de pares de nome/estimador que definem uma sequência de etapas. Todos, exceto o último estimador, devem ser transformadores (ou seja, eles devem ter um método `fit_transform()`). Os nomes podem ser o que você quiser (desde que não contenham sublinhados duplos “`__`”).

Quando você chama o método `fit()` do pipeline, ele chama `fit_transform()` sequencialmente em todos os transformadores, passando a saída de cada chamada como parâmetro para a próxima chamada, até chegar ao estimador final, o qual chama apenas o método `fit()`.

O pipeline expõe os mesmos métodos que o estimador final. Neste exemplo, o último estimador é um `StandardScaler`, que é um transformador, então o pipeline possui um método `transform()` que aplica todas as transformações aos dados em sequência (também possui um método `fit_transform` que poderíamos ter usado em vez de chamar a `fit()` e depois `transform()`).

Agora, seria bom se pudéssemos fornecer diretamente em nosso pipeline um `DataFrame` Pandas que contivesse colunas não numéricas em vez de termos que primeiro extrair manualmente as colunas numéricas em um array NumPy. Não há nada no Scikit-Learn que lide com os `DataFrames` Pandas,<sup>19</sup> mas podemos escrever um transformador personalizado para esta tarefa:

```
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Nosso `DataFrameSelector` transformará os dados selecionando os atributos desejados, descartando o resto e convertendo o `DataFrame` resultante em um array NumPy. Com isso, você pode facilmente escrever um pipeline que terá um `DataFrame` Pandas e lidar apenas com os valores numéricos: o pipeline iniciaria apenas com um `DataFrameSelector` para escolher os atributos numéricos, seguido dos outros passos de pré-processamento que discutimos anteriormente. E você também pode escrever outro pipeline com facilidade para os atributos categóricos simplesmente ao selecioná-los utilizando um `DataFrameSelector` e depois aplicando um `CategoricalEncoder`.

---

<sup>19</sup> Confira também Pull Request #3886, que deve introduzir uma classe `ColumnTransformer`, que facilitará transformações específicas de atributos. Você também pode testar `pip3 install sklearn-pandas` para obter uma classe `DataFrameMapper` com um objetivo semelhante.

```

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

```

Mas como é possível juntar esses dois pipelines em um único? A resposta é utilizar a classe `FeatureUnion` do Scikit-Learn. Você lhe dá uma lista de transformadores (que podem ser pipelines transformadores inteiros); quando o método `transform()` é chamado, ele executa cada método `transform()` em paralelo, aguarda sua saída e, em seguida, os concatena e retorna o resultado (e, claro, chamar o método `fit()` chama cada método `fit()` do transformador). Um pipeline completo que manipula ambos atributos numéricos e categóricos pode ser mais ou menos assim:

```

from sklearn.pipeline import FeatureUnion

full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])

```

E você pode executar todo o pipeline simplesmente:

```

>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ -1.15604281,   0.77194962,   0.74333089, ...,  0.        ,
         0.        ,  0.        ],
       [-1.17602483,   0.6596948 ,  -1.1653172 , ...,  0.        ,
         0.        ,  0.        ],
       [...]
      ])
>>> housing_prepared.shape
(16512, 16)

```

## Selecionar e Treinar um Modelo

Finalmente! Você enquadrou o problema, obteve os dados e os explorou, selecionou um conjunto de treinamento e um conjunto de testes e escreveu canais de transformação para limpar e preparar automaticamente seus dados para os algoritmos de Aprendizado de Máquina. Agora você está pronto para selecionar e treinar um modelo de Aprendizado de Máquina.

## Treinando e Avaliando o Conjunto de Treinamento

A boa notícia é que, graças a todas essas etapas anteriores, as coisas agora serão muito mais simples do que você pensa. Treinaremos primeiro um modelo de Regressão Linear, como fizemos no capítulo anterior:

```
from sklearn.linear_model import LinearRegression  
  
lin_reg = LinearRegression()  
lin_reg.fit(housing_prepared, housing_labels)
```

Feito! Agora, você possui um modelo de Regressão Linear funcional. Vamos tentar isso em algumas instâncias do conjunto de treinamento:

```
>>> some_data = housing.iloc[:5]  
>>> some_labels = housing_labels.iloc[:5]  
>>> some_data_prepared = full_pipeline.transform(some_data)  
>>> print("Predictions:", lin_reg.predict(some_data_prepared))  
Predictions: [ 210644.6045  317768.8069  210956.4333  59218.9888  189747.5584]  
>>> print("Labels:", list(some_labels))  
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Funciona, embora as previsões não sejam exatamente precisas (por exemplo, a primeira previsão tem cerca de 40% de erro!). Vamos medir a RMSE desse modelo de regressão em todo o conjunto de treinamento com o uso da função `mean_squared_error` do Scikit-Learn:

```
>>> from sklearn.metrics import mean_squared_error  
>>> housing_predictions = lin_reg.predict(housing_prepared)  
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)  
>>> lin_rmse = np.sqrt(lin_mse)  
>>> lin_rmse  
68628.198198489219
```

Ok, isso é melhor do que nada, mas certamente não é uma grande pontuação: a maioria dos `median_housing_values` dos bairros varia entre US\$ 120 mil e US\$ 265 mil, então um erro de margem típico de US\$ 68.628 não é muito aceitável. Este é um exemplo de um modelo de subajuste dos dados de treinamento. Quando isso acontece, pode significar que as características não fornecem informações suficientes para fazer boas previsões ou que o modelo não é suficientemente poderoso. Como vimos no capítulo anterior, as principais formas de corrigir o *subajuste* são: selecionar um modelo mais poderoso, alimentar o algoritmo de treinamento com melhores características ou reduzir as restrições no modelo. Este modelo não é regularizado, o que exclui a última opção. Você poderia tentar adicionar mais características (por exemplo, o registro da população), mas primeiro vamos tentar um modelo mais complexo para ver como ele se sai.

Treinaremos um `DecisionTreeRegressor`. Este é um modelo poderoso, capaz de encontrar relações não lineares complexas nos dados (as Árvores de Decisão serão apresentadas com mais detalhes no Capítulo 6). O código deve ser familiar agora:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

Agora que o modelo está treinado, vamos avaliá-lo no conjunto de treinamento:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Espere, o quê?! Nenhum erro? Será que esse modelo é realmente absolutamente perfeito? Claro, é muito mais provável que o modelo tenha se sobreajustado mal aos dados. Como você pode ter certeza? Como vimos anteriormente, você não pode tocar no conjunto de testes até que esteja pronto para lançar um modelo confiável, então você precisa utilizar parte do conjunto de treinamento para treinar, e parte para validar o modelo.

## Avaliando Melhor com a Utilização da Validação Cruzada

Uma maneira de avaliar o modelo da Árvore de Decisão seria utilizar a função `train_test_split` para dividir o conjunto de treinamento em um conjunto menor de treinamento e um conjunto de validação, em seguida treinar seus modelos com o conjunto menor e avaliá-los com o conjunto de validação. É um pouco trabalhoso, mas nada muito difícil e funciona muito bem.

Uma ótima alternativa é utilizar o recurso da *validação cruzada* do Scikit-Learn. O código a seguir executa a *validação cruzada K-fold*: ele divide aleatoriamente o conjunto de treinamento em 10 subconjuntos distintos chamados de partes (*folds*), então treina e avalia o modelo da Árvore de Decisão 10 vezes escolhendo uma parte (*fold*) diferente a cada uma delas para avaliação e treinando nas outras 9 partes. O resultado é um array contendo as 10 pontuações de avaliação:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                        scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```



Os recursos da validação cruzada do Scikit-Learn esperam uma função de utilidade (mais alta é melhor) ao invés de uma função de custo (mais baixa é melhor), de modo que a função de pontuação é exatamente o oposto do MSE (ou seja, um valor negativo), e é por isso que o código anterior calcula `-scores` antes de calcular a raiz quadrada.

Vamos olhar os resultados:

```
>>> def display_scores(scores):
...     print("Scores:", scores)
...     print("Mean:", scores.mean())
...     print("Standard deviation:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Scores: [ 70232.0136482   66828.46839892  72444.08721003  70761.50186201
         71125.52697653  75581.29319857  70169.59286164  70055.37863456
         75370.49116773  71222.39081244]
Mean: 71379.0744771
Standard deviation: 2458.31882043
```

Agora a Árvore de Decisão não tem uma aparência tão boa quanto antes. Na verdade, parece ser pior do que o modelo de Regressão Linear! Observe que a validação cruzada permite que você obtenha não apenas uma estimativa do desempenho do seu modelo, mas também uma medida da precisão dessa estimativa (ou seja, seu desvio padrão). A Árvore de Decisão possui uma pontuação de aproximadamente 71.379, geralmente  $\pm$  2.458. Você não teria essa informação se utilizasse apenas um conjunto de validação. Mas a validação cruzada treina o modelo várias vezes, então nem sempre é possível.

Calcularemos as mesmas pontuações para o modelo de Regressão Linear apenas para ter certeza:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
...                                 scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Scores: [ 66782.73843989  66960.118071    70347.95244419  74739.57052552
         68031.13388938  71193.84183426  64969.63056405  68281.61137997
         71552.91566558  67665.10082067]
Mean: 69052.4613635
Standard deviation: 2731.6740018
```

É isso mesmo: o modelo da Árvore de Decisão está se sobreajustando tanto que acaba sendo pior do que o modelo de Regressão Linear.

Vamos tentar um último modelo agora: `RandomForestRegressor`. Como veremos no Capítulo 7, Florestas Aleatórias funcionam com o treinamento de muitas Árvores de Decisão em subconjuntos aleatórios das características, e em seguida calculam a média de suas

previsões. Construir um modelo em cima de muitos outros modelos é chamado *Ensemble Learning*, e muitas vezes é uma ótima maneira de aumentar ainda mais os algoritmos de Aprendizado de Máquina. Ignoraremos a maior parte do código, pois é essencialmente o mesmo para os outros modelos:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
21941.911027380233
>>> display_scores(forest_rmse_scores)
Scores: [ 51650.94405471  48920.80645498  52979.16096752  54412.74042021
         50861.29381163  56488.55699727  51866.90120786  49752.24599537
         55399.50713191  53309.74548294]
Mean: 52564.1902524
Standard deviation: 2301.87380392
```

Isso é muito melhor: Florestas Aleatórias parecem muito promissoras. No entanto, note que a pontuação no conjunto de treinamento ainda é muito menor do que nos conjuntos de validação, o que significa que o modelo ainda está se sobreajustando ao conjunto de treinamento. Possíveis soluções para sobreajustes são: simplificar o modelo, restringi-lo (ou seja, regularizá-lo), ou obter muito mais dados de treinamento. No entanto, antes de mergulhar mais profundamente em Florestas Aleatórias, você deve experimentar muitos outros modelos de várias categorias de algoritmos de Aprendizado de Máquina (várias Máquinas de Vetores de Suporte com diferentes *kernels*, possivelmente uma rede neural, etc.), sem gastar muito tempo ajustando os hiperparâmetros. O objetivo é selecionar alguns modelos promissores (de dois a cinco).



Você deve salvar todos os modelos que experimenta para que possa voltar facilmente para qualquer um deles. Certifique-se de salvar os hiperparâmetros e os parâmetros treinados, bem como as pontuações de validação cruzada e talvez as previsões também. Isso permitirá que você compare facilmente as pontuações entre tipos de modelo e compare os tipos de erros que eles cometem. Você pode facilmente salvar modelos do Scikit-Learn utilizando o módulo `pickle` do Python ou usando o `sklearn.externals.joblib`, que é mais eficiente na serialização de grandes arrays NumPy:

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

## Ajuste Seu Modelo

Assumiremos que você tem uma lista restrita de modelos promissores. Agora, você precisa ajustá-los. Vejamos algumas maneiras de fazer isso.

### Grid Search

Uma maneira de fazer isso seria alterar manualmente os hiperparâmetros até encontrar uma ótima combinação de valores. Este seria um trabalho muito tedioso, e talvez você não tenha tempo para explorar muitas combinações.

Em vez disso, acione o `GridSearchCV` do Scikit-Learn para efetuar a busca por você. Você só precisa passar quais hiperparâmetros deseja que ele experimente e quais valores tentar, e ele avaliará todas as combinações de valores possíveis por meio de validação cruzada. Por exemplo, o código a seguir busca a melhor combinação de valores dos hiperparâmetros para `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```



Uma abordagem simples seria testar potências de 10 consecutivas quando você não tem ideia do valor que um hiperparâmetro deve ter (ou um número menor se você quiser uma busca mais refinada, como mostrado neste exemplo com o hiperparâmetro `n_estimators`).

Este `param_grid` pede ao Scikit-Learn que primeiro avalie todas as  $3 \times 4 = 12$  combinações de valores dos hiperparâmetros `n_estimators` e `max_features` especificados na primeira `dict` (não se preocupe com o que esses hiperparâmetros significam por enquanto, eles serão explicados no Capítulo 7), então tente todas as  $2 \times 3 = 6$  combinações de valores do hiperparâmetro na segunda `dict`, mas desta vez com o hiperparâmetro de inicialização definido como `False` em vez de `True` (que é o valor padrão para este hiperparâmetro).

De modo geral, a grid search explorará  $12 + 6 = 18$  combinações de valores do hiperparâmetro `RandomForestRegressor` e treinará cada modelo cinco vezes (já que estamos

utilizando validação cruzada de cinco partes). Em outras palavras, no geral, haverá  $18 \times 5 = 90$  rodadas de treinamento! Pode demorar muito, mas quando terminar você obterá a melhor combinação de parâmetros:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```



Uma vez que 8 e 30 são os valores máximos avaliados, você provavelmente deve tentar pesquisar outra vez com valores mais elevados, já que a pontuação pode continuar a melhorar.

Você também pode obter diretamente o melhor estimador:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                      max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=30, n_jobs=1, oob_score=False, random_state=42,
                      verbose=0, warm_start=False)
```



Se `GridSearchCV` for inicializado com `refit=True` (que é o padrão), assim que ele encontrar o melhor estimador utilizando a validação cruzada, ele o treinará novamente em todo o conjunto. Esta geralmente é uma boa ideia, pois fornecer mais dados melhora potencialmente seu desempenho.

E, claro, as pontuações de avaliação também estão disponíveis:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63647.854446 {'n_estimators': 3, 'max_features': 2}
55611.5015988 {'n_estimators': 10, 'max_features': 2}
53370.0640736 {'n_estimators': 30, 'max_features': 2}
60959.1388585 {'n_estimators': 3, 'max_features': 4}
52740.5841667 {'n_estimators': 10, 'max_features': 4}
50374.1421461 {'n_estimators': 30, 'max_features': 4}
58661.2866462 {'n_estimators': 3, 'max_features': 6}
52009.9739798 {'n_estimators': 10, 'max_features': 6}
50154.1177737 {'n_estimators': 30, 'max_features': 6}
57865.3616801 {'n_estimators': 3, 'max_features': 8}
51730.0755087 {'n_estimators': 10, 'max_features': 8}
49694.8514333 {'n_estimators': 30, 'max_features': 8}
62874.4073931 {'n_estimators': 3, 'bootstrap': False, 'max_features': 2}
54643.4998083 {'n_estimators': 10, 'bootstrap': False, 'max_features': 2}
59437.8922859 {'n_estimators': 3, 'bootstrap': False, 'max_features': 3}
52735.3582936 {'n_estimators': 10, 'bootstrap': False, 'max_features': 3}
57490.0168279 {'n_estimators': 3, 'bootstrap': False, 'max_features': 4}
51008.2615672 {'n_estimators': 10, 'bootstrap': False, 'max_features': 4}
```

Neste exemplo, obtemos a melhor solução definindo o hiperparâmetro `max_features` em 8 e o hiperparâmetro `n_estimators` em 30. A pontuação da RMSE para esta combinação é 49.694, o que é um pouco melhor do que o resultado obtido anteriormente utilizando os valores padrão do hiperparâmetro (que era 52.564). Parabéns, você ajustou com sucesso o seu melhor modelo!



Não se esqueça de que alguns dos passos da preparação de dados podem ser tratados como hiperparâmetros. Por exemplo, a grid search descobrirá automaticamente se deve ou não adicionar uma característica que você não tinha certeza (por exemplo, utilizando o hiperparâmetro `add_bedrooms_per_room` do seu transformador `CombinedAttributesAdder`). Similarmente, ele também pode ser usado para encontrar automaticamente a melhor maneira de lidar com os outliers, características perdidas, seleção das características e muito mais.

## Randomized Search

Se estiver explorando relativamente poucas combinações, a abordagem da grid search é boa, como no exemplo anterior, mas quando o *espaço de busca* do hiperparâmetro for grande, é preferível utilizar `RandomizedSearchCV` em seu lugar. Esta classe pode ser utilizada da mesma maneira que a classe `GridSearchCV`, mas, em vez de tentar todas as combinações possíveis, ela seleciona um valor aleatório para cada hiperparâmetro em cada iteração e avalia um determinado número de combinações aleatórias. Esta abordagem tem dois benefícios principais:

- Se você deixar que a pesquisa randomizada execute, por exemplo, mil iterações, essa abordagem explorará mil valores diferentes para cada hiperparâmetro (em vez de apenas alguns valores por hiperparâmetro na abordagem da grid search);
- Estabelecendo o número de iterações, você terá mais controle na pesquisa dos hiperparâmetros sobre o orçamento que deseja alocar.

## Métodos de Ensemble

Outra maneira de ajustar seu sistema é tentar combinar os modelos de melhor desempenho. O grupo (ou “ensemble”) geralmente será superior ao melhor modelo individual (assim como as Florestas Aleatórias funcionam melhor do que as Árvores de Decisão individuais em que se baseiam), especialmente se os modelos individuais tiverem tipos muito diferentes de erros. Abordaremos este tópico com mais detalhes no Capítulo 7.

## Analise os Melhores Modelos e Seus Erros

Muitas vezes você obterá boas ideias sobre o problema ao inspecionar os melhores modelos. Por exemplo, o `RandomForestRegressor` pode indicar a importância relativa de cada atributo para fazer previsões precisas:

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.33442355e-02,   6.29090705e-02,   4.11437985e-02,
       1.46726854e-02,   1.41064835e-02,   1.48742809e-02,
       1.42575993e-02,   3.66158981e-01,   5.64191792e-02,
       1.08792957e-01,   5.33510773e-02,   1.03114883e-02,
       1.64780994e-01,   6.02803867e-05,   1.96041560e-03,
       2.85647464e-03])
```

Mostraremos essas pontuações de importância ao lado de seus nomes de atributos correspondentes:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = cat_pipeline.named_steps["cat_encoder"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.0028564746373201579, 'NEAR OCEAN'),
 (0.0019604155994780701, 'NEAR BAY'),
 (6.0280386727365991e-05, 'ISLAND')]
```

Com esta informação, você pode tentar descartar algumas das características menos úteis (por exemplo, aparentemente apenas uma categoria `ocean_proximity` é realmente útil, então você pode tentar descartar as outras).

Você também deve analisar os erros específicos cometidos pelo seu sistema, depois tentar entender por que ele os faz e o que poderia solucionar o problema (adicionar características extras ou, ao contrário, se livrar das não informativas, limpar outliers, etc.).

## Avalie Seu Sistema no Conjunto de Testes

Depois de ajustar seus modelos por um tempo você terá um sistema que funciona suficientemente bem. Agora é a hora de avaliar o modelo final no conjunto de teste. Não há nada de especial neste processo; apenas obtenha as previsões e os rótulos do seu conjunto de teste, execute `full_pipeline` para transformar os dados (chame `transform()`, *não fit\_transform()!*) e avalie o modelo final no conjunto de teste:

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()
X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
```

O desempenho geralmente será um pouco pior do que o medido usando validação cruzada se você fez muitos ajustes de hiperparâmetro (porque o sistema acabou sendo ajustado para executar bem com dados de validação e provavelmente não funcionará tão bem em conjuntos desconhecidos de dados). Não é o caso neste exemplo, mas, quando isso acontece, você deve resistir à tentação de ajustar os hiperparâmetros para que os números fiquem mais atraentes no conjunto de teste; as melhorias não seriam generalizadas para novos dados.

Agora, vem a fase de pré-lançamento do projeto: você precisa apresentar sua solução (destacando o que aprendeu, o que funcionou e o que não, quais pressupostos foram feitos e quais as limitações do seu sistema); documente tudo e crie apresentações detalhadas com visualizações claras e declarações que sejam fáceis de lembrar (por exemplo, “a renda média é o principal previsor dos preços do setor imobiliário”).

## Lance, Monitore e Mantenha seu Sistema

Perfeito, você obteve aprovação para o lançamento! Você precisa preparar sua solução para a produção, principalmente conectando as fontes de dados de entrada da produção ao seu sistema e escrevendo testes.

Você também precisa escrever o código de monitoramento para verificar o desempenho ao vivo em intervalos regulares do seu sistema e acionar alertas quando ele ficar offline. Isso é importante, não apenas para capturar uma quebra súbita, mas também a degradação do desempenho. Isso é bastante comum porque os modelos tendem a “deteriorar”

à medida que os dados evoluem ao longo do tempo, a menos que sejam regularmente treinados em novos dados.

A avaliação do desempenho do seu sistema exigirá uma amostragem das previsões do sistema e sua avaliação. Isso geralmente requer uma análise humana. Esses analistas podem ser especialistas ou trabalhadores de uma plataforma de *crowdsourcing* (como a Amazon Mechanical Turk ou CrowdFlower). De qualquer forma, você precisará conectar o canal de avaliação humana ao seu sistema.

Você também deve certificar-se de avaliar a qualidade de dados de entrada do sistema. Às vezes, o desempenho se deteriorará levemente por causa de um sinal de má qualidade (por exemplo, uma falha do sensor que envia valores aleatórios ou o resultado de outra equipe que se torna obsoleto), mas pode demorar até que o desempenho se degrade o suficiente para disparar um alerta. Você pode capturar isso antecipadamente se monitorar suas entradas. O monitoramento das entradas é particularmente importante para os sistemas de *aprendizado online*.

Finalmente, você deve treinar regularmente seus modelos com a utilização de novos dados. Esse processo deve ser automatizado tanto quanto possível. Se não for, a probabilidade é que você apenas o atualize a cada seis meses (na melhor das hipóteses), e o seu desempenho pode variar bastante ao longo do tempo. Se for um sistema de *aprendizado online* certifique-se de salvar *capturas de tela* do seu estado atual em intervalos regulares para que seja possível reverter facilmente para um estágio anterior do trabalho.

## Experimente!

Esperamos que este capítulo tenha dado uma boa ideia do que é um projeto de Aprendizado de Máquina e tenha mostrado algumas das ferramentas que você utilizará para treinar um grande sistema. Como você pode ver, grande parte do trabalho está na etapa de preparação dos dados, na construção de ferramentas de monitoramento, na criação de canais de avaliação humana e na automação do treinamento regular de um modelo. Os algoritmos de Aprendizado de Máquina também são importantes, é claro, mas é preferível ficar confortável com o processo geral e ter um bom conhecimento de três ou quatro algoritmos do que gastar todo o seu tempo explorando algoritmos avançados e não ter tempo suficiente para o processo como um todo.

Então, se você ainda não o fez, agora é um bom momento para pegar um notebook, selecionar um conjunto de dados do seu interesse e tentar passar por todo o processo de A a Z. Um bom lugar para começar seria em um website de competição como o <http://kaggle.com/>: você terá um conjunto de dados para utilização, um objetivo claro e pessoas com quem compartilhar a experiência.

## Exercícios

Utilizando o conjunto de dados do setor imobiliário deste capítulo:

1. Experimente um regressor da Máquina de Vetores de Suporte (`sklearn.svm.SVR`), com vários hiperparâmetros, como `kernel="linear"` (com vários valores para o hiperparâmetro `C`) ou `kernel="rbf"` (com vários valores para os hiperparâmetros `C` e `gamma`). Não se preocupe com o significado desses hiperparâmetros por enquanto. Qual será o desempenho do melhor previsor da `SVR`?
2. Tente substituir `GridSearchCV` por `RandomizedSearchCV`.
3. Tente acrescentar um transformador no pipeline de preparação para selecionar apenas os atributos mais importantes.
4. Tente criar um pipeline único que faça a preparação completa de dados mais a previsão final.
5. Explore automaticamente algumas opções de preparação utilizando o `GridSearchCV`.

As soluções para estes exercícios estão disponíveis online nos notebooks Jupyter em <https://github.com/ageron/handson-ml>.

## Capítulo 3

# Classificação

No Capítulo 1, mencionamos que regressão (previsão de valores) e classificação (previsão de classes) são as tarefas de aprendizado supervisionado mais comuns. No Capítulo 2, utilizando diversos algoritmos como a Regressão Linear, Árvores de Decisão e Florestas Aleatórias (que será explicado em detalhes em capítulos posteriores), exploramos uma tarefa de regressão para prever preços do mercado imobiliário. Agora, voltaremos nossa atenção para os sistemas de classificação.

## MNIST

Neste capítulo, utilizaremos o conjunto de dados MNIST, composto de 70 mil pequenas imagens de dígitos escritos à mão por estudantes do ensino médio e funcionários do *US Census Bureau*. Cada imagem é rotulada com o dígito que a representa. Este conjunto tem sido tão estudado que, muitas vezes, é chamado de o "*Hello World*" do Aprendizado de Máquina: sempre que as pessoas apresentam um novo algoritmo de classificação, elas têm curiosidade em ver como será seu desempenho no MNIST. Sempre que alguém estuda o Aprendizado de Máquina, mais cedo ou mais tarde lidará com o MNIST.

O Scikit-Learn fornece muitas funções auxiliares para baixar conjuntos de dados populares. O MNIST é um deles. O código a seguir se vale do conjunto de dados MNIST:<sup>1</sup>

```
>>> from sklearn.datasets import fetch_mldata  
>>> mnist = fetch_mldata('MNIST original')  
>>> mnist  
{'COL_NAMES': ['label', 'data'],  
 'DESCR': 'mldata.org dataset: mnist-original',  
 'data': array([[0, 0, 0, ..., 0, 0, 0],  
 [0, 0, 0, ..., 0, 0, 0],
```

---

<sup>1</sup> Por padrão, o Scikit-Learn armazena conjuntos de dados baixados em um diretório chamado \$HOME/scikit\_learn\_data.

```
[0, 0, 0, ..., 0, 0, 0],  
...,  
[0, 0, 0, ..., 0, 0, 0],  
[0, 0, 0, ..., 0, 0, 0],  
[0, 0, 0, ..., 0, 0, 0]], dtype=uint8),  
'target': array([ 0.,  0.,  0., ...,  9.,  9.,  9.])}
```

Os conjuntos de dados carregados pelo Scikit-Learn geralmente possuem uma estrutura similar ao dicionário, incluindo:

- Uma chave `DESCR` que descreve o conjunto de dados;
- Uma chave de `dados` que contém um array com uma linha por instância e uma coluna por característica;
- Uma chave alvo contendo um array com os rótulos.

Vejamos estes arrays:

```
>>> X, y = mnist["data"], mnist["target"]  
>>> X.shape  
(70000, 784)  
>>> y.shape  
(70000,)
```

Há 70 mil imagens e cada imagem possui 784 características. Isso ocorre porque cada imagem tem  $28 \times 28$  pixels, e cada característica representa a intensidade de um pixel, de 0 (branco) a 255 (preto). Vamos dar uma olhada em um dígito do conjunto de dados. Utilizando a função `imshow()` do Matplotlib, você só precisa pegar um vetor de característica de uma instância, remodelá-lo para um array de  $28 \times 28$  e exibi-lo:

```
%matplotlib inline  
import matplotlib  
import matplotlib.pyplot as plt  
  
some_digit = X[36000]  
some_digit_image = some_digit.reshape(28, 28)  
  
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,  
           interpolation="nearest")  
plt.axis("off")  
plt.show()
```



Isso parece um 5, e de fato é isso o que o rótulo nos diz:

```
>>> y[36000]  
5.0
```

A Figura 3-1 mostra algumas outras imagens do conjunto de dados MNIST para dar a você uma ideia da complexidade da tarefa de classificação.



*Figura 3-1. Alguns dígitos do conjunto de dados MNIST*

Mas espere! Você sempre deve criar um conjunto de teste e deixá-lo de lado antes de inspecionar os dados de perto. O conjunto de dados MNIST na verdade já está dividido em um conjunto de treinamento (as primeiras 60 mil imagens) e um conjunto de teste (as últimas 10 mil imagens):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Vamos embaralhar o conjunto de treinamento; isso garantirá que todos os subconjuntos da validação cruzada sejam semelhantes (assim você se certifica que nenhum subconjunto fique sem algum dígito). Além disso, alguns algoritmos de aprendizado são sensíveis à ordem das instâncias de treinamento, e, se obtiverem muitas instâncias similares, funcionarão mal. Embaralhar o conjunto de dados garante que isso não ocorra:<sup>2</sup>

```
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

---

<sup>2</sup> O embaralhamento em alguns contextos pode ser uma má ideia — por exemplo, se você estiver trabalhando em dados de séries temporais (como preços no mercado de ações ou condições climáticas). Exploraremos isso nos próximos capítulos.

## Treinando um Classificador Binário

Vamos simplificar o problema por ora e tentar apenas identificar um dígito — por exemplo, o número 5. Este “5-detector” será um exemplo de *classificador binário* capaz de distinguir apenas entre duas classes: 5 e não 5. Vamos criar os vetores-alvo para esta tarefa de classificação:

```
y_train_5 = (y_train == 5) # True for all 5s, False for all other digits.  
y_test_5 = (y_test == 5)
```

Certo, agora vamos escolher um classificador e treiná-lo. Um bom lugar para começar é com um classificador de *Gradiente Descendente Estocástico* (SGD, do inglês) com a utilização da classe `SGDClassifier` do Scikit-Learn. Este classificador tem a vantagem de conseguir lidar eficientemente com conjuntos de dados muito grandes. Isso se deve, em parte, ao fato de o SGD lidar independentemente com instâncias de treinamento, uma de cada vez (o que também torna o SGD adequado para o *aprendizado online*) como veremos mais adiante. Vamos criar um `SGDClassifier` e treiná-lo em todo o conjunto de treinamento:

```
from sklearn.linear_model import SGDClassifier  
  
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train_5)
```



O `SGDClassifier` depende da aleatoriedade durante o treinamento (daí o nome “estocástico”). Você deve definir o parâmetro `random_state` se desejar resultados reproduzíveis.

Agora, você pode utilizá-lo para detectar imagens do número 5:

```
>>> sgd_clf.predict([some_digit])  
array([ True], dtype=bool)
```

O classificador supõe que esta imagem representa um 5 (`True`). Parece que adivinhou corretamente neste caso! Agora, avaliaremos o desempenho desse modelo.

## Medições de Desempenho

Avaliar um classificador é muitas vezes significativamente mais complicado do que avaliar um regressor, portanto, passaremos uma grande parte deste capítulo neste tópico. Existem muitas medições de desempenho disponíveis, então pegue outro café e prepare-se para aprender muitos novos conceitos e siglas!

## Medição da Acurácia com a Utilização da Validação Cruzada

Uma boa maneira de avaliar um modelo é utilizar a validação cruzada, assim como você fez no Capítulo 2.

### Implementando a Validação Cruzada

Às vezes, você precisará ter mais controle sobre o processo de validação cruzada do que o Scikit-Learn pode fornecer. Nesses casos, você mesmo pode implementar a validação cruzada; na verdade, ela é bem descomplicada e simples de se entender. O código a seguir faz aproximadamente a mesma coisa que a função `cross_val_score()` do Scikit-Learn e imprime o mesmo resultado:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.9502, 0.96565 and 0.96495
```

A classe `StratifiedKFold` executa a amostragem estratificada (como explicado no Capítulo 2) para produzir subconjuntos que contêm uma proporção representativa de cada classe. Em cada iteração, o código cria um clone do classificador, treina esse clone nas partes de treinamento e faz previsões na parte do teste. Então, conta o número de previsões corretas e entrega a proporção das previsões corretas.

Utilizaremos a função `cross_val_score()` para avaliar o seu modelo `SGDClassifier` com a utilização da validação cruzada *K-fold* com três partes. Lembre-se de que a validação cruzada *K-fold* significa dividir o conjunto de treinamento em *K-folds* (neste caso, três), prever e avaliar as previsões em cada conjunto utilizando um modelo treinado em conjuntos restantes (ver Capítulo 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])
```

Muito bom! Acima de 95% de acurácia (taxa das previsões corretas) em todas as partes da validação cruzada? É incrível, não? Bem, antes que você fique muito animado, vejamos um classificador muito fraco que apenas classifica cada imagem na classe “não 5”:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

Você consegue adivinhar a acurácia deste modelo? Vamos descobrir:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 ,  0.90715,  0.9128 ])
```

Sim, tem mais de 90% de acurácia! Isso se dá simplesmente porque apenas cerca de 10% das imagens são “5”, então, se você sempre adivinhar que uma imagem *não* é um 5, você estará certo cerca de 90% das vezes. Ganharia até de Nostradamus.

Isso demonstra por que a acurácia para os classificadores geralmente não é a medida preferencial de desempenho, especialmente quando você estiver lidando com *conjuntos de dados assimétricos* (ou seja, quando algumas classes forem muito mais frequentes do que outras).

## Matriz de Confusão

Uma maneira de avaliar bem melhor o desempenho de um classificador é olhar para a *matriz de confusão*. A ideia geral é contar o número de vezes que as instâncias da classe A são classificadas como classe B. Por exemplo, para saber o número de vezes que o classificador confundiu imagens de “5” com “3”, você olharia na 5<sup>a</sup> linha e 3<sup>a</sup> coluna da matriz da confusão.

Para calcular a matriz de confusão, primeiro você precisa ter um conjunto de previsões para que possam ser comparadas com os alvos reais. Você pode fazer previsões no conjunto de testes, mas vamos deixá-lo intocado por enquanto (lembre-se de que você vai querer utilizar o conjunto de testes apenas no final do seu projeto, quando tiver um classificador pronto para lançar). Em vez disso, utilize a função `cross_val_predict()`:

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Assim como a função `cross_val_score()`, `cross_val_predict()` desempenha a validação cruzada *K-fold*, mas, em vez de retornar as pontuações da avaliação, ela retorna as previsões feitas em cada parte do teste. Isso significa que você obtém uma previsão limpa para cada instância no conjunto de treinamento (“limpa”, significando que a previsão é feita por um modelo que nunca viu os dados durante o treinamento).

Utilizando a função `confusion_matrix()`, agora você está pronto para obter a matriz de confusão. Apenas passe as classes-alvo (`y_train_5`) e as classes previstas (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
       [1077, 4344]])
```

Cada linha em uma matriz de confusão representa uma *classe real*, enquanto cada coluna representa uma *classe prevista*. A primeira linha desta matriz considera imagens não 5 (a *classe negativa*): 53.272 delas foram corretamente classificadas como não 5 (elas são chamadas de *verdadeiros negativos*), enquanto as restantes 1.307 foram erroneamente classificadas como 5 (*falsos positivos*). A segunda linha considera as imagens dos 5 (a *classe positiva*): 1.077 foram classificadas erroneamente como não 5 (*falso negativo*), enquanto as restantes 4.344 foram corretamente classificadas como 5 (*verdadeiros positivos*). Um classificador perfeito teria apenas verdadeiros positivos e verdadeiros negativos, então sua matriz de confusão teria valores diferentes de zero somente na sua diagonal principal (superior esquerda para inferior direita):

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
       [0, 5421]])
```

A matriz de confusão fornece muitas informações, mas às vezes pode ser que você prefira uma métrica mais concisa. Uma coisa interessante a ser observada é a acurácia das previsões positivas; que é chamada de *precisão* do classificador (Equação 3-1).

*Equação 3-1. Precisão*

$$\text{precisão} = \frac{TP}{TP + FP}$$

TP é o número de verdadeiros positivos, e FP é o número de falsos positivos.

Uma maneira trivial de ter uma precisão perfeita é fazer uma única previsão positiva e garantir que ela seja correta (precisão = 1/1 = 100%). Isso não seria muito útil uma vez que o classificador ignoraria todas, exceto uma instância positiva. Portanto, a precisão é utilizada em conjunto com outra métrica chamada *revocação*, também conhecida como *sensibilidade* ou *taxa de verdadeiros positivos* (*TPR*, do inglês): esta é a taxa de instâncias positivas que são corretamente detectadas pelo classificador (Equação 3-2).

### Equação 3-2. Revocação

$$\text{revocação} = \frac{TP}{TP + FN}$$

FN é, naturalmente, o número de falsos negativos.

Se você estiver perdido com a matriz de confusão, a Figura 3-2 pode ajudar.

		Previsto		
		Negativa	Positiva	
Real	Negativa	8 3 2	6	
	Positiva	5 5 5	5	
				Precisão (por exemplo, 3 de 4)
				Revocação (por exemplo, 3 de 5)
				TP
				FN
				FP

Figura 3-2. Uma matriz de confusão ilustrada

## Precisão e Revocação

O Scikit-Learn fornece várias funções para calcular métricas classificadoras, incluindo precisão e revocação:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1077)
0.80132816823464303
```

Agora, o seu 5-detector não parece tão brilhante como quando você analisou sua precisão. Quando ele afirma que uma imagem representa um 5, ele está correto apenas 77% das vezes. Além disso, ele só detecta 80% dos “5”.

Muitas vezes, é conveniente combinar precisão e revocação em uma única métrica chamada *pontuação F<sub>1</sub>*, principalmente se você precisar de uma forma simples de comparar dois classificadores. A pontuação F<sub>1</sub> é a *média harmônica* da precisão e revocação (Equação 3-3). Enquanto a média regular trata igualmente todos os valores, a média harmônica dá muito mais peso aos valores mais baixos. Como resultado, o classificador só obterá uma pontuação F<sub>1</sub> alta se a revocação e a precisão forem altas.

Equação 3-3. Pontuação  $F_1$

$$F_1 = \frac{2}{\frac{1}{\text{precisão}} + \frac{1}{\text{revocação}}} = 2 \times \frac{\text{precisão} \times \text{revocação}}{\text{precisão} + \text{revocação}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

Para calcular a pontuação  $F_1$  chame a função `f1_score()`:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.78468208092485547
```

A pontuação  $F_1$  favorece classificadores com precisão e revocação similares. Isso nem sempre é o que você quer: em alguns contextos, sua preocupação é principalmente com a precisão, e, em outros, você realmente se preocupa com a revocação. Por exemplo, se você treinou um classificador para detectar vídeos que são adequados para crianças, provavelmente preferiria um classificador que rejeitasse muitos bons vídeos (baixa revocação), mas mantivesse somente os adequados (alta precisão); em detrimento de um classificador que tivesse uma revocação bem mais alta, mas que permitiria que alguns vídeos realmente ruins aparecessem em seu produto (em tais casos, é necessário adicionar um pipeline humano para verificar a seleção do classificador de vídeo). Por outro lado, suponha que você treine um classificador para detectar bandidos nas imagens do sistema de segurança: é provavelmente tranquilo se seu classificador tiver uma precisão de apenas 30%, desde que tenha 99% de revocação (com certeza os guardas de segurança receberão alguns alertas falsos, mas quase todos os ladrões serão pegos).

Infelizmente, não é possível ter os dois: aumentar a precisão reduz a revocação, e vice-versa. Isso é chamado de *compensação da precisão/revocação*.

## Compensação da Precisão/Revocação

Para entender essa compensação, vejamos como o `SGDClassifier` toma suas decisões de classificação. Para cada instância, ele calcula uma pontuação baseada em uma *função de decisão* e, se essa pontuação for maior que um limiar, ele atribui a instância à classe positiva, ou então a atribui à classe negativa. A Figura 3-3 mostra alguns dígitos posicionados a partir da pontuação inferior na esquerda para a pontuação mais alta à direita. Suponha que o *limiar de decisão* seja posicionado na seta central (entre os dois 5): você encontrará 4 verdadeiros positivos (5 reais) à direita desse limiar, e um falso positivo (na verdade, um 6). Portanto, a precisão é de 80% (4 de 5) com esse limiar. Mas a cada 6 dos 5 verdadeiros, o classificador só detecta 4, então a revocação é de 67% (4 de 6). Agora, se você aumentar o limiar (mova-o para a seta à direita) o falso positivo (o 6) torna-se um verdadeiro negativo,

aumentando assim a precisão (até 100% neste caso), mas um verdadeiro positivo torna-se um falso negativo, diminuindo a revocação para 50%. Assim, diminuir o limiar aumenta a revocação e reduz a precisão.

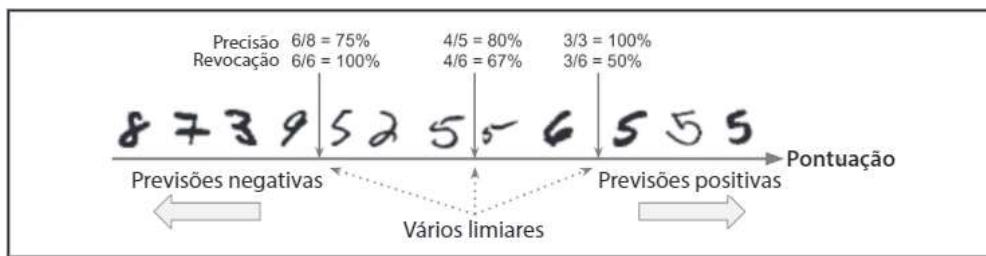


Figura 3-3. Limiar de decisão e compensação de precisão/revocação

O Scikit-Learn não permite que você defina o limiar diretamente, mas lhe dá acesso às pontuações de decisão que ele utiliza para fazer previsões. Em vez de chamar o método `predict()` do classificador, você pode chamar o método `decision_function()`, que retorna uma pontuação para cada instância e, em seguida, faz previsões com base nessas pontuações utilizando qualquer limiar desejado:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([-161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

O `SGDClassifier` utiliza um limiar igual a 0, então o código anterior retorna o mesmo resultado que o método `predict()` (isto é, `True`). Aumentaremos o limiar:

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

Isso confirma que aumentar o limiar diminui a revocação. A imagem realmente representa um 5, e o classificador o detecta quando o limiar é 0, mas o perde quando o limiar sobe para 200 mil.

Então, como você pode decidir qual limiar utilizar? Para isso, utilizando novamente a função `cross_val_predict()`, você precisará primeiro obter as pontuações de todas as instâncias no conjunto de treinamento, mas desta vez especificando que deseja que ela retorne as pontuações da decisão em vez das previsões:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Agora, utilizando a função `precision_recall_curve()` com essas pontuações, você pode calcular a precisão e a revocação para todos os limiares possíveis:

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finalmente, utilizando o Matplotlib (Figura 3-4), você pode plotar a precisão e a revocação como funções de valor do limiar:

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="center left")
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

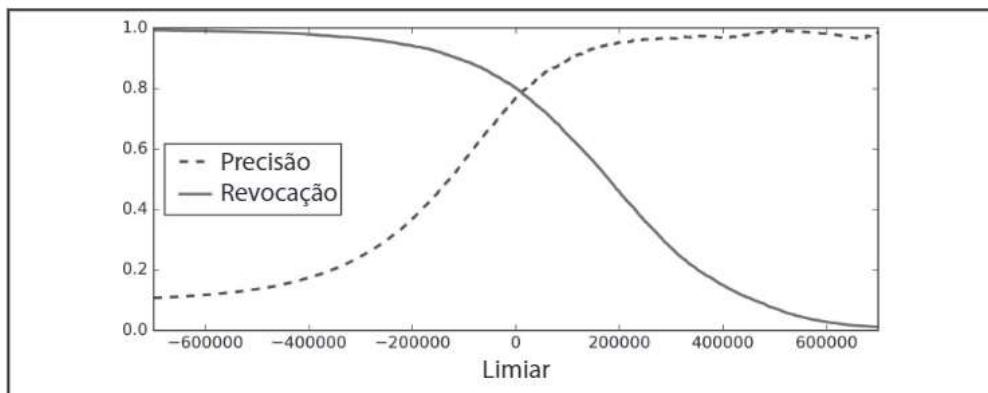


Figura 3-4. Precisão e revocação versus o limiar de decisão



Você pode se perguntar por que a curva de precisão é mais irregular do que a curva de revocação na Figura 3-4. A razão é que a precisão às vezes pode diminuir quando você aumenta o limiar (embora em geral ela aumente). Para entender o porquê, veja a Figura 3-3 e observe o que acontece quando você começa a partir do limiar central e o move apenas um dígito para a direita: a precisão passa de 4/5 (80%) para 3/4 (75%). Por outro lado, a revocação só diminuirá quando o limiar for aumentado, o que explica por que sua curva parece suave.

Agora, você pode selecionar o valor do limiar que lhe dá a melhor compensação de precisão/revocação para sua tarefa. Outra maneira de selecionar uma boa compensação de precisão/revocação é plotar a precisão diretamente contra a revocação, como mostrado na Figura 3-5.

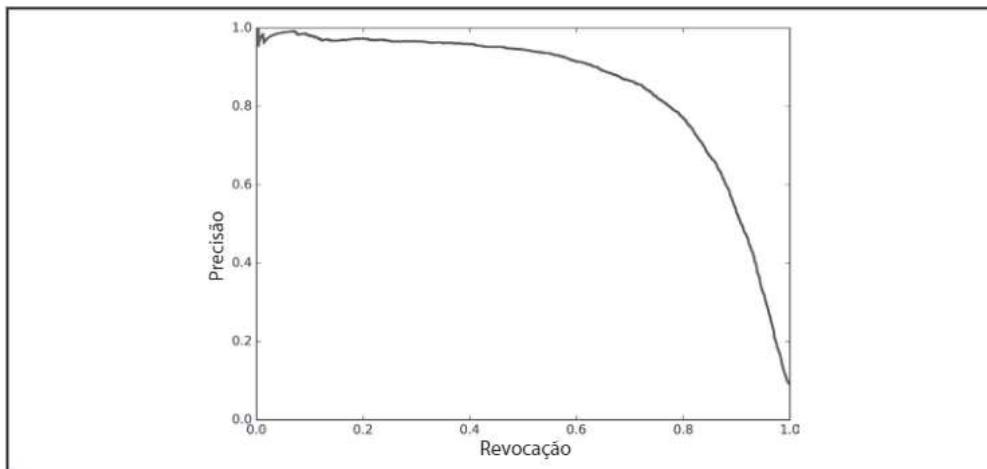


Figura 3-5. Precisão versus revocação

É possível ver que a precisão realmente começa a diminuir acentuadamente em torno de 80% de revocação. Provavelmente você selecionará uma compensação de precisão/revocação antes dessa queda — por exemplo, em torno de 60% de revocação. Mas, é claro, que a escolha depende do seu projeto.

Então, vamos supor que você almeja 90% de precisão. Você procura a primeira plotagem (aproximando um pouco) e descobre que precisa utilizar um limiar de cerca de 70 mil. Em vez de chamar o método `predict()` do classificador, você pode apenas executar este código para fazer previsões (por ora, no conjunto de treinamento):

```
y_train_pred_90 = (y_scores > 70000)
```

Verificaremos a precisão e a revocação dessas previsões:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.86592051164915484
>>> recall_score(y_train_5, y_train_pred_90)
0.69931746910164172
```

Ótimo, você tem um classificador com precisão de 90% (ou próximo o suficiente)! Como você pode ver, é muito fácil criar um classificador com praticamente qualquer precisão que desejar: basta definir um limiar alto o suficiente, e pronto. Mas, não vamos tão rápido. Um classificador de alta precisão não terá muita utilidade se a revocação dele for muito baixa!



Se alguém disser “vamos alcançar 99% de precisão”, você deve perguntar: “em qual revocação?”

## A Curva ROC

A curva das *características operacionais do receptor* (ROC, do inglês) é outra ferramenta comum utilizada com classificadores binários. É muito semelhante à curva de precisão/revocação, mas, em vez de plotar a precisão versus a revocação, a curva ROC plota a *taxa de verdadeiros positivos* (TPR, do inglês) (outro nome para revocação) versus a *taxa de falsos positivos* (FPR, do inglês). O FPR é a razão de instâncias negativas incorretamente classificadas como positivas. É igual a 1 menos a *taxa de verdadeiros negativos* (TNR, do inglês), que é a razão de instâncias negativas que são corretamente classificadas como negativas. A TNR também é chamada de *especificidade*. Portanto, a curva ROC plota a *sensibilidade* (revocação) versus  $1 - \text{especificidade}$ .

Para plotar a curva ROC primeiro você precisa calcular a TPR e FPR para vários valores de limiares usando a função `roc_curve()`:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

E, então, você pode plotar a curva FPR versus TPR usando o Matplotlib. Este código produz a plotagem na Figura 3-6:

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr, tpr)
plt.show()
```

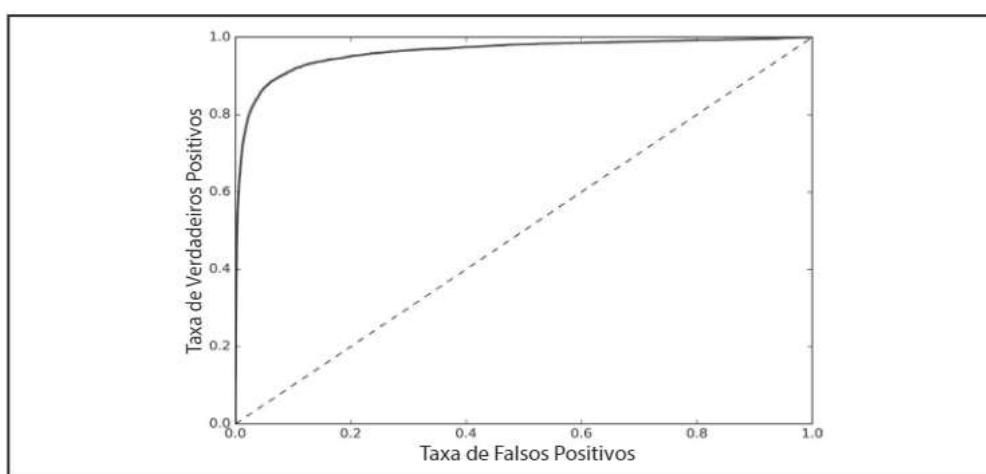


Figura 3-6. Curva ROC

Mais uma vez, existe uma compensação: quanto maior a revocação (TPR), mais falsos positivos (FPR) o classificador produz. A linha pontilhada representa a curva ROC de um classificador puramente aleatório; um bom classificador fica o mais distante dessa linha possível (em direção ao canto superior esquerdo).

Uma maneira de comparar classificadores é medir a *área abaixo da curva* (AUC, do inglês). Um classificador perfeito terá um ROC AUC igual a 1, enquanto um classificador puramente aleatório terá um ROC AUC igual a 0,5. O Scikit-Learn fornece uma função para calcular o ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.96244965559671547
```



Já que a curva ROC é tão semelhante à curva de precisão/revocação (PR), você pode se perguntar como decidir qual delas usar. Como regra geral, você deve preferir a curva PR sempre que a classe positiva for rara ou quando se preocupar mais com os falsos positivos do que com os falsos negativos, e a curva ROC em caso contrário. Por exemplo, olhando a curva ROC anterior (e a pontuação ROC AUC), você pode pensar que o classificador é realmente bom. Mas isso acontece porque existem alguns aspectos positivos (5) em comparação aos negativos (não 5). Em contraste, a curva PR deixa claro que o classificador pode melhorar (a curva poderia estar mais próxima do canto superior direito).

Vamos treinar um `RandomForestClassifier` e comparar sua curva ROC e a pontuação ROC AUC para o `SGDClassifier`. Primeiro, você precisa obter pontuações para cada instância no conjunto de treinamento. Mas, devido ao modo como funciona (veja o Capítulo 7), a classe `RandomForestClassifier` não possui um método `decision_function()`. Em vez disso, ela possui um método `predict_proba()`. Os classificadores do Scikit-Learn geralmente têm um ou outro. O método `predict_proba()` retorna um array que contém uma linha por instância e uma coluna por classe, cada uma contendo a probabilidade de a instância dada pertencer à classe dada (por exemplo, 70% de chance de a imagem representar um 5):

```
from sklearn.ensemble import RandomForestClassifier

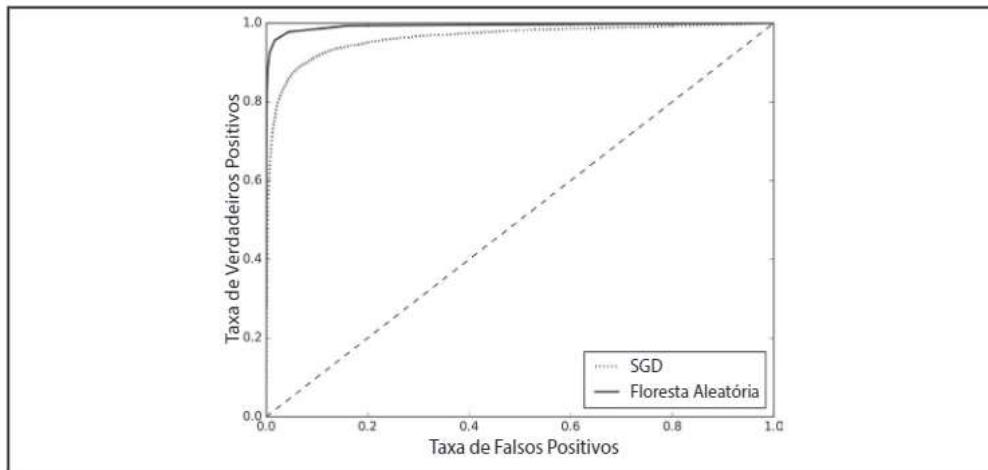
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                     method="predict_proba")
```

Mas, para plotar uma curva ROC, você precisa de pontuação, não probabilidades. Uma solução simples é utilizar a probabilidade da classe positiva como a pontuação:

```
y_scores_forest = y_probas_forest[:, 1] # pontuação = probabilidade de classe positiva
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Agora, você está pronto para plotar a curva ROC. É útil plotar a primeira curva ROC também para ver como elas se comparam (Figura 3-7):

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right")
plt.show()
```



*Figura 3-7. Comparando curvas ROC*

Como você pode ver na Figura 3-7, a curva ROC do `RandomForestClassifier` parece bem melhor que a do `SGDClassifier`: ela se aproxima muito do canto superior esquerdo. Como resultado, sua pontuação ROC AUC também é significativamente melhor:

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291
```

Tente medir as pontuações da precisão e da revocação: você deve encontrar 98,5% de precisão e 82,8% de revocação. Nada mal!

Provavelmente, agora você já sabe como treinar classificadores binários, escolher a métrica apropriada para sua tarefa, avaliar seus classificadores utilizando a validação cruzada, selecionar a compensação da precisão/revocação que corresponda às suas necessidades e comparar vários modelos utilizando as curvas ROC e pontuações ROC AUC. Agora, tentaremos detectar mais do que apenas os 5.

## Classificação Multiclasse

Enquanto os classificadores binários se distinguem entre duas classes os *classificadores multiclasses* (também chamados *classificadores multinomiais*) podem se distinguir entre mais de duas classes.

Alguns algoritmos (como os classificadores de Floresta Aleatória ou classificadores Naive-Bayes) são capazes de lidar diretamente com múltiplas classes. Outros (como os classificadores *Máquinas de Vetores de Suporte* ou classificadores Lineares) são estritamente binários. Entretanto, existem várias estratégias que você pode utilizar para realizar a classificação multiclasse com classificadores binários múltiplos.

Por exemplo, uma forma de criar um sistema que possa classificar as imagens numéricas em 10 classes (de 0 a 9) é treinar 10 classificadores binários, um para cada dígito (0-detector, 1-detector, 2-detector, e assim por diante). Então, quando você quiser classificar uma imagem, você obtém a pontuação de decisão de cada classificador para essa imagem e seleciona a classe cujo classificador produz a maior pontuação. Isso é chamado de estratégia *um contra todos* (OvA, do inglês) (também chamada *one-versus-the-rest*).

Outra estratégia é treinar um classificador binário para cada par de dígitos: um para distinguir 0s e 1s, outro para distinguir 0s e 2s, outro para 1s e 2s, e assim por diante. Isto é chamado de estratégia *um contra um* (OvO, do inglês). Se existirem  $N$  classes, você precisa treinar  $N \times (N - 1) / 2$  classificadores. O que significa treinar 45 classificadores binários para o problema do MNIST! Quando classificamos uma imagem, devemos rodá-la por todos os 45 classificadores e ver qual classe vence o maior número de duelos. A principal vantagem do OvO é que cada classificador precisa ser treinado somente para as duas classes que deve distinguir na parte do conjunto de treinamento.

Alguns algoritmos (como os classificadores *Máquinas de Vetores de Suporte*) escalam mal com o tamanho do conjunto de treinamento, então, para esses algoritmos, o OvO é melhor, pois é mais rápido treinar muitos classificadores em pequenos conjuntos do que treinar alguns classificadores em grandes conjuntos. Entretanto, OvA é o melhor para a maioria dos algoritmos de classificação binária.

O Scikit-Learn detecta quando você tenta utilizar um algoritmo de classificação binária para uma tarefa de classificação multiclasse, e automaticamente executa o OvA (exceto para classificadores SVM, para os quais ele utiliza OvO). Tentaremos isso com o `SGDClassifier`:

```
>>> sgd_clf.fit(X_train, y_train) # y_train, não y_train_5  
>>> sgd_clf.predict([some_digit])  
array([ 5.])
```

Essa foi fácil! Este código treina o `SGDClassifier` no conjunto de treinamento utilizando as classes-alvo originais de 0 a 9 (`y_train`) em vez das classes-alvo 5 contra todos (`y_train_5`). Então, ele faz uma previsão (correta, neste caso). Nos bastidores, o Scikit-Learn, na verdade, treinou 10 classificadores binários, obteve sua pontuação de decisão para a imagem e selecionou a classe com a maior pontuação.

Para ver que este é realmente o caso, chamaremos o método `decision_function()`. Em vez de retornar apenas uma pontuação por instância, ele agora retorna 10 pontuações, uma por classe:

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ -311402.62954431, -363517.28355739, -446449.5306454 ,
       -183226.61023518, -414337.15339485,  161855.74572176,
      -452576.39616343, -471957.14962573, -518542.33997148,
      -536774.63961222]])
```

A pontuação mais alta é realmente aquela que corresponde à classe 5:

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> sgd_clf.classes_[5]
5.0
```



Quando um classificador é treinado, ele armazena a lista das classes-alvo em seu atributo `classes_` em ordem de valor. Nesse caso, o índice de cada classe no array `classes_` combina convenientemente com a própria classe (por exemplo, a classe no índice 5 também é a classe 5), mas, no geral, você não terá tanta sorte.

Se você quiser forçar o Scikit-Learn a utilizar *um contra um* ou *um contra todos*, você pode utilizar a classe `OneVsOneClassifier` ou a `OneVsRestClassifier`. Crie uma instância e passe um classificador binário para seu construtor. Por exemplo, utilizando a estratégia OvO, com base no `SGDClassifier`, este código cria um classificador multiclasse:

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

Treinar um `RandomForestClassifier` é muito fácil:

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

Desta vez, o Scikit-Learn não precisou executar OvA ou OvO porque os classificadores da Floresta Aleatória podem classificar instâncias diretamente em múltiplas classes. Você pode chamar o `predict_proba()` para obter a lista de probabilidades que o classificador atribuiu a cada instância para cada classe:

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ,  0. ]])
```

Podemos ver que o classificador está bastante confiante sobre sua previsão: o 0,8 no 5º índice no array significa que o modelo estima uma probabilidade de 80% de a imagem representar um 5. Ele também acha que a imagem poderia ser um 0 ou um 3 (10% de chance cada).

Agora, é claro que você deve avaliar esses classificadores. Como de costume, utilize a validação cruzada. Vamos avaliar a precisão do `SGDClassifier` utilizando a função `cross_val_score()`:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187,  0.84899245,  0.86652998])
```

Ele consegue acima de 84% em todas as partes do teste. Se você utilizar um classificador aleatório, obterá 10% de precisão, então não é uma pontuação tão ruim, mas você ainda pode melhorar. Por exemplo, dimensionar as entradas aumenta a precisão em mais de 90% (como discutido no Capítulo 2):

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([ 0.91011798,  0.90874544,  0.906636  ])
```

## Análise de Erro

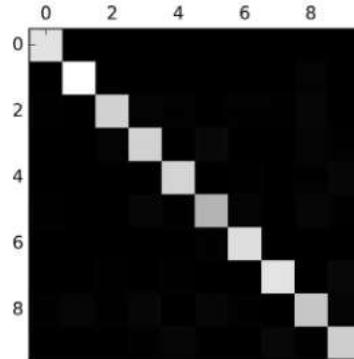
Claro, se este fosse um projeto real, você seguiria as etapas da sua lista de verificação do projeto de Aprendizado de Máquina (consulte o Apêndice B): explorar opções de preparação de dados, testar vários modelos, selecionar os melhores e ajustar seus hiperparâmetros utilizando o `GridSearchCV` e automatizando o máximo possível, como você fez no capítulo anterior. Aqui, assumiremos que você achou um modelo promissor e quer encontrar maneiras de melhorá-lo. Uma maneira de fazer isso é analisar os tipos de erros que ele comete.

Primeiro, olhe para a matriz de confusão. Você precisa fazer as previsões com a função `cross_val_predict()`, então chamar a função `confusion_matrix()`, assim como fez anteriormente:

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
>>> conf_mx = confusion_matrix(y_train, y_train_pred)
>>> conf_mx
array([[5725,     3,   24,    9,   10,   49,   50,   10,   39,    4],
       [  2, 6493,   43,   25,    7,   40,    5,   10,  109,    8],
       [ 51,   41, 5321,  104,   89,   26,   87,   60,  166,   13],
       [ 47,   46, 141, 5342,    1,  231,   40,   50,  141,   92],
       [ 19,   29,   41,   10, 5366,    9,   56,   37,   86,  189],
       [ 73,   45,   36,  193,   64, 4582,  111,   30,  193,   94],
       [ 29,   34,   44,    2,   42,   85, 5627,   10,   45,    0],
       [ 25,   24,   74,   32,   54,   12,    6, 5787,   15,  236],
       [ 52,  161,   73,  156,   10, 163,   61,   25, 5027,  123],
       [ 43,   35,   26,   92,  178,   28,    2,  223,   82, 5240]])
```

São muitos números. Utilizando a função `matshow()` do Matplotlib, será mais conveniente olhar para uma representação da imagem na matriz de confusão:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



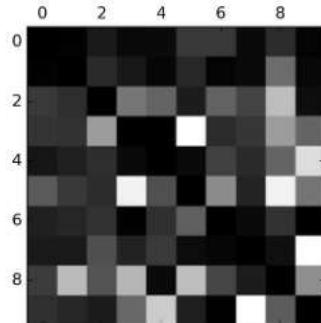
Essa matriz de confusão parece muito boa, já que a maioria das imagens está na diagonal principal, o que significa que foram classificadas corretamente. Os 5 parecem um pouco mais escuros do que os outros dígitos, o que poderia significar que existem menos imagens de 5 no conjunto de dados ou que o classificador não funcionou tão bem nos 5 quanto em outros dígitos. Na verdade, você pode comprovar que ambos são o caso.

Focaremos a plotagem nos erros. Primeiro, você precisa dividir cada valor na matriz de confusão pelo número de imagens na classe correspondente para que possa comparar as taxas de erro em vez do número absoluto de erros (o que tornaria as classes mais frequentes parecerem injustamente mal arrumadas):

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Agora, preencheremos a diagonal com zeros para manter apenas os erros e plotar o resultado:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



Aqui, você pode ver claramente os tipos de erros que o classificador comete. Lembre-se de que as linhas representam as classes reais, enquanto as colunas representam as classes previstas. As colunas para as classes 8 e 9 são bastante brilhantes, o que informa que muitas imagens são erroneamente classificadas como 8 ou 9. Da mesma forma, as linhas para as classes 8 e 9 também são muito brilhantes, informando que 8 e 9 geralmente são confundidos com outros dígitos. Por outro lado, algumas linhas são bastante escuras, como a linha 1: isso significa que a maioria dos 1 são classificados corretamente (poucos são confundidos com 8). Observe que os erros não são perfeitamente simétricos; por exemplo, há mais 5 classificados erroneamente como 8 do que o inverso.

Analizar a matriz de confusão lhe dá informações sobre maneiras de melhorar seu classificador. Olhando para esta plotagem, percebemos que seus esforços devem focar a melhoria da classificação de 8 e 9, bem como corrigir a confusão específica do 3/5. Por exemplo, você poderia tentar reunir mais dados de treinamento para esses dígitos. Ou criar novas características que ajudassem o classificador — por exemplo, escrevendo um algoritmo para contar o número de curvas fechadas (por exemplo, o 8 tem dois, o 6 tem um, 5 não tem nenhum). Ou você poderia pré-processar as imagens (utilizando, por exemplo, o Scikit-Image, Pillow ou OpenCV) para que alguns padrões se destaqueem mais, como os loops fechados.

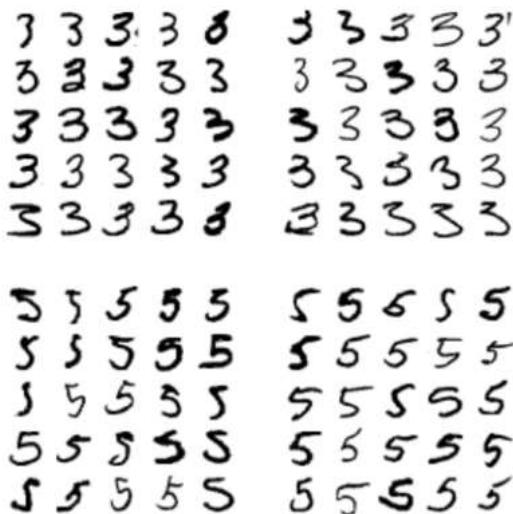
Analizar erros individuais também pode ser uma boa maneira de obter informações sobre o que o seu classificador está fazendo e por que ele está falhando, mas é mais difícil e demorado. Por exemplo, vamos plotar exemplos de 3 e 5 (a função `plot_digits()` utiliza apenas a função `imshow()` do Matplotlib; confira o notebook do Jupyter deste capítulo para mais detalhes):

```

cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()

```



Os dois blocos  $5 \times 5$  à esquerda mostram dígitos classificados como 3, e os dois blocos  $5 \times 5$  à direita mostram imagens classificadas como 5. Alguns dos dígitos que o classificador errou (ou seja, nos blocos inferior esquerdo e superior direito) estão tão mal escritos que mesmo um ser humano teria problemas para classificá-los (por exemplo, o 5 na 8<sup>a</sup> linha e 1<sup>a</sup> coluna realmente parece com um 3). No entanto, a maioria das imagens mal classificadas parecem erros óbvios para nós, e é difícil entender por que o classificador cometeu estes erros<sup>3</sup>. A razão é que usamos um `SGDClassifier` simples, que é um modelo linear. Tudo o que ele faz é atribuir um peso por classe a cada pixel, e, quando ele vê uma nova imagem, apenas resume as intensidades dos pixels analisados para obter uma pontuação para cada classe. Então, como os 3 e 5 diferem apenas em alguns pixels, esse modelo facilmente os confundirá.

---

<sup>3</sup> Lembre-se, porém, que nosso cérebro é um sistema de reconhecimento de padrões fantástico, e nosso sistema visual faz um pré-processamento complexo antes que qualquer informação chegue à nossa consciência, então o fato disso parecer simples não significa que de fato seja.

A principal diferença entre 3 e 5 é a posição da pequena linha que une a linha superior ao arco inferior. Se você desenhar um 3 com a junção ligeiramente deslocada para a esquerda, o classificador pode classificá-lo como um 5, e vice-versa. Em outras palavras, esse classificador é bastante sensível à mudança de imagem e sua rotação. Portanto, uma forma de reduzir a confusão do 3/5 seria pré-processar as imagens para garantir que elas estejam bem centradas e não rotacionadas demais. Isso provavelmente ajudará a reduzir outros erros também.

## Classificação Multilabel

Até agora, nós sempre atribuímos cada instância a apenas uma classe. Em alguns casos, pode ser que você queira que seu classificador emita várias classes para cada instância. Por exemplo, considere um classificador de reconhecimento facial: o que ele deve fazer se reconhecer várias pessoas na mesma imagem? Claro que ele deve anexar um rótulo por pessoa reconhecida. Digamos que o classificador foi treinado para reconhecer três faces: Alice, Bob e Charlie; então, quando mostramos uma imagem de Alice e Charlie, ele deve mostrar [1, 0, 1] (que significa "Alice sim, Bob não, Charlie sim"). Esse sistema de classificação que mostra vários rótulos binários é chamado de sistema de *classificação multilabel*.

Nós ainda não entraremos no reconhecimento facial, mas vejamos um exemplo mais simples apenas para fins de ilustração:

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Este código cria um array `y_multilabel` contendo dois rótulos-alvo para cada imagem numérica: o primeiro indica se o dígito é ou não é grande (7, 8 ou 9) e o segundo indica se é ou não é ímpar. As próximas linhas criam uma instância do `KNeighborsClassifier` (que aceita a classificação multilabel, mas nem todos os classificadores aceitam) utilizando o array de múltiplos destinos e nós a treinamos. Agora, você pode fazer uma previsão e perceber que ela mostra dois rótulos:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]], dtype=bool)
```

E ela acertou! O dígito 5 realmente não é grande (`False`) e é ímpar (`True`).

Há muitas maneiras de avaliar um classificador multilabel, e selecionar a métrica certa depende realmente do seu projeto. Por exemplo, uma abordagem seria medir a pontuação  $F_1$  para cada rótulo individual (ou qualquer outra métrica classificadora binária discutida anteriormente), então simplesmente calcular a pontuação média. Esse código calcula a pontuação média  $F_1$  em todos os rótulos:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.97709078477525002
```

Isso pressupõe que todos os rótulos são igualmente importantes, o que pode não ser o caso. Por exemplo, se você tem muito mais fotos de Alice do que de Bob ou Charlie, pode ser que você queira dar mais peso a fotos da Alice na pontuação do classificador. Uma opção simples é dar a cada rótulo um peso igual ao seu *suporte* (ou seja, o número de instâncias com aquele rótulo-alvo). Para fazer isso, basta definir `average="weighted"` no código anterior.<sup>4</sup>

## Classificação Multioutput

O último tipo de tarefa de classificação que discutiremos aqui é chamado de classificação *multioutput-multiclass* (ou simplesmente *classificação multioutput*). É simplesmente uma generalização da classificação multilabel em que cada rótulo pode ser multiclasse (ou seja, pode ter mais de dois valores possíveis).

Para ilustrar isso, construiremos um sistema que remova o ruído das imagens. Ele tomará como entrada uma imagem numérica ruidosa, e mostrará (espero) uma imagem numérica limpa, representada como um array de intensidade de pixels assim como as imagens MNIST. Observe que a saída do classificador é multilabel (um rótulo por pixel) e cada rótulo pode ter vários valores (a intensidade dos pixels varia de 0 a 255). É, portanto, um exemplo de um sistema de classificação multioutput.



Os limites entre classificação e regressão podem não ser muito claros, como é o caso descrito. Provavelmente, a previsão da intensidade do pixel é mais parecida com a regressão do que com a classificação. Além disso, os sistemas multioutput não se limitam às tarefas de classificação; é possível até ter um sistema que mostra vários rótulos por instância, incluindo rótulos de classe e de valor.

<sup>4</sup> O Scikit-Learn oferece algumas outras opções de média e métricas do classificador *multilabel*; veja a documentação para mais detalhes.

Começaremos criando os conjuntos de treinamento e testes pegando as imagens MNIST e adicionando ruído às suas intensidades de pixel por meio da função `randint()` do NumPy. As imagens-alvo serão as imagens originais:

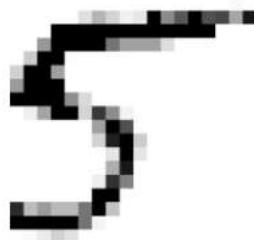
```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Vamos dar uma olhada em uma imagem do conjunto de testes (sim, estes são os dados de teste, então você deve estar franzindo a testa agora):



À esquerda temos a imagem confusa de entrada, e à direita está a imagem-alvo limpa. Agora, treinaremos o classificador para fazer com que ele limpe esta imagem:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



Parece bem próxima da imagem-alvo! Isso conclui nosso tour de classificação. Esperamos que agora você saiba como selecionar boas métricas para as tarefas de classificação, escolher a compensação apropriada da precisão/revocação, comparar classificadores e, no geral, criar bons sistemas de classificação para uma variedade de tarefas.

## Exercícios

1. Tente construir um classificador para o conjunto de dados do MNIST que obtenha uma acurácia acima de 97% no conjunto de teste. Dica: o `KNeighborsClassifier` funciona muito bem para esta tarefa; você só precisa encontrar bons valores do hiperparâmetro (experimente uma grid search nos pesos e hiperparâmetros `n_neighbors`).
2. Escreva uma função que possa mudar uma imagem do MNIST em qualquer direção (esquerda, direita, para cima ou para baixo) em um pixel.<sup>5</sup> Então, para cada imagem no conjunto de treinamento, crie quatro cópias deslocadas (uma por direção) e as adicione ao conjunto de treinamento. Finalmente, treine seu melhor modelo neste conjunto expandido de treinamento e meça sua acurácia no conjunto de teste. Você notará que seu modelo funciona ainda melhor agora! Esta técnica de crescimento artificial do conjunto de treinamento é chamada de *data augmentation* ou *expansão do conjunto de treinamento*.
3. Encare o conjunto de dados *Titanic*. Um ótimo lugar para começar é no Kaggle (<https://www.kaggle.com/c/titanic>).
4. Construa um classificador de spam (um exercício mais desafiador):
  - Baixe exemplos de spam e e-mails normais dos conjuntos públicos de dados do Apache SpamAssassin (<https://spamassassin.apache.org/old/publiccorpus/>);
  - Descompacte os conjuntos de dados e procure se familiarizar com o formato dos dados;
  - Divida os conjuntos de dados em um conjunto de treinamento e um conjunto de testes;
  - Escreva um pipeline de preparação de dados para converter cada e-mail em um vetor de características. Seu pipeline de preparação deve transformar um e-mail em um vetor (esparso) que indica a presença ou ausência de cada palavra possível. Por exemplo, se todos os e-mails contiverem apenas quatro palavras, “Hello”, “how”, “are”, “you”, então o e-mail “Hello you Hello Hello you” seria convertido em um vetor [1, 0, 0, 1] (significando que “Hello” está presente, “how” está ausente, “are” está ausente e “you” está presente), ou [3, 0, 0, 2] se você preferir contar o número de ocorrências de cada palavra;

---

<sup>5</sup> Você pode usar a função `shift()` do módulo `scipy.ndimage.interpolation`. Por exemplo, `shift(image, [2, 1], cval=0)` desloca a imagem 2 pixels para baixo e 1 pixel para a direita.

- Pode ser que você queira adicionar hiperparâmetros ao seu pipeline de preparação para controlar se deseja ou não retirar os cabeçalhos de e-mail, converter cada e-mail em minúsculas, remover pontuação, substituir todas as URLs por “URL”, substituir todos os números por “NUMBER” ou mesmo reduzir, ou seja, cortar as finalizações de palavras. Existem bibliotecas Python disponíveis para fazer isso;
- Em seguida, experimente vários e veja se consegue construir um bom classificador de spam com revocação e precisão altas.

As soluções para estes exercícios estão disponíveis online nos notebooks do Jupyter em <https://github.com/ageron/handson-ml>.

## Capítulo 4

# Treinando Modelos

Até agora, temos tratado os modelos de Aprendizado de Máquina e seus algoritmos basicamente como caixas pretas. Se você fez alguns dos exercícios nos capítulos anteriores, deve ter ficado surpreso com o quanto você consegue fazer sem nenhum conhecimento sobre o que acontece nos bastidores: você otimizou um sistema de regressão, melhorou um classificador de imagens de dígitos e até mesmo construiu um filtro de spam do zero, tudo isso sem saber como eles funcionam. De fato, em várias situações você não precisa realmente saber os detalhes da implementação.

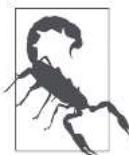
Entretanto, ter uma boa compreensão de como as coisas funcionam pode ajudá-lo a rapidamente focar o modelo apropriado, o algoritmo de treinamento correto a ser usado e um bom conjunto de hiperparâmetros para sua tarefa. Entender o que acontece nos bastidores também ajudará a depurar problemas e executar de forma mais eficiente a análise de erro. Por fim, a maioria dos tópicos discutidos neste capítulo será essencial para a compreensão, construção e treinamento das redes neurais (discutidas na Parte II deste livro).

Neste capítulo, começaremos avaliando o modelo de Regressão Linear, um dos modelos mais simples existentes. Discutiremos duas formas bem diferentes de treiná-lo:

- Utilizando uma equação direta de “forma fechada”, que calcula diretamente os parâmetros do modelo que melhor se encaixam no conjunto de treinamento (ou seja, os parâmetros do modelo que minimiza a função de custo em relação ao conjunto de treinamento);
- Utilizando uma abordagem iterativa de otimização chamada *Gradiente Descendente* (GD), que gradualmente pega os parâmetros-modelo para minimizar a função custo no conjunto de treinamento, às vezes convergindo-o para o mesmo conjunto de parâmetros do primeiro método. Abordaremos algumas variantes do Gradiente Descendente que utilizaremos muitas vezes quando estudarmos as redes neurais na Parte II: *Batch GD*, *Mini batch GD* e *Stochastic GD*.

Em seguida, abordaremos a Regressão Polinomial, um modelo mais complexo que pode se ajustar em conjuntos de dados não lineares. Como este modelo tem mais parâmetros do que a Regressão Linear ele é mais propenso ao sobreajuste dos dados de treinamento, então veremos como detectar se isso acontece ou não, utilizando curvas de aprendizado, e, depois, abordaremos técnicas de regularização que podem reduzir o risco de sobreajuste no conjunto.

Finalmente, estudaremos mais dois modelos que são comumente utilizados para as tarefas de classificação: Regressão Logística e Regressão Softmax.



Teremos algumas equações matemáticas neste capítulo que utilizam noções básicas de cálculo e álgebra linear. Para entender essas equações, você precisará saber o que são vetores e matrizes, como transpô-las, o que é o produto escalar, o que é a matriz inversa e o que são derivadas parciais. Se você não estiver familiarizado com esses conceitos, por favor acompanhe os tutoriais introdutórios de álgebra linear e cálculo disponíveis como notebooks do Jupyter no material suplementar online [em inglês]. Para aqueles verdadeiramente alérgicos à matemática, é recomendável ver este capítulo mesmo assim e pular as equações; espero que o texto seja suficiente para ajudar você a entender a maioria dos conceitos.

## Regressão Linear

No Capítulo 1, abordamos um modelo simples de regressão sobre a satisfação de vida:  $life\_satisfaction = \theta_0 + \theta_1 \times GDP\_per\_capita$ .

Este modelo é apenas uma função linear da característica de entrada `GDP_per_capita`.  $\theta_0$  e  $\theta_1$  são os parâmetros do modelo.

Geralmente, um modelo linear faz uma previsão calculando uma soma ponderada das características de entrada, mais uma constante chamada de *termo de polarização* (também chamada *coeficiente linear*), como mostrado na Equação 4-1.

*Equação 4-1. Previsão do modelo de Regressão Linear*

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$  é o valor previsto;
- $n$  é o número de características;
- $x_i$  é o valor da  $i$ -ésima característica;

- $\theta_j$  é o parâmetro do modelo  $j$  (incluindo o termo de polarização  $\theta_0$  e os pesos das características  $\theta_1, \theta_2, \dots, \theta_n$ ).

Isso pode ser escrito de maneira muito mais concisa usando uma forma vetorial, como mostrado na Equação 4-2.

*Equação 4-2. Previsão do modelo de Regressão Linear (forma vetorializada)*

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- $\theta$  é o *vetor de parâmetro* do modelo, que contém o termo de polarização  $\theta_0$  e os pesos das características  $\theta_1$  a  $\theta_n$ ;
- $\theta^T$  é a transposição de  $\theta$  (um vetor linha em vez de um vetor coluna);
- $\mathbf{x}$  é o *vetor de características* da instância, que contém  $x_0$  a  $x_n$ , com  $x_0$  sempre igual a 1;
- $\theta^T \cdot \mathbf{x}$  é o produto escalar de  $\theta^T$  e  $\mathbf{x}$ ;
- $h_{\theta}$  é a função de hipótese, que utiliza os parâmetros do modelo  $\theta$ .

Certo, este é o modelo de Regressão Linear, mas e agora, como o treinamos? Bem, lembre-se que treinar um modelo significa configurar seus parâmetros para que tenham o melhor ajuste no conjunto de treinamento. Para este propósito, é preciso primeiro de uma medida de quanto bem (ou mal) o modelo se adaptará aos dados de treinamento. No Capítulo 2, vimos que a medida de desempenho mais comum de um modelo de regressão é a Raiz do Erro Quadrático Médio (RMSE) (Equação 2-1). Portanto, para treinar um modelo de Regressão Linear, você deve encontrar o valor de  $\theta$  que minimize o RMSE. Na prática, é mais simples minimizar o Erro Quadrático Médio (MSE) do que a RMSE, e leva ao mesmo resultado (porque o valor que minimiza uma função também minimiza sua raiz quadrada).<sup>1</sup>

O MSE de uma hipótese  $h_{\theta}$  de Regressão Linear em um conjunto de treinamento  $X$  é calculado usando a Equação 4-3.

*Equação 4-3. Função MSE de custo para um modelo de Regressão Linear*

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

---

<sup>1</sup> Não raro um algoritmo de aprendizagem tenta otimizar uma função diferente da medida de desempenho usada para avaliar o modelo final. Em geral, isso se dá porque é mais fácil computar essa função, por possuir propriedades de diferenciação úteis que a medida de desempenho não tem ou por querermos restringir o modelo durante o treinamento, como veremos quando discutirmos sobre regularização.

A maioria destas notações foi apresentada no Capítulo 2 (veja “Notações” no Capítulo 2). A única diferença é que escrevemos  $h_\theta$  em vez de  $h$  para deixar claro que o modelo é parametrizado pelo vetor  $\theta$ . Para simplificar as anotações, escreveremos apenas  $MSE(\theta)$  em vez de  $MSE(X, h_\theta)$ .

## Método dos Mínimos Quadrados

Para encontrar o valor de  $\theta$ , que minimiza a função de custo, existe uma *solução de forma fechada* — em outras palavras, uma equação matemática que dá o resultado diretamente. Isto é chamado de *Método dos Mínimos Quadrados* (Equação 4-4).<sup>2</sup>

*Equação 4-4. Método dos Mínimos Quadrados*

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

- $\hat{\theta}$  é o valor de  $\theta$  que minimiza a função de custo.
- $y$  é o vetor dos valores do alvo contendo  $y^{(1)}$  a  $y^{(m)}$ .

Geraremos alguns dados de aparência linear para testar esta equação (Figura 4-1):

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

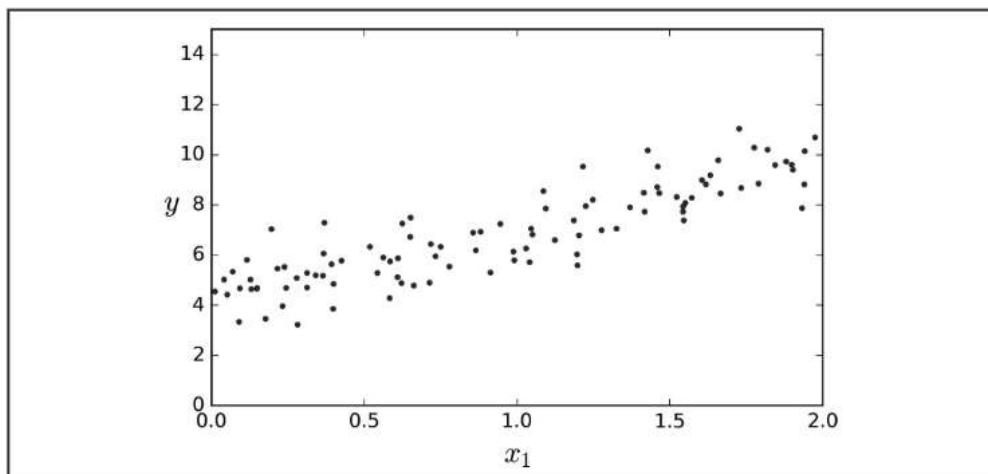


Figura 4-1. Conjunto de dados lineares gerado aleatoriamente

<sup>2</sup> A demonstração de que isso retorna o valor de  $\theta$  que minimiza a função de custo está fora do escopo deste livro.

Agora, calcularemos  $\hat{\theta}$  usando o Método dos Mínimos Quadrados. Utilizaremos a função `inv()` do módulo de Álgebra Linear do NumPy (`np.linalg`) para calcular o inverso de uma matriz, e o método `dot()` para a multiplicação da matriz:

```
X_b = np.c_[np.ones((100, 1)), X] # adiciona x0 = 1 a cada instância
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

A função vigente que usamos para gerar os dados é  $y = 4 + 3x_1 + \text{ruído gaussiano}$ . Vamos ver o que a equação encontrou:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Esperávamos por  $\theta_0 = 4$  e  $\theta_1 = 3$  em vez de  $\theta_0 = 4,215$  e  $\theta_1 = 2,770$ . Perto o suficiente, mas o ruído tornou impossível recuperar os parâmetros exatos da função original.

Agora, você pode fazer previsões utilizando  $\hat{\theta}$ :

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # adiciona x0 = 1 a cada instância
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Vamos plotar as previsões deste modelo (Figura 4-2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

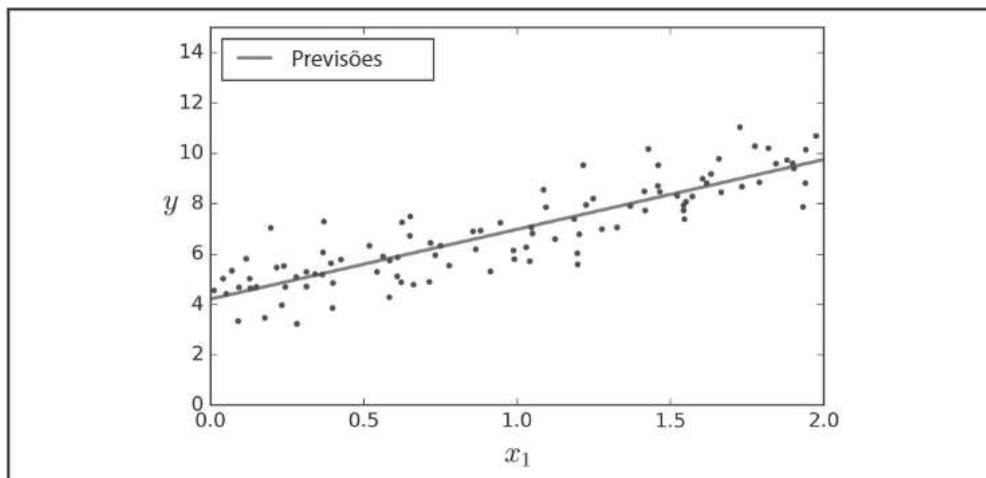


Figura 4-2. Previsões do modelo de Regressão Linear

O código equivalente utilizando o Scikit-Learn se parece com isso:<sup>3</sup>

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

## Complexidade Computacional

O Método dos Mínimos Quadrados calcula o inverso de  $X^T \cdot X$ , que é a matriz  $n \times n$  (em que  $n$  é o número de características). A *complexidade computacional* de inverter tal matriz é tipicamente de  $O(n^{2.4})$  a  $O(n^3)$  (dependendo da implementação). Em outras palavras, se dobrar o número de características, você multiplica o tempo de computação por aproximadamente  $2^{2.4} = 5,3$  para  $2^3 = 8$ .



O Método dos Mínimos Quadrados fica muito lento quando o número de características cresce muito (por exemplo, 100 mil).

Encarando pelo lado positivo, esta equação é linear em relação ao número de instâncias no conjunto de treinamento (ela é  $O(m)$ ), de modo que lida eficientemente com grandes conjuntos de treinamento, desde que possam caber na memória.

Além disso, ao treinar seu modelo de Regressão Linear (utilizando o Método dos Mínimos Quadrados ou qualquer outro algoritmo), suas previsões ficam muito rápidas: a complexidade computacional é linear no que se refere ao número de instâncias em que você deseja fazer previsões e ao número de características. Em outras palavras, fazer previsões em duas vezes mais instâncias (ou o dobro de recursos) levará aproximadamente o dobro do tempo.

Agora, veremos maneiras muito diferentes de treinar um modelo de Regressão Linear, mais adequado para casos em que haja um grande número de recursos ou muitas instâncias de treinamento para caber na memória.

---

<sup>3</sup> Observe que o Scikit-Learn separa o termo de polarização (`intercept_`) dos pesos da característica (`coef_`).

## Gradiente Descendente

*Gradiente Descendente* é um algoritmo de otimização muito genérico capaz de encontrar ótimas soluções para uma ampla gama de problemas. A ideia geral do Gradiente Descendente é ajustar iterativamente os parâmetros para minimizar uma função de custo.

Suponha que você esteja perdido nas montanhas em um denso nevoeiro; você só consegue sentir a inclinação do solo sob seus pés. Uma boa estratégia para chegar rapidamente ao fundo do vale é descer em direção à encosta mais íngreme. Isso é exatamente o que o Gradiente Descendente faz: ele mede o gradiente local da função de erro em relação ao vetor de parâmetro  $\theta$ , e vai na direção do gradiente descendente. Quando o gradiente for zero, você atingiu um mínimo!

Concretamente, você começa preenchendo  $\theta$  com valores aleatórios (isto é chamado *inicialização aleatória*), e então o melhora gradualmente, dando um pequeno passo por vez, cada passo tentando diminuir a função de custo (por exemplo, o MSE), até que o algoritmo *converja* para um mínimo (veja Figura 4-3).

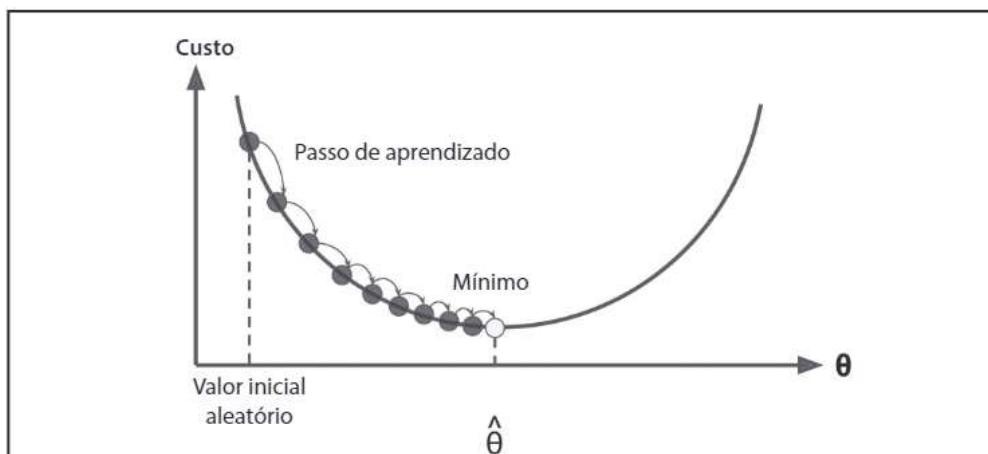


Figura 4-3. Gradiente Descendente

O tamanho dos passos é um parâmetro importante do Gradiente Descendente, determinado pelo hiperparâmetro *taxa de aprendizado*. Se a taxa de aprendizado for muito pequena, o algoritmo terá que passar por muitas iterações para convergir, o que demorará muito (veja a Figura 4-4).

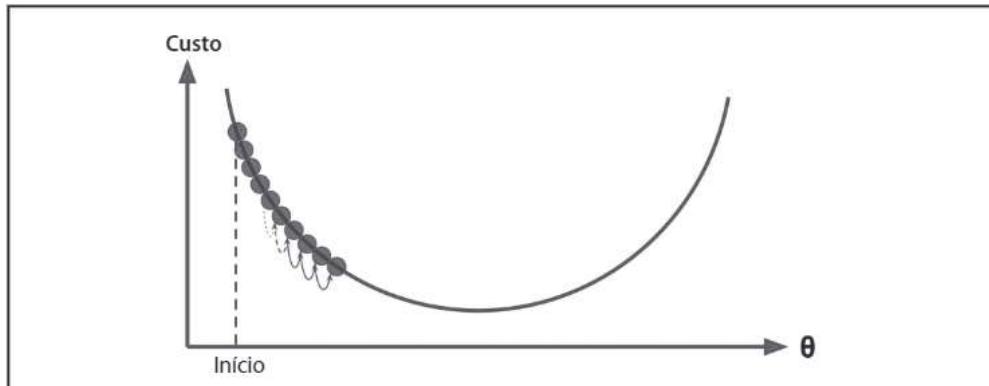


Figura 4-4. Taxa de aprendizado muito pequena

Por outro lado, se a taxa de aprendizado for muito alta, você pode atravessar o vale e acabar do outro lado, possivelmente até mais alto do que estava antes. Isso pode tornar o algoritmo divergente com valores cada vez maiores, não encontrando uma boa solução (veja a Figura 4-5).

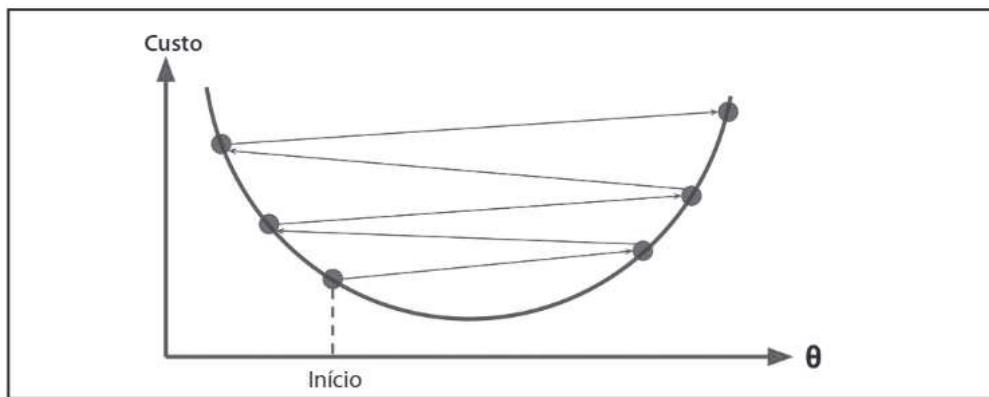


Figura 4-5. Taxa de aprendizado muito grande

Finalmente, nem todas as funções de custo se parecem com tigelas regulares. Podem haver buracos, cumes, planaltos e todo tipo irregular de terreno fazendo com que a convergência ao mínimo seja muito difícil. A Figura 4-6 mostra os dois principais desafios do Gradiente Descendente: se a inicialização aleatória iniciar o algoritmo à esquerda, ela converge para um *mínimo local*, o que não é tão bom quanto o *mínimo global*. Se começar pela direita, então demorará muito para atravessar o planalto e, se você parar cedo demais, nunca alcançará o mínimo global.

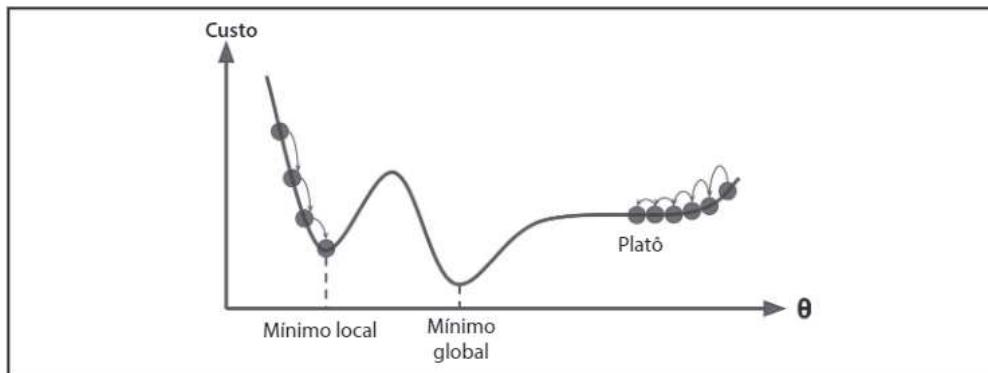


Figura 4-6. Armadilhas do Gradiente Descendente

Felizmente, a função de custo do MSE para um modelo de Regressão Linear é uma *função convexa*, o que significa que, se você escolher quaisquer dois pontos na curva, o segmento de linha que os une nunca a cruza. Isso implica que não há mínimos locais, apenas um mínimo global. É também uma função contínua com uma inclinação que nunca se altera abruptamente.<sup>4</sup> Estes dois fatos geram uma ótima consequência: o Gradiente Descendente tem a garantia de se aproximar arbitrariamente do mínimo global (se você esperar o tempo suficiente e se a taxa de aprendizado não for muito alta).

Na verdade, a função de custo tem a forma de uma tigela, mas, se as características tiverem escalas muito diferentes, pode ter a forma de uma tigela alongada. A Figura 4-7 mostra o Gradiente Descendente em um conjunto de treinamento em que as características 1 e 2 têm a mesma escala (à esquerda) e em um conjunto de treinamento em que a característica 1 tem valores muito menores do que a característica 2 (à direita).<sup>5</sup>

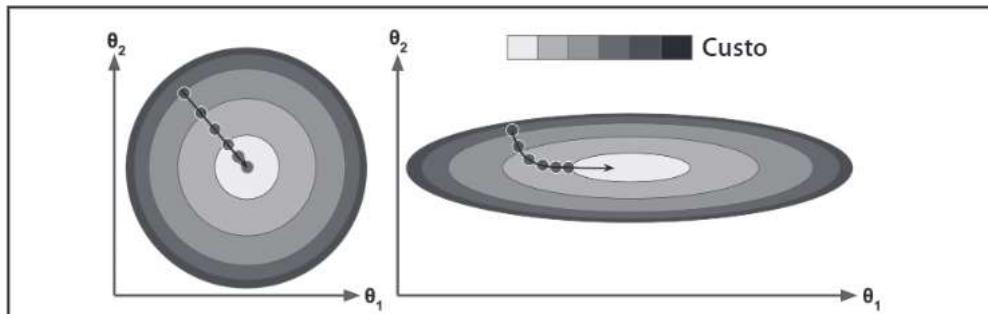


Figura 4-7. Gradiente Descendente com e sem escalonamento das características

<sup>4</sup> Tecnicamente, sua derivada é a *Lipschitz contínua*.

<sup>5</sup> Uma vez que a característica 1 é menor, é preciso uma mudança maior em  $\theta_1$  para afetar a função de custo, razão pela qual a tigela é alongada ao longo do eixo  $\theta_1$ .

Como você pode ver à esquerda, o algoritmo do Gradiente Descendente vai diretamente para o mínimo, atingindo-o rapidamente, enquanto à direita primeiro avança em sentido quase ortogonal em direção ao mínimo global e termina com uma longa marcha em um vale quase plano. Eventualmente ele alcançará o mínimo, mas demorará muito.



Ao utilizar o Gradiente Descendente, você deve garantir que todas as características tenham uma escala similar (por exemplo, utilizando a classe `StandardScaler` do Scikit-Learn), ou então demorará muito mais para convergir.

Este diagrama também ilustra o fato de que treinar um modelo significa procurar uma combinação de parâmetros do modelo que minimizem uma função de custo (em relação ao conjunto de treinamento). É uma pesquisa no *espaço de parâmetro* do modelo: quanto mais parâmetros um modelo possui, mais dimensões este espaço tem, e mais difícil será a busca: procurar por uma agulha em um palheiro de 300 dimensões é muito mais complicado do que em três dimensões. Felizmente, como no caso da Regressão Linear a função de custo é convexa, a agulha simplesmente está na parte inferior da tigela.

## Gradiente Descendente em Lote

Para implementar o Gradiente Descendente, é preciso calcular o gradiente da função de custo em relação a cada parâmetro do modelo  $\theta_j$ . Em outras palavras, você precisa calcular quanto mudará a função de custo se você modificar somente um pouco do  $\theta_j$ . Isto é chamado *derivada parcial*. É como perguntar: “qual é a inclinação de uma montanha sob meus pés se eu apontar para o leste?”, e, então, fazer a mesma pergunta apontando para o norte (e assim por diante para todas as outras dimensões, se puder imaginar um universo com mais de três). A Equação 4-5 calcula a derivada parcial da função de custo em relação ao parâmetro  $\theta_j$ , notado por  $\frac{\partial}{\partial \theta_j} \text{MSE}(\theta)$ .

*Equação 4-5. Derivadas parciais da função de custo*

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Em vez de calcular individualmente essas derivadas parciais, você pode utilizar a Equação 4-6 para calculá-las de uma só vez. O vetor gradiente, descrito  $\nabla_{\theta} \text{MSE}(\theta)$ , contém todas as derivadas parciais da função de custo (uma para cada parâmetro do modelo).

*Equação 4-6. Vetor gradiente da função de custo*

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$



Observe que esta fórmula envolve cálculos em cada etapa do Gradiente Descendente sobre o conjunto completo de treinamento  $\mathbf{X}$ ! É por isso que o algoritmo é chamado de *Gradiente Descendente em Lote*: ele utiliza todo o lote de dados em cada etapa. Como resultado, ele é terrivelmente lento para grandes conjuntos (em breve veremos algoritmos de gradiente descendente muito mais rápidos). No entanto, o Gradiente Descendente se dimensiona bem com a quantidade de características; treinar um modelo de Regressão Linear, quando há centenas de milhares de características, será muito mais rápido se utilizarmos o Gradiente Descendente do que se utilizarmos o Método dos Mínimos Quadrados.

Uma vez que você tenha o vetor gradiente, que aponta para cima, basta ir na direção oposta para descer. Isto significa subtrair  $\nabla_{\theta} \text{MSE}(\theta)$  de  $\theta$ . É aqui que a taxa de aprendizado  $\eta$  entra em cena:<sup>6</sup> multiplicar o vetor gradiente por  $\eta$  para determinar o tamanho do passo para baixo (Equação 4-7).

*Equação 4-7. Passo do Gradiente Descendente*

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Vejamos uma implementação rápida desse algoritmo:

```
eta = 0.1 # taxa de aprendizado
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # inicialização aleatória

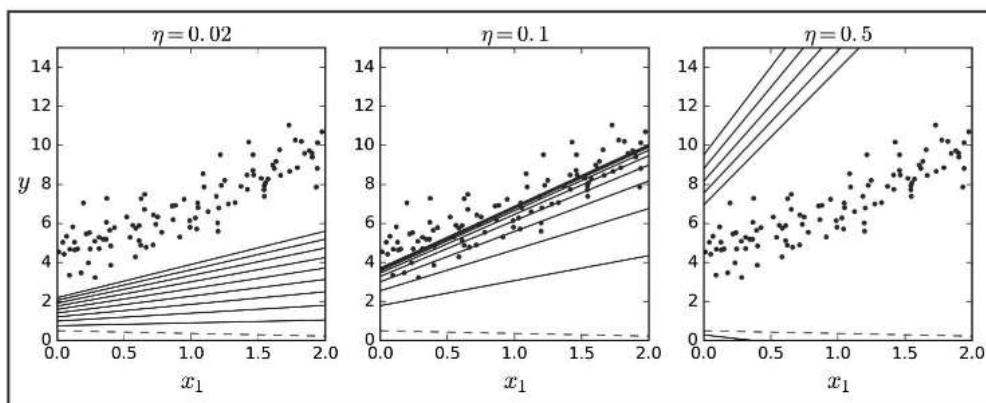
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

<sup>6</sup> Eta ( $\eta$ ) é a 7ª letra do alfabeto Grego.

Isto não foi muito difícil! Observe o `theta` resultante:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

É exatamente isso que o Método dos Mínimos Quadrados encontrou! O Gradiente Descendente funcionou perfeitamente. Mas, e se você tivesse utilizado uma taxa de aprendizado `eta` diferente? Utilizando três taxas de aprendizado diferentes, a Figura 4-8 mostra os primeiros 10 passos do Gradiente Descendente (a linha tracejada representa o ponto inicial).



*Figura 4-8. O Gradiente Descendente com várias taxas de aprendizado*

A taxa de aprendizado à esquerda é muito baixa: o algoritmo acabará por alcançar a solução, mas demorará muito. No meio, a taxa de aprendizado parece muito boa: com apenas algumas iterações já convergiu para a solução. À direita, a taxa de aprendizado é muito alta: o algoritmo diverge, pulando por todo o lado e, de fato, ficando cada vez mais longe da solução.

Você pode utilizar a grid search para encontrar uma boa taxa de aprendizado (veja o Capítulo 2). No entanto, talvez seja melhor limitar o número de iterações para que a grid search possa eliminar modelos que demoram muito para convergir.

Você deve estar se perguntando como é possível configurar o número de iterações. Se esse número for muito baixo, você ainda estará longe da solução ideal quando o algoritmo parar, mas, se for muito alto, perderá tempo quando os parâmetros do modelo não mudarem mais. Uma solução simples é definir um grande número de iterações para interromper o algoritmo quando o vetor do gradiente se tornar pequeno, ou seja, quando sua norma se tornar menor do que o minúsculo número  $\epsilon$  (chamado *tolerância*), porque isso acontece quando o Gradiente Descendente (quase) atinge o mínimo.

## Taxa de Convergência

Quando a função de custo for convexa e sua inclinação não mudar abruptamente (como é o caso da função de custo MSE), o Gradiente Descendente em Lote com uma taxa fixa de aprendizado convergirá para a solução ideal, mas talvez seja necessário esperar um pouco: pode levar  $O(1/\epsilon)$  iterações para atingir o melhor dentro de um intervalo de  $\epsilon$  dependendo da forma da função de custo. Se você dividir a tolerância por 10 para ter uma solução mais precisa, o algoritmo terá que correr cerca de 10 vezes mais.

## Gradiente Descendente Estocástico

O principal problema com o Gradiente Descendente em Lote é o fato de que ele utiliza todo o conjunto de treinamento para calcular os gradientes em cada passo, o que o torna muito lento quando o conjunto for grande. No extremo oposto, o *Gradiente Descendente Estocástico* (SGD) escolhe uma instância aleatória no conjunto de treinamento em cada etapa e calcula os gradientes baseado apenas nesta única instância. Obviamente, isso torna o algoritmo muito mais rápido, pois tem poucos dados para manipular em cada iteração. Também permite treinar em grandes conjuntos de treinamento, uma vez que apenas uma instância precisa estar na memória a cada iteração (o SGD pode ser implementado como um algoritmo out-of-core.<sup>7</sup>)

Por outro lado, devido a sua natureza estocástica (ou seja, aleatória), esse algoritmo é bem menos regular do que o Gradiente Descendente em Lote: em vez de diminuir suavemente até atingir o mínimo, a função de custo vai subir e descer, diminuindo apenas na média. Ao longo do tempo, ele acabará muito perto do mínimo, mas, ao chegar lá, ele continuará a rebater, nunca se estabilizando (veja a Figura 4-9). Assim, quando o algoritmo para, os valores dos parâmetros finais serão bons, mas não ótimos.

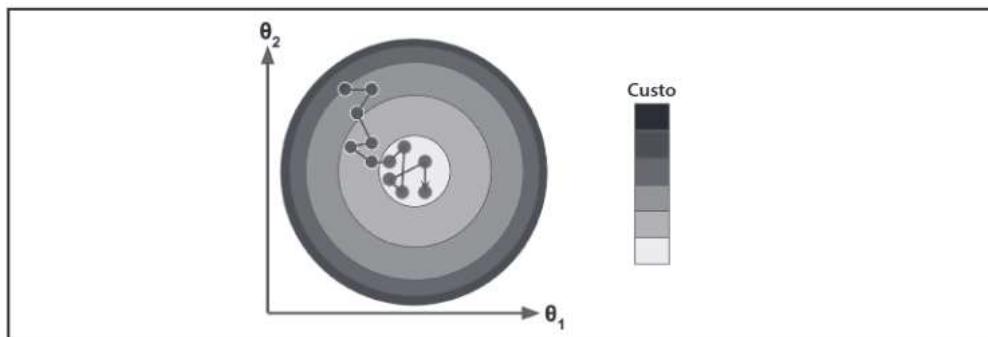


Figura 4-9. Gradiente Descendente Estocástico

<sup>7</sup> Algoritmos out-of-core são discutidos no Capítulo 1.

Quando a função de custo é muito irregular (como na Figura 4-6), isso pode, na verdade, ajudar o algoritmo a pular fora do mínimo local, de modo que o Gradiente Descendente Estocástico terá uma chance maior de encontrar o mínimo global do que o Gradiente Descendente em Lote.

Desta forma, a aleatoriedade é boa para escapar do ótimo local, mas ruim porque significa que o algoritmo nunca pode se estabelecer no mínimo. Uma solução para este dilema é reduzir gradualmente a taxa de aprendizado. As etapas começam grandes (o que ajuda a fazer rápidos progressos e a escapar dos mínimos locais) e depois diminuem, permitindo que o algoritmo se estabeleça no mínimo global. Este processo é chamado *recozimento simulado*, porque se assemelha ao processo de recozimento na metalurgia no qual o metal fundido é resfriado lentamente. A função que determina a taxa de aprendizado em cada iteração é chamada de *cronograma de aprendizado*. Se a taxa de aprendizado for reduzida rapidamente, você pode ficar preso em um mínimo local, ou mesmo acabar congelado a meio caminho do mínimo. Se a taxa de aprendizado for reduzida lentamente, você pode saltar em torno do mínimo por um longo período de tempo e acabar com uma solução insuficiente se parar de treinar muito cedo.

Utilizando um simples cronograma de aprendizado, este código implementa o Gradiente Descendente Estocástico:

```
n_epochs = 50
t0, t1 = 5, 50 # hiperparâmetros de aprendizado

def learning_schedule(t):
    return t0 / (t + t1)

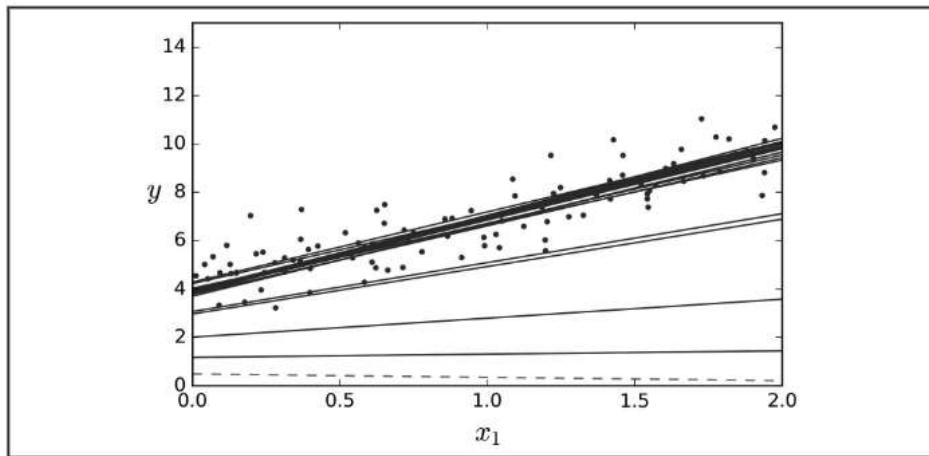
theta = np.random.randn(2,1) # inicialização aleatória

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

Por convenção, replicamos por rodadas de  $m$  iterações; cada rodada é chamada de *época*. Enquanto o código Gradiente Descendente em Lote replica mil vezes pelo conjunto de treinamento, este código passa apenas 50 vezes pelo conjunto e atinge uma boa solução:

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

A Figura 4-10 mostra os primeiros 10 passos do treinamento (repare na irregularidade deles).



*Figura 4-10. Os primeiros 10 passos do Gradiente Descendente Estocástico*

Observe que, como as instâncias são escolhidas aleatoriamente, algumas poderão ser escolhidas várias vezes por época, enquanto outras podem nem ser escolhidas. Outra abordagem seria minimizar o conjunto de treinamento, passar instância por instância, embaralhar novamente, e assim por diante, se quiser ter certeza de que o algoritmo passará em cada época por todas as instâncias. No entanto, isso geralmente diminuirá a convergência.

Para executar a Regressão Linear usando SGD com Scikit-Learn, você pode aplicar a classe `SGDRegressor`, que otimiza o padrão da função de custo de erro quadrático. O código a seguir roda 50 épocas, iniciando com uma taxa de aprendizado de 0,1 (`eta0=0.1`), usando o cronograma padrão de aprendizado (diferente do anterior) e dispensando qualquer regularização (`penalty=None`; mais detalhes em breve):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

Mais uma vez, você encontra uma solução muito próxima da retornada pelo Método dos Mínimos Quadrados:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.16782089]), array([ 2.72603052]))
```

## Gradiente Descendente em Minilotes

O último algoritmo do Gradiente Descendente que analisaremos é denominado *Gradiente Descendente em Minilotes*. É fácil entender esse algoritmo quando se sabe Gradiente Descendente em Lotes e Estocástico: em vez de calcular os gradientes a cada etapa com base no conjunto completo de treinamento (como o GD em Lotes) ou com base em apenas uma instância (como em GD Estocástico), o GD em Minilotes faz o cálculo dos gradientes em pequenos conjuntos aleatórios de instâncias chamados de *minilotes*. A principal vantagem do GD em Minilotes em relação ao GD Estocástico é que você pode obter um ganho de desempenho na otimização de hardware das operações da matriz, especialmente quando são utilizadas GPUs.

O progresso do algoritmo no espaço dos parâmetros é menos errático do que com o SGD, especialmente com minilotes bem grandes. Como resultado, o GD em Minilotes ficará um pouco mais perto do mínimo que o SGD. Mas, por outro lado, pode ser mais difícil escapar do mínimo local (no caso dos problemas que sofrem do mínimo local, ao contrário da Regressão Linear, como vimos anteriormente). A Figura 4-11 mostra os caminhos seguidos pelos três algoritmos do Gradiente Descendente no espaço do parâmetro durante o treinamento. Todos terminam perto do mínimo, mas o caminho do GD em Lote realmente para no mínimo, enquanto tanto o GD Estocástico como o GD Minilote continuam a caminhar. No entanto, não se esqueça de que o GD em Lote demora muito para dar cada passo e o GD Estocástico e o GD Minilote também atingiram o mínimo se você utilizasse uma boa programação de aprendizado.

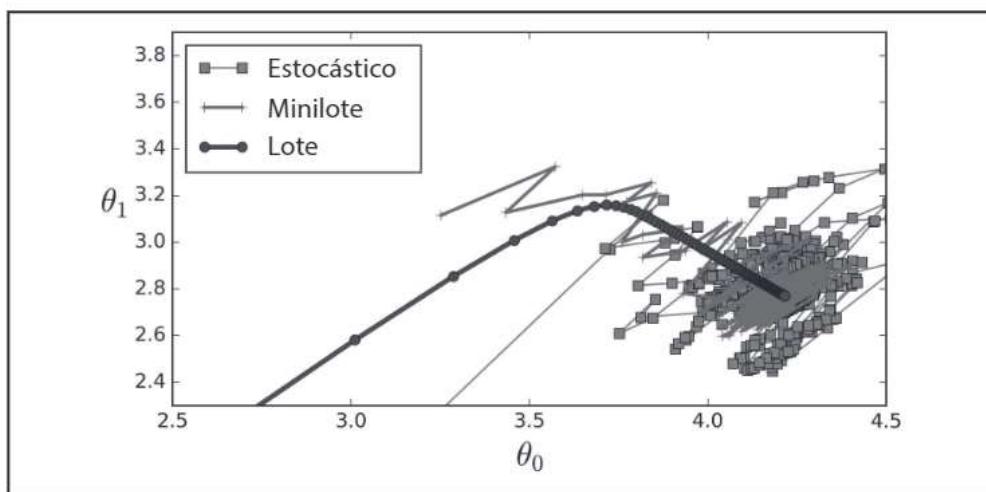


Figura 4-11. Caminhos do Gradiente Descendente no espaço de parâmetro

Compararemos os algoritmos que discutimos até agora para a Regressão Linear<sup>8</sup> (lembre-se de que  $m$  é o número de instâncias de treinamento e  $n$  é o número de características); veja a Tabela 4-1.

*Tabela 4-1. Comparação de algoritmos para a Regressão Linear.*

Algoritmo	Grande $m$	Suporte out-of-core	Grande $n$	Hiperparâmetros	Escalonamento requerido	Scikit-Learn
Método dos Mínimos Quadrados	Rápido	Não	Lento	0	Não	LinearRegression
GD Lote	Lento	Não	Rápido	2	Sim	n/a
GD Estocástico	Rápido	Sim	Rápido	$\geq 2$	Sim	SGDRegressor
GD Minilote	Rápido	Sim	Rápido	$\geq 2$	Sim	SGDRegressor



Quase não há diferença após o treinamento: todos estes algoritmos acabam com modelos muito semelhantes e fazem previsões exatamente da mesma maneira.

## Regressão Polinomial

E se seus dados forem realmente mais complexos do que uma simples linha reta? Surpreendentemente, você pode utilizar um modelo linear para acomodar dados não lineares. Uma maneira simples de fazer isso é acrescentar potências de cada característica como novas características e, em seguida, treinar um modelo linear neste conjunto estendido de características. Esta técnica é chamada de *Regressão Polinomial*.

Vejamos um exemplo. Primeiro, vamos gerar algum dado não linear baseado em uma simples *equação quadrática*<sup>9</sup> (mais algum ruído; veja a Figura 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

<sup>8</sup> Enquanto o Método dos Mínimos Quadrados só pode executar a Regressão Linear, os algoritmos de Gradiente Descendente podem ser usados para treinar muitos outros modelos, como veremos.

<sup>9</sup> Uma equação quadrática é da forma  $y = ax^2 + bx + c$ .

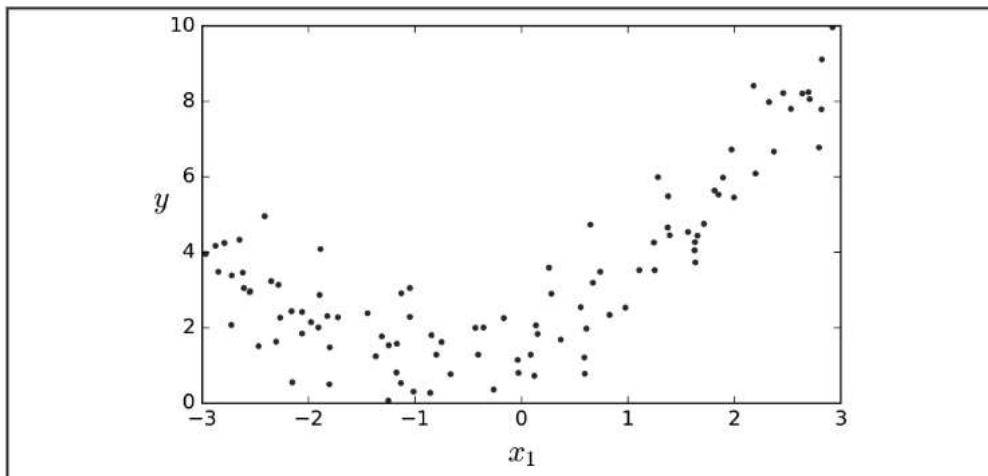


Figura 4-12. Conjunto gerado de dados não lineares e ruidosos

Claramente, uma linha reta nunca acomodará esses dados corretamente. Então, utilizaremos a classe `PolynomialFeatures` do Scikit-Learn para transformar nossos dados de treinamento adicionando o quadrado (polinômio de 2º grau) de cada característica como novas no conjunto de treinamento (neste caso, há apenas uma característica):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

`X_poly` agora contém a característica original de `X` mais o quadrado desta característica. Agora, você pode acomodar um modelo `LinearRegression` a estes dados estendidos de treinamento (Figura 4-13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```

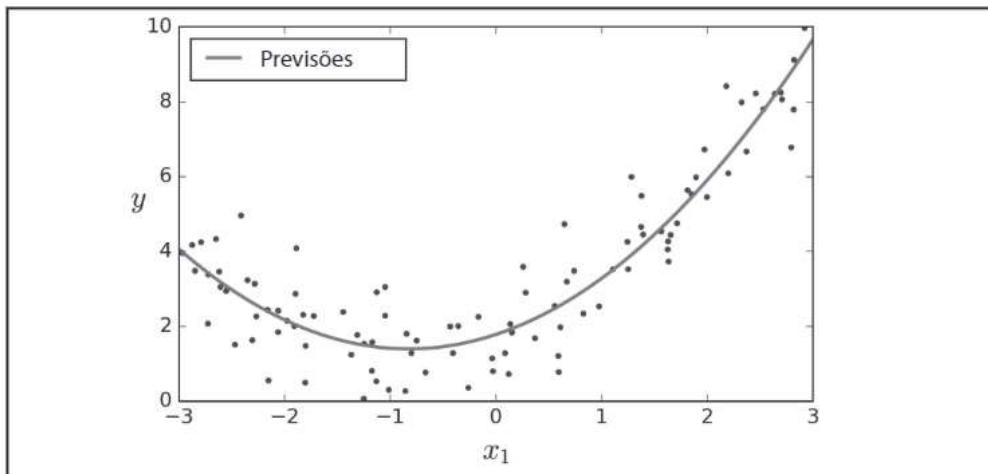


Figura 4-13. Previsões do modelo de Regressão Polinomial

Nada mal: o modelo estima  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$  quando na verdade a função original era  $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Ruído Gaussiano}$ .

Observe que, quando existirem várias características, a Regressão Polinomial é capaz de encontrar relações entre elas (algo que um modelo simples de Regressão Linear não pode fazer). Isto é possível porque a `PolynomialFeatures` também adiciona todas as combinações de características até o grau fornecido. Por exemplo, se houvesse duas características `PolynomialFeatures`,  $a$  e  $b$ , com `degree=3`, elas não apenas adicionariam as características  $a^2$ ,  $a^3$ ,  $b^2$ , e  $b^3$ , mas também as combinações  $ab$ ,  $a^2b$  e  $ab^2$ .



`PolynomialFeatures(degree=d)` transforma um array que contém  $n$  características em um que contém  $\frac{(n+d)!}{d!n!}$  características, sendo  $n!$  o fatorial de  $n$ , igual a  $1 \times 2 \times 3 \times \dots \times n$ . Cuidado com a explosão combinatória do número de características!

## Curvas de Aprendizado

Ao executar a Regressão Polinomial de alto grau, provavelmente você acomodará os dados de treinamento muito melhor do que com a Regressão Linear simples. Por exemplo, a Figura 4-14 aplica um modelo polinomial de 300 graus aos dados de treinamento anteriores e compara o resultado com um modelo linear puro e um modelo quadrático (polinômio de 2º grau). Observe como o modelo polinomial de 300 graus se move para se aproximar o máximo possível das instâncias de treinamento.

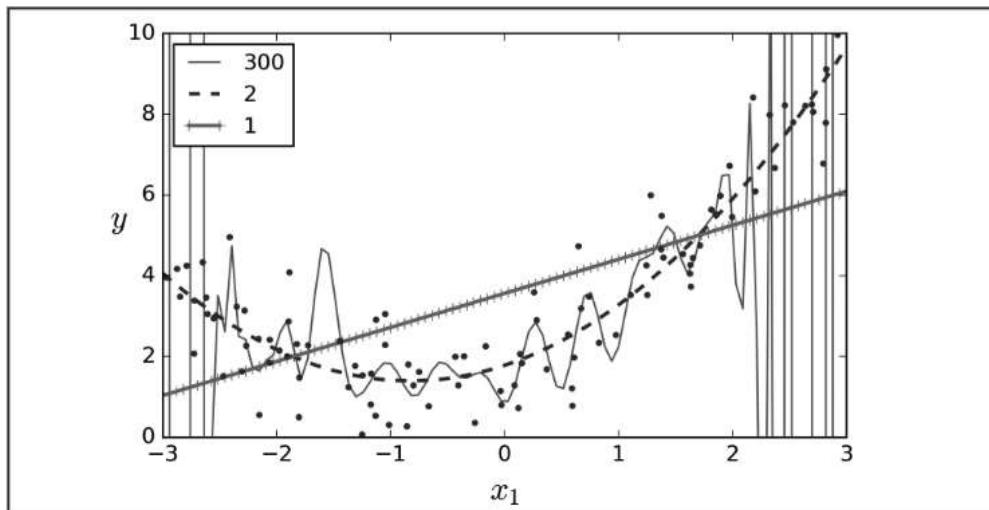


Figura 4-14. Regressão Polinomial de Grau Elevado

Naturalmente, esse modelo de Regressão Polinomial de Grau Elevado está se sobreajustando excessivamente aos dados de treinamento, enquanto o modelo linear está se subajustando a ele. O modelo que generalizará melhor neste caso é o modelo quadrático. Isso faz sentido porque os dados foram gerados com a utilização de um modelo quadrático, mas, em geral, você não saberá qual função gerou os dados, então como você pode decidir a complexidade de seu modelo? Como você pode dizer que seu modelo está se sobreajustando ou subajustando aos dados?

No Capítulo 2, você utilizou a validação cruzada para obter uma estimativa do desempenho da generalização de um modelo. Se um modelo funciona bem nos dados de treinamento, mas generaliza mal de acordo com as métricas da validação cruzada, seu modelo é sobreajustado. Se ele funciona mal em ambos, então é subajustado. Esta é uma forma de saber quando um modelo é muito simples ou muito complexo.

Outra maneira seria olhar para as *curvas de aprendizado*: são plotagens de desempenho do modelo no conjunto de treinamento e no conjunto de validação como uma função do tamanho do conjunto de treinamento (ou a iteração de treinamento). Para gerar as plotagens, basta treinar o modelo várias vezes em subconjuntos de tamanhos diferentes no conjunto de treinamento. O código a seguir define uma função que plota as curvas de aprendizado de alguns dados de treinamento em um dado modelo:

```

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")

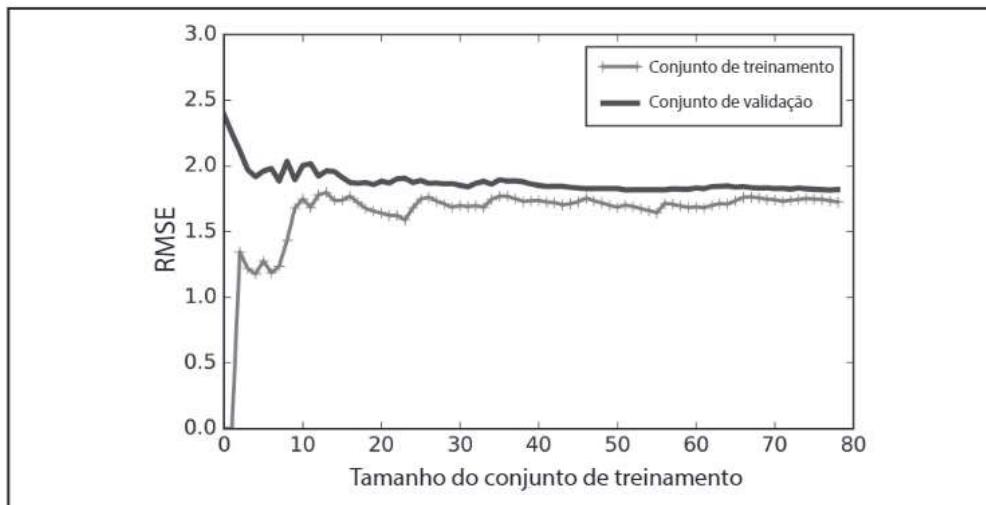
```

Vejamos as curvas de aprendizado do modelo de Regressão Linear Simples (uma linha reta, Figura 4-15):

```

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)

```



*Figura 4-15. Curvas de Aprendizado*

Isso merece uma explicação. Primeiro, vejamos o desempenho nos dados de treinamento: quando há apenas uma ou duas instâncias no conjunto de treinamento, o modelo se ajusta perfeitamente, e é por isso que a curva começa em zero. Mas, ao adicionarmos novas instâncias, torna-se impossível que o modelo se ajuste perfeitamente, seja porque são dados ruidosos ou porque não são dados lineares. Assim, esse erro subirá até atingir um platô, um momento em que adicionar novas instâncias ao conjunto de treinamento não tornará o erro médio muito melhor ou pior. Agora, vejamos a performance do modelo nos

dados de validação. Quando o modelo é treinado em poucas instâncias de treinamento, é incapaz de generalizar adequadamente e é por isso que o erro de validação é inicialmente bem grande. Então, à medida que apresentamos mais exemplos de treinamento, o modelo aprende, e, assim, o erro de validação diminui lentamente. No entanto, mais uma vez, uma linha reta não consegue fazer um bom trabalho na modelagem dos dados, então o erro acaba em um platô muito próximo da outra curva.

Essas curvas de aprendizado são típicas de um modelo subajustado. Ambas atingiram um platô; estão próximas e bastante altas.



Se o seu modelo estiver subajustando os dados de treinamento, não adiantará acrescentar mais exemplos de treinamento. Você precisa utilizar um modelo mais complexo ou obter melhores características.

Agora, vejamos as curvas de aprendizado de um modelo polinomial de 10º grau nos mesmos dados (Figura 4-16):

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
    ("lin_reg", LinearRegression()),
])

plot_learning_curves(polynomial_regression, X, y)
```

Essas curvas de aprendizado se parecem com as anteriores, mas há duas diferenças muito importantes:

- O erro nos dados de treinamento é muito inferior ao encontrado no modelo de Regressão Linear;
- Há uma lacuna entre as curvas. Isso significa que o modelo tem um desempenho significativamente melhor nos dados de treinamento do que nos dados de validação, marca registrada de um modelo sobreajustado. No entanto, se você utilizasse um conjunto de treinamento muito maior, as duas curvas continuariam se aproximando.

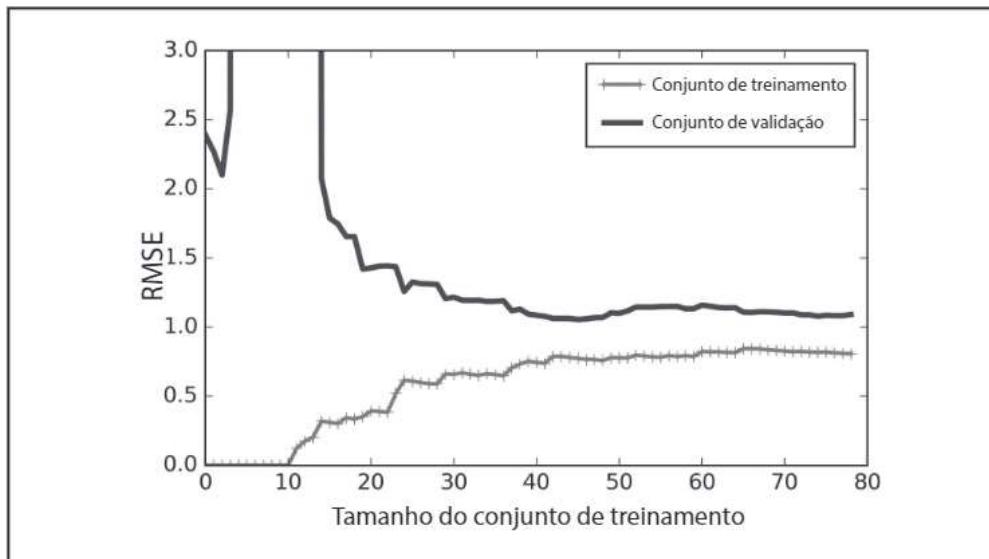


Figura 4-16. Curvas de Aprendizado para o modelo polinomial



Uma maneira de melhorar um modelo de sobreajuste é alimentá-lo com mais dados de treinamento até que o erro de validação atinja o erro de treinamento.

## A Compensação do Viés/Variância

Um importante resultado teórico das estatísticas e do Aprendizado de Máquina é o fato de que o erro de generalização de um modelo pode ser expresso como a soma de três erros muito diferentes:

### Viés

Esta parte do erro de generalização deve-se a hipóteses erradas, como assumir que os dados são lineares quando, na verdade, são quadráticos. Um modelo com *viés* elevado provavelmente se subajustará aos dados de treinamento.<sup>10</sup>

### Variância

Esta parte deve-se à sensibilidade excessiva do modelo a pequenas variações nos dados de treinamento. Um modelo com muitos graus de liberdade (como um modelo polinomial de alto grau) provavelmente terá uma alta variação e, portanto, se sobreajustará aos dados de treinamento.

<sup>10</sup> Essa noção de viés não deve ser confundida com o termo encontrado em modelos lineares.

### Erro Irreduzível

Esta parte deve-se ao ruído dos dados em si. A única maneira de reduzir essa parte do erro é limpar os dados (por exemplo, corrigir as fontes de dados, como sensores quebrados, ou detectar e remover outliers).

Aumentar a complexidade de um modelo geralmente aumentará sua variância e reduzirá seu viés. Por outro lado, reduzir a complexidade de um modelo aumenta seu viés e reduz sua variância. É por isso que se chama compensação.

## Modelos Lineares Regularizados

Como vimos nos Capítulos 1 e 2, uma boa maneira de reduzir o sobreajuste é regularizar o modelo (isto é, restringi-lo): quanto menor for o grau de liberdade, mais difícil será para sobreajustar os dados. Por exemplo, uma forma simples de regularizar um modelo polinomial é reduzir o número de graus polinomiais.

Para um modelo linear, a regularização é normalmente alcançada ao restringir os pesos do modelo. Veremos a *Regressão de Ridge*, *Regressão Lasso* e *Elastic Net*, que implementam três maneiras diferentes de restringir os pesos.

## Regressão de Ridge

A *Regressão de Ridge* (também chamada de *Regularização de Tikhonov*) é uma versão regularizada da Regressão Linear: um *termo de regularização* igual a  $\alpha \sum_{i=1}^n \theta_i^2$  é adicionado à função de custo. Isso força o algoritmo de aprendizado a não apenas ajustar os dados, mas também manter os pesos do modelo o mais reduzidos possível. Observe que o termo de regularização só deve ser adicionado à função de custo durante o treinamento. Uma vez treinado o modelo, você deve avaliá-lo utilizando a medida de desempenho não regularizada.



É bastante comum que a função de custo utilizada durante o treinamento seja diferente da medida de desempenho utilizada para o teste. Além da regularização, outra razão pela qual elas podem ser diferentes é que uma boa função de custo de treinamento deve ter derivadas que aceitem bem a otimização, enquanto a medida de desempenho utilizada para testes deve ser o mais próxima possível do objetivo final. Um bom exemplo disso é um classificador treinado com uma função de custo, como a *log loss* (será discutida em breve), mas avaliada com a utilização de precisão/revocação.

O hiperparâmetro  $\alpha$  controla o quanto você quer regularizar o modelo. Se  $\alpha = 0$ , então a Regressão de Ridge é apenas uma Regressão Linear. Se  $\alpha$  for muito grande, então todos

os pesos acabarão próximos de zero e o resultado será uma linha plana que passa pela média dos dados. A Equação 4-8 apresenta a função de custo da Regressão de Ridge.<sup>11</sup>

*Equação 4-8. Função de custo da Regressão de Ridge*

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Repare que o termo de polarização  $\theta_0$  não é regularizado (a soma começa em  $i = 1$ , não 0). Se definirmos  $w$  como o vetor de pesos das características ( $\theta_1$  a  $\theta_n$ ), então o termo de regularização é simplesmente igual a  $\frac{\alpha}{2}(\|w\|_2)^2$ , em que  $\|\cdot\|_2$  representa a norma  $\ell_2$  do vetor de peso.<sup>12</sup> Para o Gradiente Descendente, adicione  $\alpha w$  ao vetor gradiente MSE (Equação 4-6).



É importante dimensionar os dados (por exemplo, utilizando um `StandardScaler`) antes de executar a Regressão de Ridge, pois ela é sensível à escala dos recursos de entrada. Isso é verdade para a maioria dos modelos regularizados.

A Figura 4-17 mostra vários modelos Ridge treinados em alguns dados lineares com a utilização de diferentes valores de  $\alpha$ . À esquerda, modelos simples de Ridge são utilizados levando a previsões lineares. À direita, utilizando `PolyomialFeatures(degree=10)`, os dados são primeiramente expandidos, então escalonados com `StandardScaler`, e finalmente os modelos Ridge são aplicados às características resultantes: isto é uma Regressão Polinomial com regularização Ridge. Observe como o aumento de  $\alpha$  leva a previsões mais planas (ou seja, menos extremas, mais razoáveis); isso reduz a variância do modelo, mas aumenta seu *víés*.

Tal como acontece com a Regressão Linear, podemos executar a Regressão de Ridge por meio de uma equação em forma fechada ou pelo Gradiente Descendente. Os prós e contras são os mesmos. A Equação 4-9 mostra a solução de forma fechada ( $A$  é a matriz<sup>13</sup> de identidade  $n \times n$  com exceção de um 0 na célula superior esquerda, correspondente ao termo de polarização).

11 É comum usar a notação  $J(\theta)$  para funções de custo que não possuem um nome curto; muitas vezes, usaremos essa notação ao longo deste livro. O contexto deixará claro qual função de custo é discutida.

12 Normas são abordadas no Capítulo 2.

13 Uma matriz quadrada cheia de “0” exceto pelo “1” na diagonal principal (superior esquerda até a inferior direita).

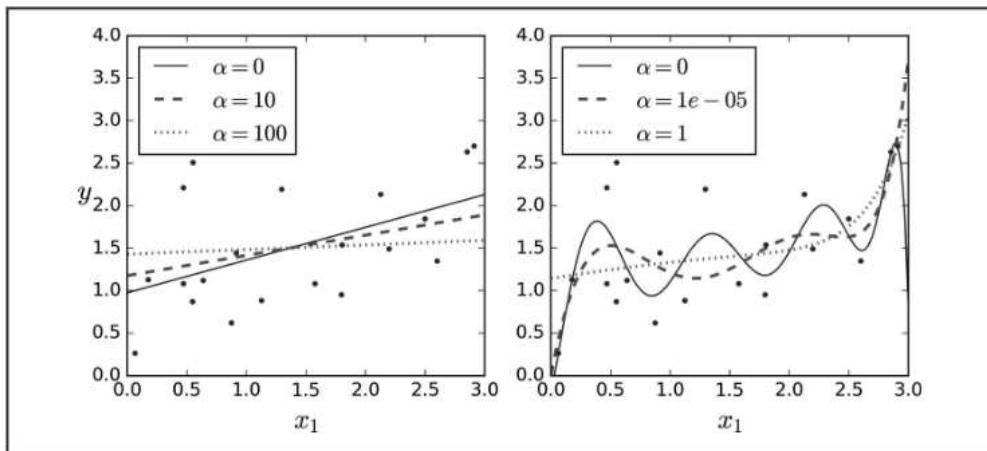


Figura 4-17. Regressão de Ridge

Equação 4-9. Solução em forma fechada de Regressão de Ridge

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Veja como executar a Regressão de Ridge com Scikit-Learn utilizando uma solução de forma fechada (uma variante da Equação 4-9 utilizando uma técnica de fatoração de matriz de André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([ 1.55071465])
```

E utilizando Gradiente Descendente Estocástico:<sup>14</sup>

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([ 1.13500145])
```

O hiperparâmetro `penalty` define o tipo de termo de regularização para uso. Especificar "`l2`" indica que você quer que o SGD adicione um termo de regularização à função de custo igual a metade do quadrado da norma  $\ell_2$  do vetor de peso: isto é simplesmente Regressão de Ridge.

<sup>14</sup> Como alternativa, você pode utilizar a classe `Ridge` com o solucionador "sag". O GD Médio estocástico é uma variante da SGD. Para mais detalhes, veja a apresentação "Minimizing Finite Sums with the Stochastic Average Gradient Algorithm" (<http://goo.gl/vxVya2>) por Mark Schmidt *et al.* da Universidade da Colúmbia Britânica.

## Regressão Lasso

*Least Absolute Shrinkage and Selection Operator Regression* (simplesmente chamada Regressão Lasso) é outra versão regularizada da Regressão Linear: como a Regressão de Ridge, ela adiciona um termo de regularização à função de custo, mas utiliza a norma  $\ell_1$  do vetor de peso em vez da metade do quadrado da norma  $\ell_2$  (veja a Equação 4-10).

Equação 4-10. Função de custo da Regressão Lasso

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

A Figura 4-18 mostra a mesma coisa que a Figura 4-17, mas substitui modelos Ridge por modelos Lasso e utiliza valores de  $\alpha$  menores.

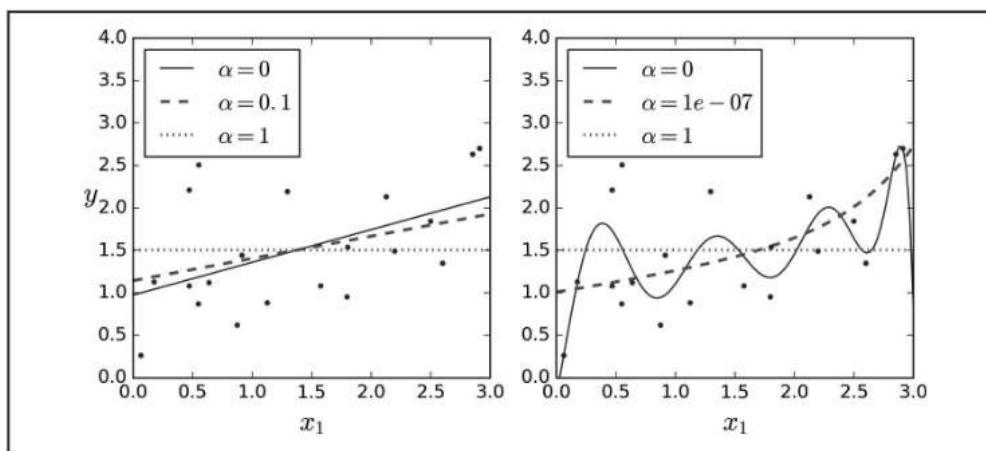


Figura 4-18. Regressão Lasso

Uma característica importante da Regressão Lasso é que ela tende a eliminar completamente os pesos das características menos importantes (ou seja, ajustá-las para zero). Por exemplo, a linha pontilhada na plotagem à direita na Figura 4-18 (com  $\alpha = 10^{-7}$ ) parece quadrática, quase linear: todos os pesos para as características polinomiais de alto grau são iguais a zero. Em outras palavras, a Regressão Lasso executa automaticamente a seleção de características e exibe um modelo esparsa (ou seja, com poucos pesos de características diferentes de zero).

Você pode ter uma ideia do porquê é esse o caso olhando a Figura 4-19: na plotagem à esquerda, os contornos de fundo (elipses) representam uma função de custo MSE não regulada ( $\alpha = 0$ ), e o círculo branco mostra o caminho do Gradiente Descendente em Lote com aquela função de custo. Os contornos do primeiro plano (diamantes)

representam a penalidade  $\ell_1$  e os triângulos mostram o caminho BGD apenas para esta penalidade ( $\alpha \rightarrow \infty$ ). Observe como o caminho primeiro atinge  $\theta_1 = 0$ , então rola por uma calha até atingir  $\theta_2 = 0$ . Na plotagem superior à direita, os contornos representam a mesma função de custo mais uma penalidade de  $\ell_1$  com  $\alpha = 0,5$ . O mínimo global está no eixo  $\theta_2 = 0$ . O BGD primeiro alcança  $\theta_2 = 0$ , depois rola a calha até atingir o mínimo global. As duas plotagens de baixo mostram o mesmo, mas utilizam uma penalidade  $\ell_2$  em vez disso. O mínimo regularizado está mais próximo de  $\theta = 0$  do que o mínimo não regulamentado, mas os pesos não são totalmente eliminados.

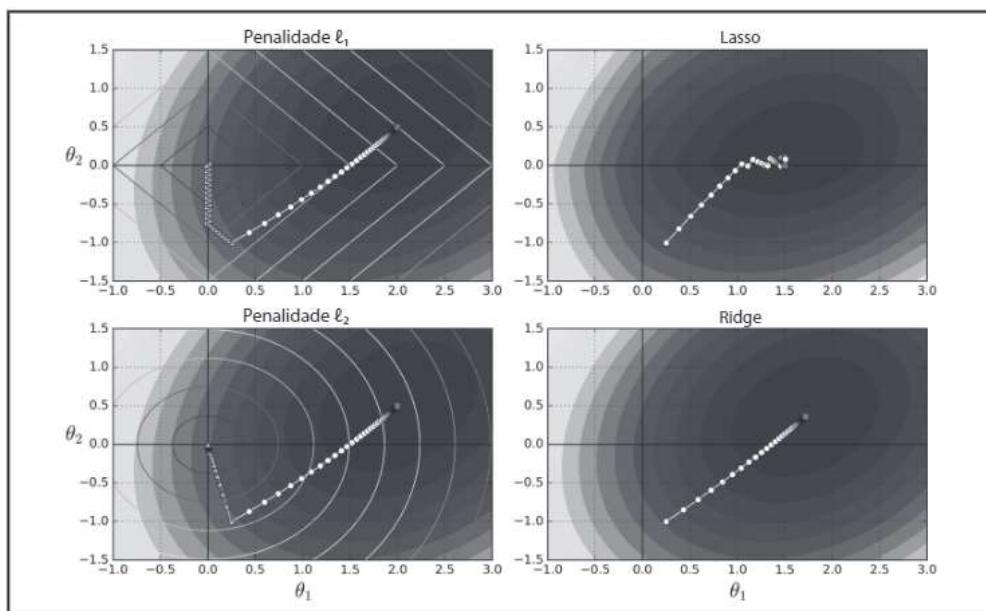


Figura 4-19. Regularização Lasso versus Ridge



Na função de custo Lasso, o caminho do BGD tende a saltar pela calha para o final. Isso ocorre porque a inclinação muda abruptamente para  $\theta_2 = 0$ . Você precisa reduzir gradualmente a taxa de aprendizado para realmente convergir para o mínimo global.

A função de custo Lasso não é diferenciável em  $\theta_i = 0$  (para  $i = 1, 2, \dots, n$ ), mas o Gradiente Descendente ainda funciona bem se você utilizar um *vetor subgradiente*  $\mathbf{g}$ <sup>15</sup> quando qualquer  $\theta_i$  for igual a 0. A Equação 4-11 mostra uma equação de vetor subgradiente que você pode utilizar para o Gradiente Descendente com a função de custo Lasso.

<sup>15</sup> Você pode encarar um vetor subgradiente em um ponto não diferenciável como um vetor intermediário entre os vetores de gradiente em torno desse ponto.

*Equação 4-11. Vetor subgradiente da Regressão Lasso*

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sinal}(\theta_1) \\ \text{sinal}(\theta_2) \\ \vdots \\ \text{sinal}(\theta_n) \end{pmatrix} \quad \text{onde} \quad \text{sinal}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Veja um pequeno exemplo do Scikit-Learn utilizando a classe **Lasso**. Observe que você poderia utilizar o `SGDRegressor(penalty="l1")` em vez disso.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

## Elastic Net

Elastic Net é um meio termo entre a Regressão de Ridge e a Regressão Lasso. O termo de regularização é uma simples mistura dos termos de regularização Ridge e Lasso, e você pode controlar a taxa de mistura  $r$ . Quando  $r = 0$ , a Elastic Net é equivalente à Regressão de Ridge, e, quando  $r = 1$ , ela é equivalente à Regressão Lasso (veja a Equação 4-12).

*Equação 4-12. Função de custo Elastic Net*

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

Então, quando utilizar uma simples Regressão Linear (por exemplo, sem qualquer regularização), Ridge, Lasso ou Elastic Net? Quase sempre é preferível ter pelo menos um pouco de regularização, então geralmente você deve evitar uma simples Regressão Linear. Ridge é um bom padrão, mas, se você suspeitar que apenas algumas características são úteis, deve preferir Lasso ou Elastic Net, pois elas tendem a reduzir a zero os pesos das características inúteis, conforme discutido. No geral, a Elastic Net é preferida à Lasso porque esta pode se comportar erraticamente quando o número de características for maior que o número de instâncias de treinamento ou quando várias características estiverem fortemente correlacionadas.

Segue um breve exemplo utilizando o **ElasticNet** do Scikit-Learn (`l1_ratio` corresponde à taxa de mixagem  $r$ ):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

## Parada Antecipada

Uma maneira bem diferente de regularizar algoritmos de aprendizado iterativos como os Gradientes Descendentes é interromper o treinamento assim que o erro de validação atingir um mínimo. Isto é chamado de *parada antecipada*. A Figura 4-20 mostra um modelo complexo (neste caso, um modelo de Regressão Polinomial de alto grau) sendo treinado com o uso do Grandiente Descendente em Lote. Conforme as épocas ocorrem, o algoritmo aprende e seu erro de previsão (RMSE) no conjunto de treinamento naturalmente decai, assim como seu erro de previsão no conjunto de validação. Entretanto, após um tempo, o erro de validação para de decair e começa a subir, indicando que o modelo começou a se sobreajustar aos dados de treinamento. Com a parada antecipada você para o treinamento assim que o erro de validação atingir o mínimo. É uma técnica de regularização tão simples e eficiente que Geoffrey Hinton a chamou de “*lindo almoço grátis*”.

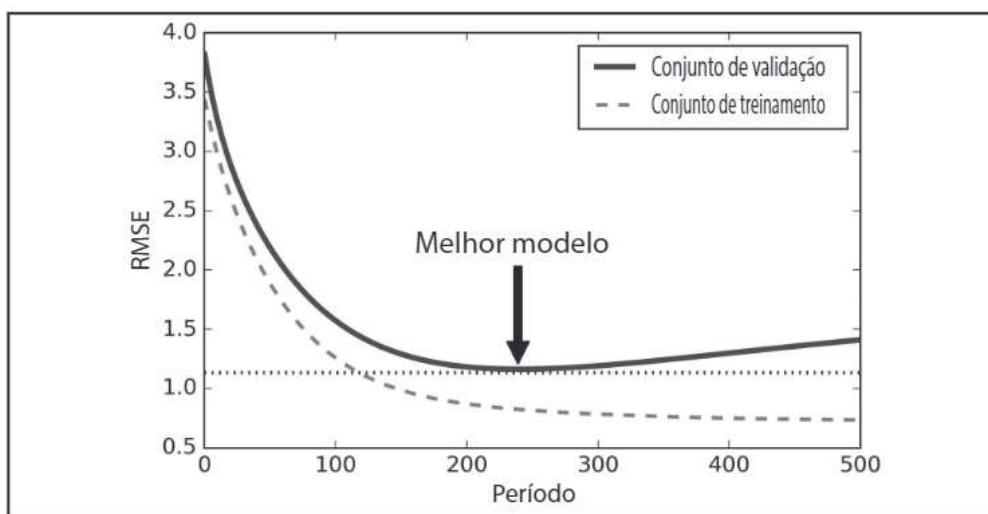


Figura 4-20. Regularização da Parada Antecipada



Com o Gradiente Descendente Estocástico e o Minilote, as curvas não são tão suaves, e pode ser difícil saber se você atingiu o mínimo ou não. Uma solução seria parar somente após o erro de validação ficar acima do mínimo por um período (quando você tiver certeza que o modelo não ficará melhor), então retroceda seus parâmetros ao ponto onde o erro de validação estava no mínimo.

Esta é uma implementação básica de parada antecipada:

```

from sklearn.base import clone
# prepare os dados
poly_scaler = Pipeline([
    ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler()) ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):

    sgd_reg.fit(X_train_poly_scaled, y_train) # continua de onde parou
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)

```

Note que, com `warm_start=True`, quando chama o método `fit()`, ele apenas continua o treinamento onde foi deixado em vez de recomeçar do início.

## Regressão Logística

Como discutimos no Capítulo 1, alguns algoritmos de regressão também podem ser utilizados para classificação (e vice-versa). A *Regressão Logística* (também chamada de *Regressão Logit*) é comumente utilizada para estimar a probabilidade de uma instância pertencer a uma determinada classe (por exemplo, qual é a probabilidade desse e-mail ser spam?). Se a probabilidade estimada for maior que 50%, então o modelo prevê que a instância pertence a essa classe (chamada de classe positiva, rotulada como “1”), ou então ela prevê que não (isto é, pertence à classe negativa, rotulada “0”). Isso o transforma em um classificador binário.

### Estimando Probabilidades

Então, como funciona? Assim como um modelo de Regressão Linear, um modelo de Regressão Logística calcula uma soma ponderada das características de entrada (mais um termo de polarização), mas, em vez de gerar o resultado diretamente como o modelo de Regressão Linear, gera a *logística* desse resultado (veja a Equação 4-13).

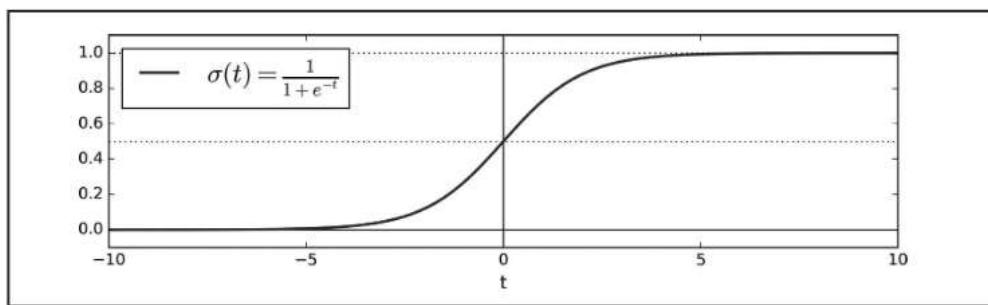
*Equação 4-13. Modelo de regressão logística probabilidade estimada (forma vetorizada)*

$$\hat{p} = h_{\theta}(x) = \sigma(\theta^T \cdot x)$$

A logística — também chamada de *logit*, subscrita  $\sigma()$  — é uma função *sigmóide* (ou seja, formato-S) que mostra um número entre 0 e 1. É definida como mostrado na Equação 4-14 e Figura 4-21.

*Equação 4-14. Função Logística*

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



*Figura 4-21. Função Logística*

Uma vez que o modelo de Regressão Logística estimou a probabilidade  $\hat{p} = h_{\theta}(x)$  que a instância  $x$  pertence à classe positiva, ela pode fazer facilmente sua previsão  $\hat{y}$  (veja a Equação 4-15).

*Equação 4-15. Previsão do modelo de regressão logística*

$$\hat{y} = \begin{cases} 0 & \text{se } \hat{p} < 0.5, \\ 1 & \text{se } \hat{p} \geq 0.5. \end{cases}$$

Observe que  $\sigma(t) < 0.5$  quando  $t < 0$ , e  $\sigma(t) \geq 0.5$  quando  $t \geq 0$ , então um modelo de Regressão Logística prevê 1 se  $\theta^T \cdot x$  for positivo, e 0 se for negativo.

## Treinamento e Função de Custo

Agora, você já sabe como um modelo de Regressão Logística estima as probabilidades e faz previsões. Mas como ele é treinado? O objetivo do treinamento é definir o vetor do parâmetro  $\theta$  para que o modelo estime altas probabilidades para instâncias positivas ( $y = 1$ ) e baixas probabilidades para instâncias negativas ( $y = 0$ ). A função de custo incorpora essa ideia mostrada na Equação 4-16 para uma instância de treinamento única  $x$ .

*Equação 4-16. Função de custo de uma instância de treinamento única*

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{se } y = 1, \\ -\log(1 - \hat{p}) & \text{se } y = 0. \end{cases}$$

Esta função de custo faz sentido porque  $-\log(t)$  cresce muito quando  $t$  se aproxima de 0, então o custo será maior se o modelo estimar uma probabilidade próxima a 0 para uma instância positiva, e também será muito maior se o modelo estimar uma probabilidade próxima a 1 para uma instância negativa. Por outro lado,  $-\log(t)$  é próximo a 0 quando  $t$  for próximo de 1, então o custo será próximo de 0 se a probabilidade estimada for próxima de 0 para uma instância negativa ou próxima de 1 para uma instância positiva, que é exatamente o que queremos.

A função de custo em relação a todo o conjunto de treinamento é simplesmente o custo médio em relação a todas as instâncias de treinamento. Ela pode ser escrita em uma simples expressão (como você pode verificar facilmente), chamada *log loss*, mostrada na Equação 4-17.

*Equação 4-17. Função de custo de Regressão Logística (log loss)*

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

A má notícia é que não há equação de forma fechada conhecida para calcular o valor de  $\theta$  que minimize esta função de custo (não há uma equivalente do Método dos Mínimos Quadrados). Mas a boa notícia é que esta função de custo é convexa, então o Gradiente Descendente (ou qualquer outro algoritmo de otimização) se certifica de encontrar o mínimo global (se a taxa de aprendizado não for muito grande e se você esperar o suficiente). As derivadas parciais da função de custo com relação ao  $j$ -ésimo modelo do parâmetro  $\theta_j$  é dada pela Equação 4-18.

*Equação 4-18. Derivada parcial da função de custo logística*

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

Esta equação se parece muito com a Equação 4-5: ela calcula o erro de previsão para cada instância, o multiplica pelo valor da  $j$ -ésima característica, e então calcula a média em relação a todas as instâncias de treinamento. Uma vez que tenha o vetor de gradiente contendo todas as derivadas parciais, você pode utilizá-las no algoritmo do Gradiente Descendente. É isso: agora você já sabe como treinar um modelo de Regressão Logística. Para o GD Estocástico, você poderia pegar uma instância por vez, e para o GD Minilote você poderia utilizar um minilote por vez.

## Fronteiras de Decisão

Utilizaremos o conjunto de dados da íris para ilustrar a Regressão Logística. Este é um conjunto de dados famoso que contém o comprimento e a largura das sépalas e pétalas de 150 flores de íris de três espécies diferentes: Iris-Setosa, Iris-Versicolor e Iris-Virginica (veja a Figura 4-22).

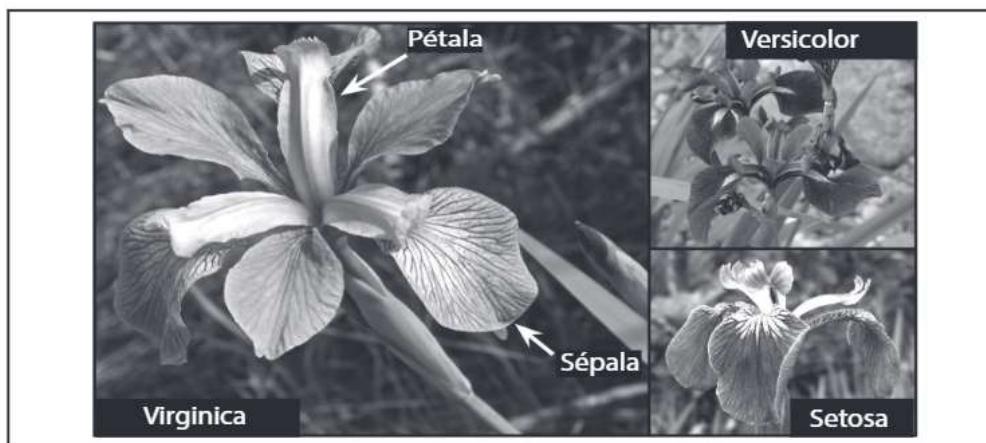


Figura 4-22. Flores de três espécies de plantas de íris<sup>16</sup>

Tentaremos construir um classificador para detectar o tipo Iris-Virginica baseado sómente na característica do comprimento da pétala. Primeiro carregaremos os dados:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:] # largura da pétala
>>> y = (iris["target"] == 2).astype(np.int) # 1 se for Iris Virginica, caso contrário, 0
```

Agora, treinaremos um modelo de Regressão Logística:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Vejamos as probabilidades estimadas do modelo para flores com larguras de pétalas variando de 0 a 3 cm (Figura 4-23):

---

<sup>16</sup> Fotos reproduzidas a partir das páginas correspondentes da Wikipédia. Foto da Iris-Virginica de Frank Mayfield (Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>)), foto da Iris-Versicolor de D. Gordon E. Robertson (Creative Commons BY- SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>))), e a foto Iris-Setosa é de domínio público.

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris-Virginica")
plt.plot(X_new, y_proba[:, 0], "b--", label="Not Iris-Virginica")
# mais código Matplotlib para deixar a imagem bonita
```

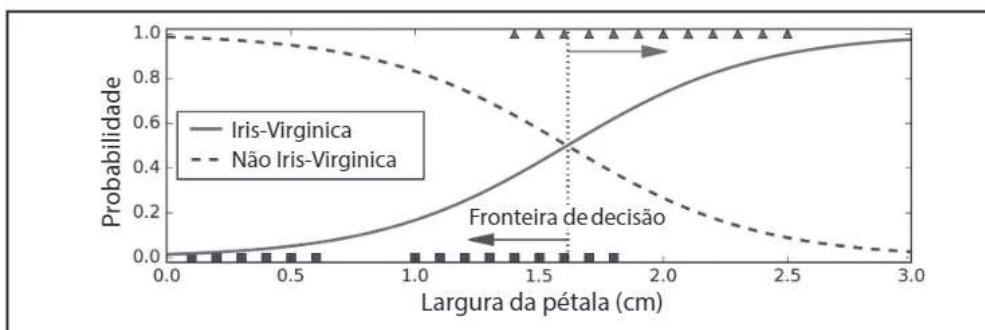


Figura 4-23. Probabilidades estimadas e limites da decisão

A largura da pétala das flores Iris-Virginica (representada por triângulos) varia de 1,4cm a 2,5cm, enquanto as outras íris (representadas por quadrados) geralmente têm uma largura menor de pétala, variando de 0,1cm a 1,8cm. Observe que há um pouco de sobreposição. Quando está acima de cerca de 2cm, o classificador confia muito que seja uma flor Iris-Virginica (ele mostra uma alta probabilidade para essa classe), e se estiver abaixo de 1cm ele é muito confiante de que não é uma Iris-Virginica (alta probabilidade para a classe “Não Iris-Virginica”). Entre estes extremos, o classificador não tem tanta certeza. No entanto, se você pedir que ele preveja a classe (utilizando o método `predict()` em vez do método `predict_proba()`), ele retornará a classe que for mais provável. Portanto, existe uma *fronteira de decisão*, em torno de 1,6cm, na qual ambas as probabilidades são iguais a 50%: se a largura da pétala for superior a 1,6 cm, o classificador preverá que a flor é uma Iris-Virginica, ou então que não é (mesmo que não esteja muito confiante):

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

A Figura 4-24 mostra o mesmo conjunto de dados, mas desta vez exibindo duas características: largura e comprimento da pétala. Uma vez treinado, o classificador de Regressão Logística pode estimar a probabilidade de uma nova flor ser uma Iris-Virginica com base nessas duas características. A linha tracejada representa os pontos onde o modelo estima uma probabilidade de 50%: esta é a fronteira de decisão do modelo. Observe que é uma fronteira linear.<sup>17</sup> Cada linha paralela representa os pontos onde o modelo produz uma probabilidade específica, de 15% (inferior esquerda) a 90% (superior direita). Todas as

<sup>17</sup> É o conjunto de pontos  $x$  tal que  $\theta_0 + \theta_1x_1 + \theta_2x_2 = 0$  define uma linha reta.

flores além da linha superior direita têm mais de 90% de chance de serem Iris-Virginica, de acordo com o modelo.

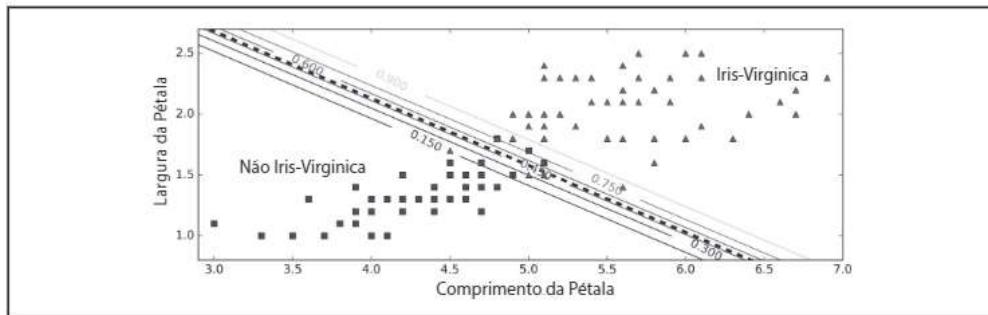


Figura 4-24. Fronteiras de decisão linear

Assim como outros modelos lineares, os modelos de Regressão Logística podem ser regularizados usando as penalidades  $\ell_1$  ou  $\ell_2$ . O Scikit-Learn adiciona uma penalidade  $\ell_2$  por padrão.



O hiperparâmetro que controla a força de regularização de um modelo `LogisticRegression` do Scikit-Learn não é `alfa` (como em outros modelos lineares), mas seu inverso: `C`. Quanto maior o valor de `C`, menos o modelo é regularizado.

## Regressão Softmax

O modelo de Regressão Logística pode ser generalizado para suportar múltiplas classes diretamente sem a necessidade de treinar e combinar vários classificadores binários (como discutido no Capítulo 3). Isso é chamado *Regressão Softmax*, ou *Regressão Logística Multinomial*.

A ideia é bem simples: quando dada uma instância  $x$ , o modelo de Regressão Softmax primeiro calcula uma pontuação  $s_k(x)$  para cada classe  $k$ , então estima a probabilidade de cada classe aplicando a função *softmax* (também chamada *exponencial normalizada*) às pontuações. A equação para calcular  $s_k(x)$  deve ser familiar, pois é exatamente como a equação de previsão da Regressão Linear (veja a Equação 4-19).

Equação 4-19. Pontuação Softmax para a classe  $k$

$$s_k(x) = (\theta^{(k)})^T \cdot x$$

Observe que cada classe tem seu próprio vetor de parâmetro dedicado  $\theta^{(k)}$ . Todos esses vetores são armazenados tipicamente como linhas na *matriz de parâmetro*  $\Theta$ .

Uma vez calculada a pontuação de cada classe para a instância  $x$ , você pode estimar a probabilidade  $\hat{p}_k$  de a instância pertencer à classe  $k$  ao executar as pontuações através da função softmax (Equação 4-20): ela calcula a exponencial de cada pontuação e a normaliza (dividindo pela soma de todas as exponenciais).

*Equação 4-20. Função Softmax*

$$\hat{p}_k = \sigma(s(x))_k = \frac{\exp(s_k(x))}{\sum_{j=1}^K \exp(s_j(x))}$$

- $K$  é o número de classes;
- $s(x)$  é um vetor que contém as pontuações de cada classe para a instância  $x$ ;
- $\sigma(s(x))_k$  é a probabilidade estimada de que a instância  $x$  pertença à classe  $k$  dadas as pontuações de cada classe para aquela instância.

Assim como o classificador de Regressão Logística, o classificador de Regressão Softmax prevê a classe com a maior probabilidade estimada (que é simplesmente a classe com a maior pontuação), como mostrado na Equação 4-21.

*Equação 4-21. Previsão do classificador de Regressão Softmax*

$$\hat{y} = \operatorname{argmax}_k \sigma(s(x))_k = \operatorname{argmax}_k s_k(x) = \operatorname{argmax}_k \left( (\theta^{(k)})^T \cdot x \right)$$

- O operador *argmax* retorna o valor de uma variável que maximiza uma função. Nesta equação ele retorna o valor de  $k$ , que maximiza a probabilidade estimada  $\sigma(s(x))_k$ .



O classificador da Regressão Softmax prevê apenas uma classe de cada vez (ou seja, ele é multiclasse, não multioutput), portanto, deve ser usado apenas com classes mutuamente exclusivas, como diferentes tipos de plantas. Você não pode usá-lo para reconhecer várias pessoas em uma única imagem.

Agora que você sabe como o modelo estima as probabilidades e faz previsões, vamos dar uma olhada no treinamento. O objetivo é ter um modelo que estime uma alta probabilidade para a classe-alvo (e consequentemente uma baixa probabilidade para as outras). Minimizar a função de custo mostrada na Equação 4-22, denominada *entropia cruzada*, deve levar a esse objetivo porque penaliza o modelo quando ele calcula uma baixa probabilidade para uma classe-alvo. A entropia cruzada é frequentemente utilizada para medir a qualidade da combinação de um conjunto de probabilidades de classe estimadas com as classes-alvo (vamos utilizá-la muitas vezes nos capítulos seguintes).

*Equação 4-22. Função de custo de entropia cruzada*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log (\hat{p}_k^{(i)})$$

- $y_k^{(i)}$  é igual a 1 se a classe-alvo para a instância  $i$ -ésima for  $k$ ; caso contrário, será igual a 0.

Observe que, quando há apenas duas classes ( $K = 2$ ), essa função de custo é equivalente à função de custo da Regressão Logística (*log loss*; veja a Equação 4-17).

## Entropia Cruzada

A entropia cruzada originou-se da teoria da informação. Suponha que você deseja transmitir informações sobre o clima de forma eficiente todos os dias. Se houver oito opções (ensolarado, chuvoso, etc.), você poderia programar cada opção utilizando 3 bits desde que  $2^3 = 8$ . No entanto, se você acha que será ensolarado quase todos os dias, seria muito mais eficiente programar “ensolarado” com apenas um bit (0) e as outras sete opções com 4 bits (começando com 1). A entropia cruzada mede o número médio de bits por opção que você realmente envia. Se a sua suposição for perfeita, a entropia cruzada será apenas igual à entropia do próprio clima (ou seja, sua imprevisibilidade intrínseca). Mas, se seus pressupostos estiverem errados (por exemplo, se chover com frequência), a entropia cruzada será melhor por uma quantidade chamada *divergência Kullback–Leibler*.

A entropia cruzada entre duas distribuições de probabilidade  $p$  e  $q$  é definida como  $H(p, q) = -\sum_x p(x) \log q(x)$  (pelo menos quando as distribuições são discretas).

O vetor gradiente desta função de custo com relação à  $\theta^{(k)}$  é dado pela Equação 4-23:

*Equação 4-23. Vetor gradiente de entropia cruzada para classe k*

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Agora, você pode calcular o vetor de gradiente para cada classe e, em seguida, utilizar o Gradiente Descendente (ou qualquer outro algoritmo de otimização) para encontrar a matriz de parâmetro  $\Theta$  que minimiza a função de custo.

Utilizaremos a Regressão Softmax para classificar as íris nas três classes. O `LogisticRegression` do Scikit-Learn utiliza um contra todos automaticamente quando você o treina em mais de duas classes, mas você também pode configurar o hiperparâmetro `multi_class` para “`multinomial`” a fim de alternar para a Regressão Softmax. Você também deve especificar um solucionador que suporte a Regressão Softmax, como o solucionador “`lbfgs`” (veja a documentação do Scikit-Learn para mais detalhes). Ele também aplica a regularização  $\ell_2$  por padrão, que você pode controlar utilizando o hiperparâmetro `C`.

```
X = iris["data"][:, (2, 3)] # comprimento da pétala, largura da pétala
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Então, na próxima vez que você encontrar uma íris com pétalas de 5cm de comprimento e 2cm de largura, você pode pedir ao seu modelo para lhe dizer que tipo de íris é, e ele responderá Iris-Virginica (classe 2) com 94,2% de probabilidade (ou Iris-Versicolor com 5,8% de probabilidade):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07,  5.75276067e-02,  9.42471760e-01]])
```

A Figura 4-25 mostra os limites de decisão resultantes, representados pelas cores de fundo. Observe que as fronteiras de decisão entre duas classes são lineares. A figura também mostra as probabilidades para a classe Iris-Versicolor, representada pelas linhas curvas (por exemplo, a linha marcada com 0,450 representa o limite de probabilidade de 45%). Observe que o modelo pode prever uma classe que tenha uma probabilidade estimada abaixo de 50%. Por exemplo, no ponto em que todos os limites da decisão se encontram, todas as classes têm uma mesma probabilidade estimada de 33%.

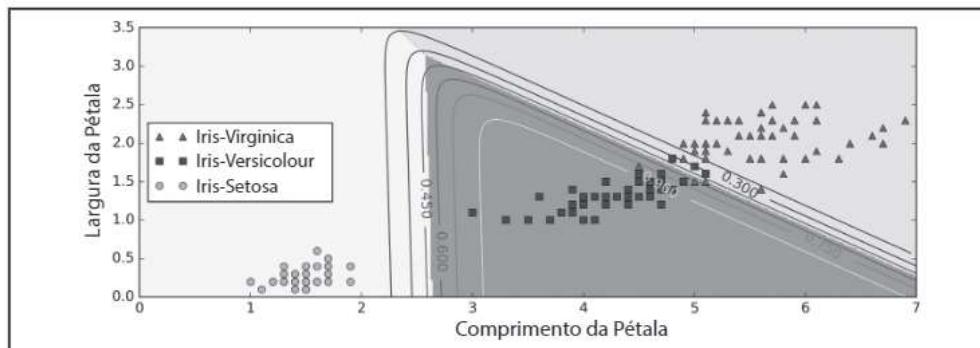


Figura 4-25. As fronteiras de decisão da Regressão Softmax

## Exercícios

1. Qual algoritmo de treinamento de Regressão Linear podemos utilizar se tivermos um conjunto de treinamento com milhões de características?
2. Suponha que as características do seu conjunto de treinamento tenham escalas muito diferentes. Que algoritmos podem sofrer com isso, e como? O que você pode fazer a respeito?

3. O Gradiente Descendente pode ficar preso em um mínimo local ao treinar um modelo de Regressão Logística?
  4. Se todos os algoritmos do Gradiente Descendente forem executados com tempo suficiente, eles o levarão ao mesmo modelo?
  5. Suponha que você utilize o Gradiente Descendente em Lote e plote seu erro de validação em cada época. Se você notar que o erro de validação sempre aumenta, o que provavelmente está acontecendo? Como consertar isso?
  6. É uma boa ideia parar o Gradiente Descendente em Minilote imediatamente quando o erro de validação aumentar?
  7. Qual algoritmo Gradiente Descendente (entre aqueles que discutimos) se aproximará mais rapidamente da solução ideal? Qual realmente convergirá? Como você pode fazer os outros convergirem também?
  8. Suponha que esteja utilizando a Regressão Polinomial. Você plota as curvas de aprendizado e percebe que existe um grande hiato entre o erro de treinamento e o de validação. O que está acontecendo? Quais são as três maneiras de resolver isso?
  9. Suponha que você esteja utilizando a Regressão de Ridge e perceba que o erro de treinamento e o de validação são quase iguais e bastante altos. Você diria que o modelo sofre de um viés elevado ou de alta variância? Devemos aumentar o hiperparâmetro  $\alpha$  ou reduzi-lo?
10. Por que você utilizaria:
- Regressão de Ridge em vez de Regressão Linear simples (ou seja, sem qualquer regularização)?
  - Lasso em vez de Regressão de Ridge?
  - Elastic Net em vez de Lasso?
11. Suponha que você deseja classificar fotos como exteriores/inteiros e dia/noite. Você deve implementar dois classificadores de Regressão Logística ou um de Regressão Softmax?
12. Implemente o Gradiente Descendente em Lote com parada antecipada para a Regressão Softmax (sem utilizar o Scikit-Learn).

Soluções dos exercícios disponíveis no Apêndice A.

## Capítulo 5

# Máquinas de Vetores de Suporte

Uma Máquina de Vetores de Suporte (SVM) é um modelo muito poderoso e versátil de Aprendizado de Máquina capaz de realizar classificações lineares ou não lineares, de regressão e até mesmo detecção de outliers. É um dos modelos mais populares no Aprendizado de Máquina e qualquer pessoa interessada no tema deve tê-lo em sua caixa de ferramentas. As SVM são particularmente adequadas para a classificação de conjuntos de dados complexos, porém de pequeno ou médio porte.

Este capítulo explicará seus conceitos básicos, como utilizá-las e como funcionam.

## Classificação Linear das SVM

A ideia fundamental por trás das SVM é melhor explicada com algumas imagens. A Figura 5-1 mostra parte do conjunto de dados da íris que foi introduzido no final do Capítulo 4. As duas classes podem ser separadas facilmente com uma linha reta (elas são *separadas linearmente*). A plotagem à esquerda mostra os limites de decisão de três possíveis classificadores lineares. O modelo cujo limite de decisão é representado pela linha tracejada é tão ruim que nem sequer consegue separar as classes corretamente. Os outros dois modelos funcionam perfeitamente neste conjunto, mas seus limites de decisão chegam tão perto das instâncias que provavelmente não funcionarão tão bem em novas instâncias. Em contraste, a linha contínua na plotagem à direita representa o limite de decisão de um classificador SVM; esta linha não somente separa as duas classes, mas também fica o mais longe possível das instâncias de treinamento mais próximas. Você pode pensar em um classificador SVM como o preenchimento da via mais larga possível entre as classes (representada pelas linhas tracejadas paralelas). Isso é chamado de *classificação de margens largas*.

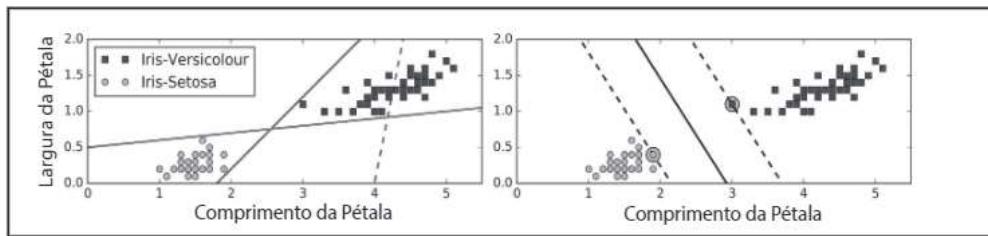


Figura 5-1. Classificação de margens largas

Observe que adicionar mais instâncias de treinamento “fora da via” não afeta o limite de decisão: ele está totalmente determinado (ou “suportado”) pelas instâncias localizadas na borda da via. Essas instâncias são chamadas de *vetores de suporte* (elas estão circuladas na Figura 5-1).



As SVM são sensíveis às escalas das características, como você pode ver na Figura 5-2: na plotagem à esquerda, a escala vertical é muito maior do que a horizontal, então a via mais larga possível está próxima da horizontal. Após o escalonamento das características (por exemplo, ao utilizarmos o `StandardScaler` do Scikit-Learn), o limite de decisão parece bem melhor (na plotagem à direita).

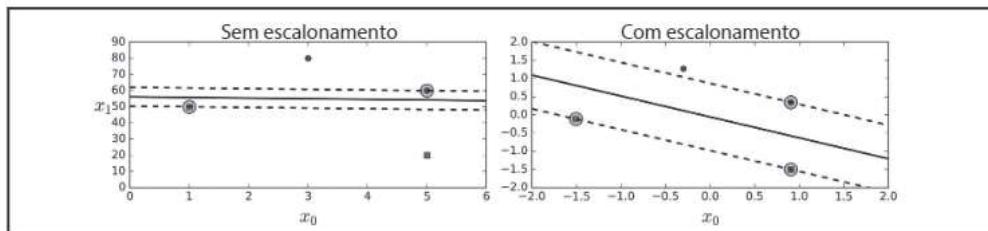


Figura 5-2. Sensitividade ao escalonamento das características

## Classificação de Margem Suave

Se impusermos estritamente que todas as instâncias estejam fora da via e do lado direito, estaremos aplicando uma *classificação de margem rígida*, que possui duas questões principais. Primeiro, ela só funciona se os dados forem linearmente separáveis e, segundo, ela é bastante sensível a outliers. A Figura 5-3 mostra o conjunto de dados da íris com apenas um outlier adicional: à esquerda, é impossível encontrar uma margem rígida e, à direita, a fronteira de decisão termina bem diferente do que vimos na Figura 5-1 sem o outlier, e provavelmente não generalizará tão bem.

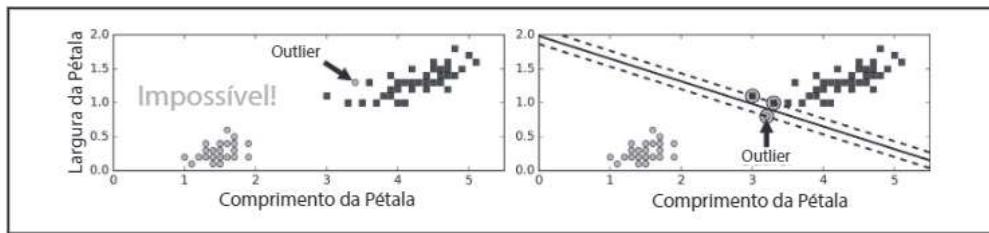


Figura 5-3. Sensibilidade da margem rígida aos outliers

É preferível utilizar um modelo mais flexível a fim de evitar esses problemas. O objetivo é encontrar um bom equilíbrio entre manter a via o mais larga possível e limitar as *violações de margem* (ou seja, as instâncias que acabam no meio da via ou mesmo do lado errado). Isto é chamado *classificação de margem suave*.

Nas classes SVM do Scikit-Learn você controla esse equilíbrio ao utilizar o hiperparâmetro  $C$ : um valor menor de  $C$  leva a uma via mais larga, mas com mais violações das margens. A Figura 5-4 mostra os limites de decisão e as margens de dois classificadores SVM de margem suave em um conjunto de dados não linearmente separáveis. À esquerda, ao utilizar um alto valor de  $C$ , o classificador faz menos violações na margem, mas fica com uma margem menor. À direita, ao utilizar um baixo valor de  $C$ , a margem fica muito maior, mas muitas instâncias ficam na via. No entanto, parece provável que o segundo classificador generalizará melhor: na verdade, mesmo neste conjunto de treinamento ele comete menos erros de previsão já que a maioria das violações da margem está do lado correto do limite de decisão.

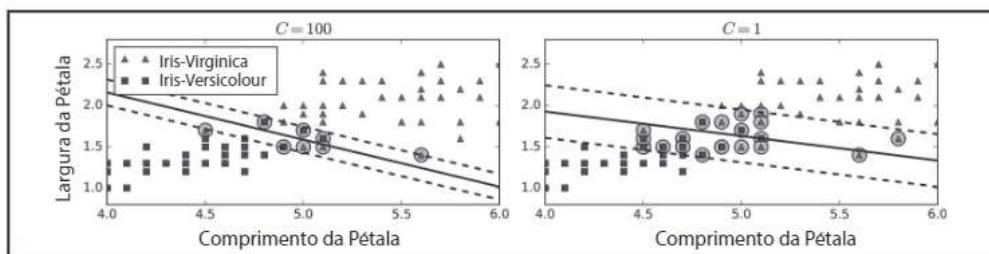


Figura 5-4. Menos violações de margem versus grandes margens



Se seu modelo SVM está sobreajustado, você pode tentar regularizá-lo reduzindo o valor de  $C$ .

O código do Scikit-Learn a seguir carrega o conjunto de dados da íris, escalona as características e treina um modelo SVM linear (utilizando a classe `LinearSVC` com  $C = 1$  e a

função *hinge loss* descrita brevemente) para detectar as flores Iris-Virginica. O modelo resultante está representado à direita da Figura 5-4.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # comprimento da pétala, largura da pétala
y = (iris["target"] == 2).astype(np.float64) # Iris Virginica

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])
svm_clf.fit(X, y)
```

Então, como de costume, você pode utilizar o modelo para fazer previsões:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```



Ao contrário dos classificadores de Regressão Logística, os classificadores SVM não apresentam probabilidades para cada classe.

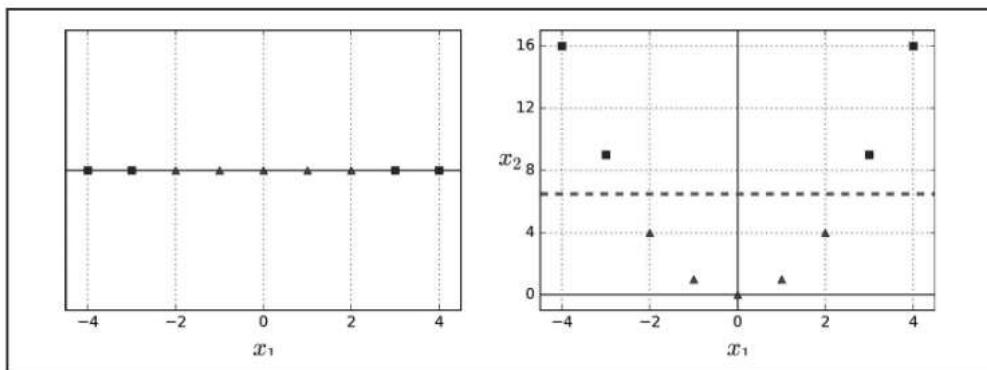
Como alternativa, você poderia utilizar a classe `SVC` ao aplicar `SVC(kernel="linear", C=1)`, mas ela é muito mais lenta, principalmente em grandes conjuntos de treinamento, portanto não é recomendável. Outra opção seria utilizar a classe `SGDClassifier` com `SGDClassifier(loss="hinge", alpha=1/(m*C))`, o que aplicará o Gradiente Descendente Estocástico regular (ver Capítulo 4) no treinamento de um classificador linear SVM. Ele não converge tão rápido quanto a classe `LinearSVC`, mas pode ser útil para manipular grandes conjuntos de dados que não cabem na memória (treinamento out-of-core), ou para manipular tarefas de classificação online.



A classe `LinearSVC` regulariza o termo de polarização, então você deve centrar primeiro o conjunto de treinamento subtraindo sua média. Este processo é automatizado se você escalar os dados utilizando o `StandardScaler`. Além disso, certifique-se de configurar o hiperparâmetro `loss` para "hinge", pois ele não é o valor padrão. Finalmente, para um melhor desempenho, você deve configurar o hiperparâmetro `dual` para `False`, a menos que existam mais características do que instâncias de treinamento (discutiremos a dualidade ainda neste capítulo).

## Classificação SVM Não Linear

Embora os classificadores lineares SVM sejam eficientes e funcionem surpreendentemente bem em muitos casos, muitos conjuntos de dados nem sequer estão próximos de serem linearmente separáveis. Adicionar mais características, como as polinomiais (como você fez no Capítulo 4) é uma abordagem válida para lidar com os conjuntos de dados não lineares; em alguns casos, isso pode resultar em um conjunto de dados linearmente separável. Considere a plotagem à esquerda na Figura 5-5: ela representa um conjunto de dados simples com apenas uma característica  $x_1$ . Como você pode ver, este conjunto de dados não é linearmente separável. Mas, se adicionarmos uma segunda característica,  $x_2 = (x_1)^2$ , o conjunto de dados 2D resultante será perfeitamente separável linearmente.



*Figura 5-5. Adicionando características para tornar um conjunto de dados linearmente separável*

Para implementar esta ideia usando o Scikit-Learn, criamos um `Pipeline` contendo um transformador `PolynomialFeatures` (discutido em “Regressão Polinomial” na página 123), seguido por uma `StandardScaler` e uma `LinearSVC`. Testaremos isso no conjunto de dados em formato de luas (veja a Figura 5-6):

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```

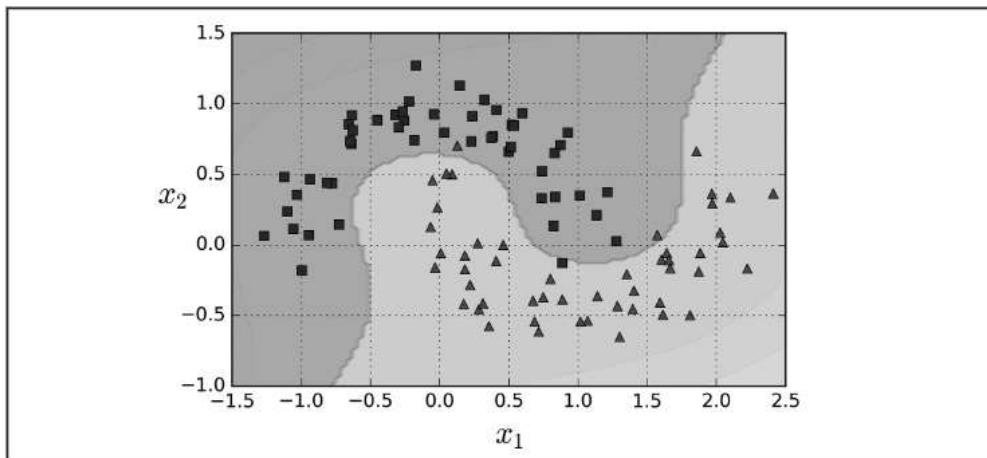


Figura 5-6. Classificador Linear SVM com a utilização de características polinomiais

## Kernel Polinomial

Implementar a adição de características polinomiais é simples e pode funcionar bem em todos os tipos de algoritmos de Aprendizado de Máquina (não apenas as SVMs), mas com um baixo grau polinomial não podemos lidar com conjuntos de dados muito complexo, e com um alto grau polinomial criamos um grande número de características, o que torna o modelo muito mais lento.

Felizmente, ao utilizar as SVM, aplicamos uma técnica matemática quase milagrosa chamada *truque do kernel* (que já será explicado). Isso permite obter o mesmo resultado como se você adicionasse inúmeras características polinomiais, mesmo com polinômios de alto grau, sem realmente precisar adicioná-las. Portanto, não há nenhuma explosão combinatória da quantidade de características, pois não adicionamos nenhuma na verdade. Este truque é implementado pela classe `SVC`. Vamos testá-lo no conjunto de dados em formato de luas:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

Ao utilizar um kernel polinomial de 3º grau, este código treina um classificador SVM. Ele está representado à esquerda da Figura 5-7. À direita, temos outro classificador SVM utilizando um kernel polinomial de 10º grau. Obviamente, se o seu modelo estiver se sobreajustando é necessário reduzir seu grau polinomial. Por outro lado, se estiver se subajustando, é preciso tentar aumentá-lo. O hiperparâmetro `coef0` controla o quanto o modelo é influenciado por polinômios de alto grau versus polinômios de baixo grau.

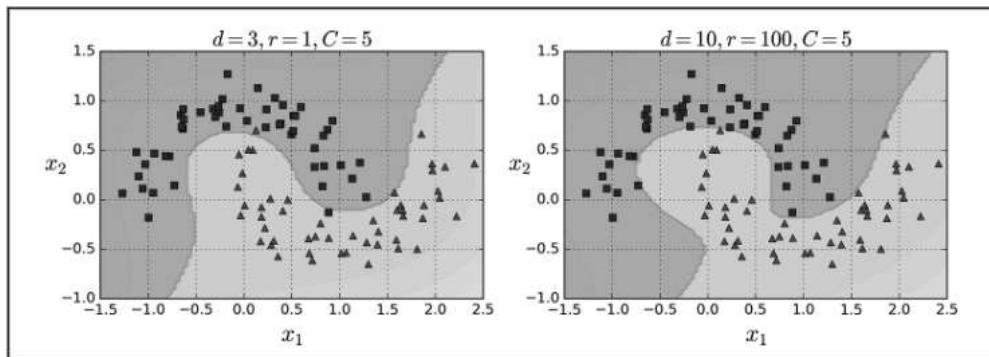


Figura 5-7. Classificadores SVM com um kernel polinomial



Uma abordagem comum para encontrarmos os valores corretos do hiperparâmetro é utilizar a busca em grade (consulte o Capítulo 2). Muitas vezes é mais rápido fazer uma grid search grosseira e depois uma grid search mais refinada em torno dos melhores valores encontrados. Ter bom senso sobre o que cada hiperparâmetro realmente faz também pode ajudar você a procurar na parte certa do espaço do hiperparâmetro.

## Adicionando Características de Similaridade

Outra técnica para se resolver problemas não lineares é a adição de características calculadas ao utilizar uma *função de similaridade*, que mede o quanto cada instância se assemelha a um *ponto de referência* específico. Por exemplo, vamos pegar o conjunto de dados unidimensional discutido anteriormente e adicionar dois pontos de referência em  $x_1 = -2$  e  $x_1 = 1$  (veja a plotagem à esquerda na Figura 5-8). A seguir, definimos a função de similaridade como *Função de Base Radial* (RBF, em inglês) Gaussiana com  $\gamma = 0,3$  (veja a Equação 5-1).

*Equação 5-1. RBF Gaussiana*

$$\phi_\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

É uma função em forma de sino que varia de 0 (muito longe do ponto de referência) a 1 (no ponto de referência). Agora estamos prontos para calcular as novas características. Por exemplo, observe a instância  $x_1 = -1$ : localizada a uma distância de 1 do primeiro ponto de referência e 2 do segundo. Portanto, suas novas características são  $x_2 = \exp(-0,3 \times 1^2) \approx 0,74$  e  $x_3 = \exp(-0,3 \times 2^2) \approx 0,30$ . A plotagem à direita da Figura 5-8 mostra o conjunto de dados transformado (descartando as características originais). Como você pode ver, agora ele é linearmente separável.

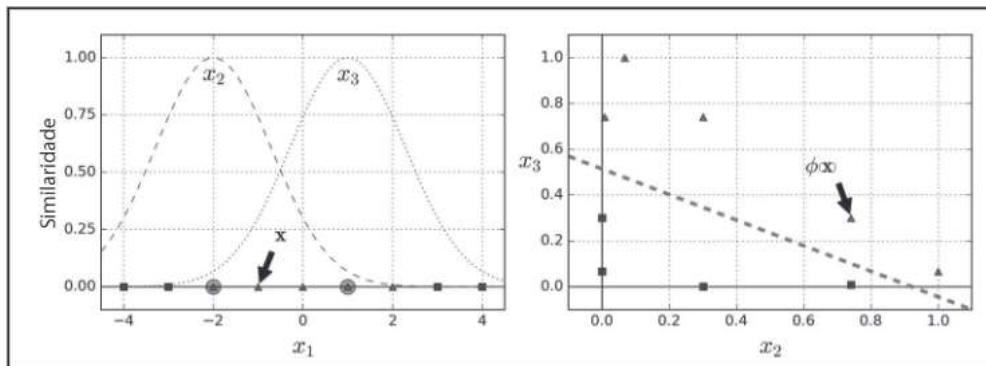


Figura 5-8. Características de similaridade utilizando a RBF Gaussiana

Você deve estar se perguntando como podemos selecionar os pontos de referência. A abordagem mais simples seria criar um ponto na localização de cada instância no conjunto de dados. Isso criará muitas dimensões e aumentará as chances de o conjunto de treinamento transformado ser linearmente separável. A desvantagem é que um conjunto de treinamento com  $m$  instâncias e  $n$  características se transforma em um conjunto de treinamento com  $m$  instâncias e  $m$  características (assumindo que você descarte as características originais). Se o conjunto for muito grande, você ficará com um número igualmente grande de características.

## Kernel RBF Gaussiano

Assim como o método de características polinomiais, o método de características de similaridade pode ser útil com qualquer algoritmo do Aprendizado de Máquina, mas poderá ter um custo computacional elevado para calcular todas as características adicionais, especialmente em grandes conjuntos de treinamento. No entanto, mais uma vez, o truque do *kernel* faz sua mágica na SVM: possibilita a obtenção de um resultado semelhante, como se você tivesse adicionado muitas características de similaridade sem realmente precisar adicioná-las. Vamos tentar o *kernel RBF Gaussiano* com a utilização da classe SVC:

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
])
rbf_kernel_svm_clf.fit(X, y)
```

Este modelo está representado na parte inferior esquerda da Figura 5-9. As outras plotagens mostram modelos treinados com diferentes valores dos hiperparâmetros `gamma` ( $\gamma$ ) e `C`. Aumentar `gamma` estreitará a curva em forma de sino (veja a plotagem à esquerda da Figura 5-8), e como resultado cada raio de influência da instância será menor: mexer

ao redor de instâncias individuais torna a fronteira de decisão mais irregular. Por outro lado, um pequeno valor de `gamma` torna a curva em forma de sino mais ampla, de modo que as instâncias ficam com um maior raio de influência e o limite de decisão fica mais suave. Então  $\gamma$  age como um hiperparâmetro de regularização: se seu modelo estiver sobreajustado você deve reduzi-lo e, se estiver subajustado, você deve aumentá-lo (como o hiperparâmetro  $C$ ).

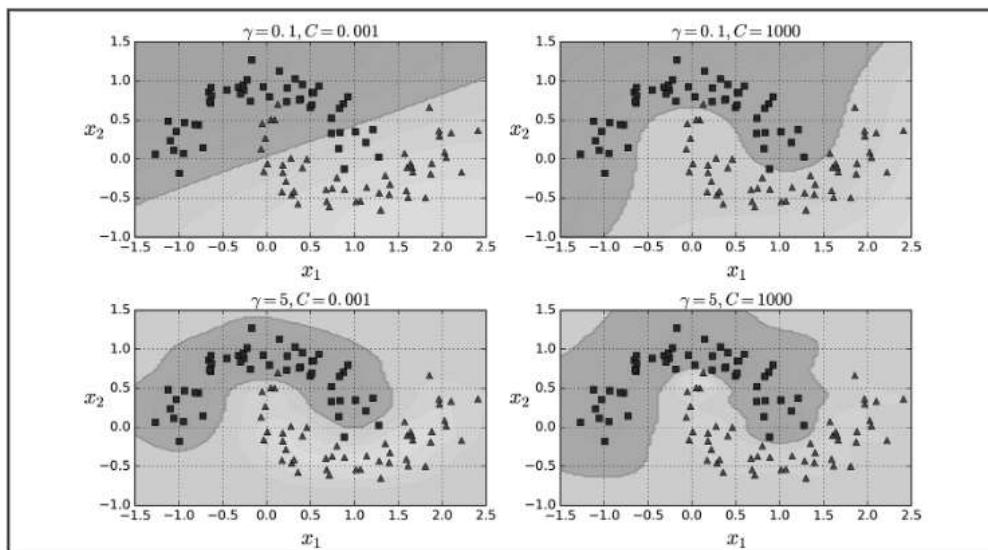


Figura 5-9. Classificadores SVM utilizando o kernel RBF

Existem outros kernels, mas eles são utilizados muito mais raramente. Por exemplo, alguns kernels são especializados em estruturas de dados específicas. Algumas vezes, *string kernels* são utilizados na classificação de documentos de textos ou sequências de DNA (por exemplo, ao utilizar o *string subsequence kernel* ou kernels baseados em *distância Levenshtein*).



Com tantos kernels para escolher, como decidir qual deles utilizar? Como regra geral, devemos sempre tentar o kernel linear primeiro (lembre-se de que o `LinearSVC` é muito mais rápido do que o `SVC` (`kernel="linear"`)), especialmente se o conjunto de treinamento for muito grande ou se tiver muitas características. Se o conjunto de treinamento não for muito grande, você também deve tentar o kernel RBF Gaussiano; ele funciona bem na maioria dos casos. Se tiver tempo livre e poder de computação, experimente também alguns outros kernels que utilizam a validação cruzada e a grid search, especialmente se houver kernels especializados para a estrutura de dados do seu conjunto de treinamento.

## Complexidade Computacional

A classe `LinearSVC` é baseada na biblioteca *liblinear*, que implementa um algoritmo otimizado (<http://goo.gl/R635CH>) para o SVM linear.<sup>1</sup> Ele não suporta o truque do kernel, mas escalona quase linearmente com o número de instâncias de treinamento e o número de características: sua complexidade de tempo de treinamento é mais ou menos  $O(m \times n)$ .

O algoritmo demorará mais se você precisar de uma precisão muito alta. Isso é controlado pelo hiperparâmetro de tolerância  $\epsilon$  (chamado `tol` no Scikit-Learn). Na maioria das tarefas de classificação, a tolerância padrão é suficiente.

A classe `SVC` é baseada na biblioteca *libsvm*, que implementa um algoritmo (<http://goo.gl/a8HkE3>) que suporta o truque do *kernel*.<sup>2</sup> A complexidade do período de treinamento geralmente está entre  $O(m^2 \times n)$  e  $O(m^3 \times n)$ . Infelizmente, isso significa que ele fica terrivelmente lento quando o número de instâncias de treinamento cresce (por exemplo, centenas de milhares de instâncias). Este algoritmo é perfeito para conjuntos de treinamento complexos, sejam eles pequenos ou médios. No entanto, ele se escalona bem com o número de características, especialmente com *características esparsas* (ou seja, quando cada instância possui poucas características diferentes de zero). Neste caso, o algoritmo dimensiona grosseiramente com o número médio de características diferentes de zero por instância. A Tabela 5-1 compara as classes de classificação SVM do Scikit-Learn.

Tabela 5-1. Comparação das classes do Scikit-Learn para a classificação SVM

Classe	Complexidade do período	Suporte out-of-core	Escalonamento requerido	Truque do kernel requerido
<code>LinearSVC</code>	$O(m \times n)$	Não	Sim	Não
<code>SGDClassifier</code>	$O(m \times n)$	Sim	Sim	Não
<code>SVC</code>	$O(m^2 \times n)$ to $O(m^3 \times n)$	Não	Sim	Sim

## Regressão SVM

Como mencionamos anteriormente, o algoritmo SVM é bastante versátil: ele não só suporta a classificação linear e não linear, mas também a Regressão Linear e não linear. O truque é reverter o objetivo: ao invés de tentar preencher a maior via possível entre duas classes, limitando as violações da margem, a Regressão SVM tenta preencher o maior número possível de instâncias *na* via limitando as violações da margem (ou seja, instâncias *fora*

1 “A Dual Coordinate Descent Method for Large-scale Linear SVM”, Lin *et al.*(2008).

2 “Sequential Minimal Optimization (SMO)”, J. Platt (1998).

da via). A largura da via é controlada por um hiperparâmetro  $\epsilon$ . A Figura 5-10 mostra dois modelos SVM de Regressão Linear treinados em algum dado linear aleatório, um com uma margem larga ( $\epsilon = 1,5$ ) e o outro com uma margem pequena ( $\epsilon = 0,5$ ).

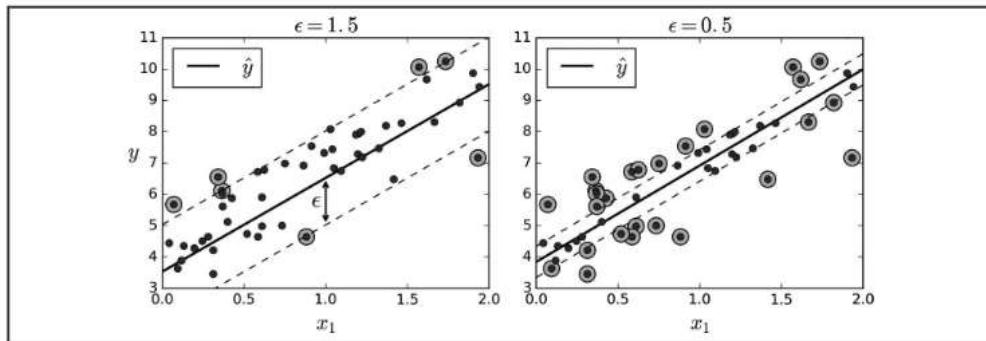


Figura 5-10. Regressão SVM

Adicionar mais instâncias de treinamento dentro da margem não afeta as previsões do modelo; assim, o modelo é chamado de  $\epsilon$ -insensitive.

Você pode utilizar a classe `LinearSVR` do Scikit-Learn para executar a regressão SVM linear. O código a seguir produz o modelo representado à esquerda na Figura 5-10 (os dados de treinamento devem ser escalonados e centrados primeiro):

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Para enfrentar tarefas de regressão não linear você pode utilizar um modelo SVM kernelizado. Por exemplo, a Figura 5-11 mostra a Regressão SVM em um conjunto de treinamento quadrático aleatório utilizando um kernel polinomial de 2º grau. Existe pouca regularização na plotagem à esquerda (isto é, um grande valor  $C$ ) e muito mais regularização na plotagem à direita (isto é, um pequeno valor  $C$ ).

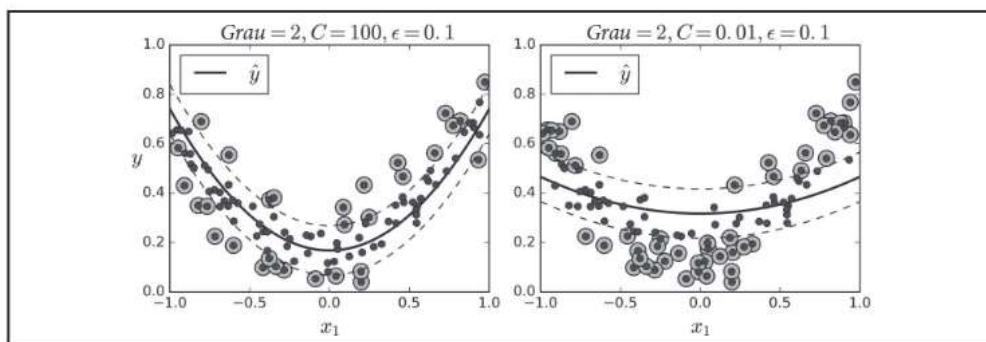


Figura 5-11. Regressão SVM utilizando um kernel polinomial de 2º grau

O código a seguir produz o modelo representado à esquerda da Figura 5-11 usando a classe `SVR` do Scikit-Learn (que suporta o truque do kernel). A classe `SVR` é o equivalente de regressão da classe `SVC`, e a classe `LinearSVR` é o equivalente de regressão da classe `LinearSVC`. A classe `LinearSVR` escalona linearmente com o tamanho do conjunto de treinamento (assim como a classe `LinearSVC`), enquanto a classe `SVR` fica muito mais lenta à medida que o conjunto de treinamento cresce (assim como a classe `SVC`).

```
from sklearn.svm import SVR  
  
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)  
svm_poly_reg.fit(X, y)
```



As SVM também podem ser utilizadas para a detecção de outliers; veja a documentação do Scikit-Learn para mais detalhes.

## Nos Bastidores

Esta seção explica como as SVM fazem previsões e como funcionam seus algoritmos de treinamento, começando com classificadores SVM lineares. Você pode ignorá-la e seguir direto para os exercícios no final deste capítulo se estiver apenas começando com o Aprendizado de Máquina, e voltar mais tarde quando quiser obter uma compreensão mais profunda das SVM.

Primeiro, uma palavra sobre notações: no Capítulo 4 utilizamos a convenção de colocar todos os parâmetros dos modelos em um vetor  $\theta$ , incluindo o termo de polarização  $\theta_0$  e o peso da característica de entrada de  $\theta_1$  a  $\theta_n$ , e adicionamos uma entrada de polarização  $x_0 = 1$  para todas instâncias. Neste capítulo, utilizaremos uma convenção diferente, que é mais conveniente (e mais comum) ao lidarmos com as SVM: o termo de polarização será chamado  $b$  e o vetor do peso das características será chamado  $w$ . Nenhuma característica de polarização será adicionada aos vetores das características de entrada.

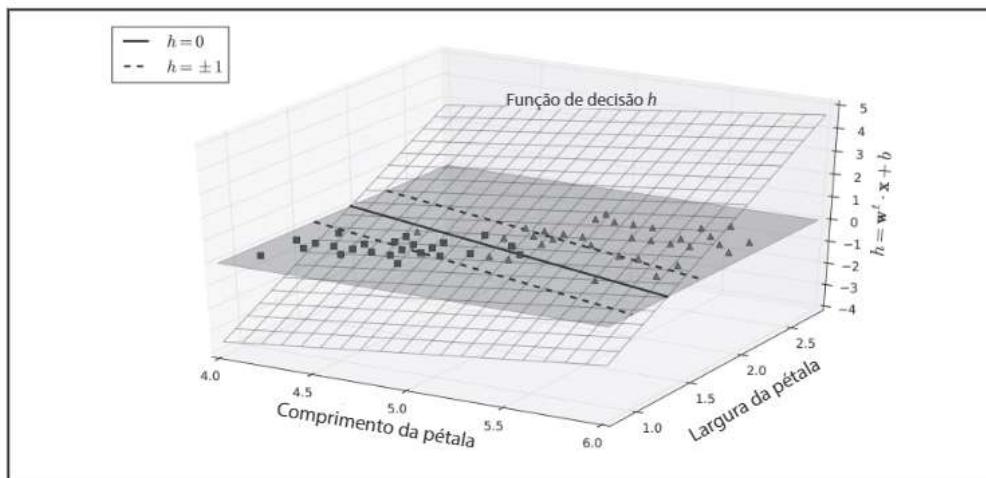
## Função de Decisão e Previsões

O modelo de classificador SVM linear prevê a classe de uma nova instância  $x$  calculando a função de decisão  $w^T \cdot x + b = w_1 x_1 + \dots + w_n x_n + b$ : se o resultado for positivo, a classe prevista  $\hat{y}$  será a classe positiva (1), ou então será a classe negativa (0); veja a Equação 5-2.

*Equação 5-2. Previsão do classificador SVM linear*

$$\hat{y} = \begin{cases} 0 & \text{se } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 & \text{se } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

A Figura 5-12 mostra a função de decisão que corresponde ao modelo à direita da Figura 5-4: é um plano bidimensional, uma vez que este conjunto de dados tem duas características (largura e comprimento da pétala). O limite de decisão é o conjunto de pontos no qual a função de decisão é igual a 0: é a interseção de dois planos, que é uma linha reta (representada pela linha sólida espessa).<sup>3</sup>



*Figura 5-12. Função de decisão para o conjunto de dados da íris*

As linhas tracejadas representam os pontos nos quais a função de decisão é igual a 1 ou a -1: são paralelas e na mesma distância da fronteira de decisão, formando uma margem ao seu redor. Treinar um classificador SVM linear significa encontrar os valores de  $w$  e  $b$  que tornam esta margem o mais ampla possível ao mesmo tempo em que evitam violações de margem (*margem rígida*) ou as limitam (*margem suave*).

## Objetivo do Treinamento

Considere a inclinação da função de decisão: ela é igual à norma do vetor de peso,  $\| w \|$ . Se dividirmos esta inclinação por 2, os pontos em que a função de decisão é igual a  $\pm 1$  estarão duas vezes mais distantes da fronteira de decisão. Em outras palavras, dividir a

<sup>3</sup> De uma forma mais geral, quando houver  $n$  características, a função de decisão será um *hiperplano*  $n$ -dimensional, e o limite de decisão será um hiperplano  $(n-1)$  dimensional.

inclinação por 2 multiplicará a margem por 2. Talvez seja mais fácil visualizar em 2D na Figura 5-13. Quanto menor o vetor de peso  $w$ , maior a margem.

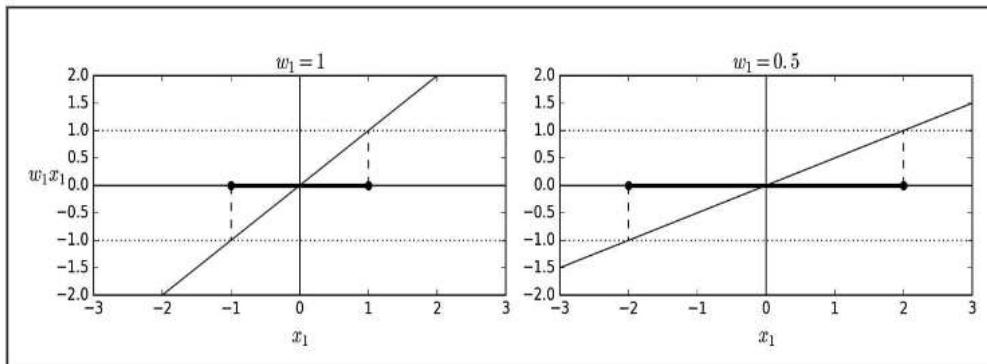


Figura 5-13. Um vetor de peso menor resulta em uma margem maior

Então, queremos minimizar  $\|w\|$  para obter uma margem maior. Entretanto, se também quisermos evitar qualquer violação da margem (*margem rígida*), precisamos que a função de decisão seja maior que 1 para todas as instâncias positivas de treinamento e menor que -1 para as instâncias negativas. Se definirmos  $t^{(i)} = -1$  para instâncias negativas (se  $y^{(i)} = 0$ ) e  $t^{(i)} = 1$  para instâncias positivas (se  $y^{(i)} = 1$ ), então poderemos expressar esta restrição como  $t^{(i)}(w^T \cdot x^{(i)} + b) \geq 1$  para todas as instâncias.

Podemos, portanto, expressar o objetivo do classificador SVM linear com margem rígida como o problema de *otimização restrita* na Equação 5-3.

Equação 5-3. Objetivo do classificador SVM linear com margem rígida

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimizar}} \quad \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \\ & \text{sujeito a} \quad t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{para } i = 1, 2, \dots, m \end{aligned}$$



Estamos minimizando  $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$ , que é igual a  $\frac{1}{2} \| \mathbf{w} \|^2$ , em vez de minimizar  $\| \mathbf{w} \|$ . Isto se dá porque obteremos o mesmo resultado (uma vez que os valores de  $w$  e  $b$  que minimizam um valor também minimizam metade do seu quadrado), mas  $\frac{1}{2} \| \mathbf{w} \|^2$  tem uma derivada simples (apenas  $w$ ) enquanto  $\| \mathbf{w} \|$  não é diferenciável em  $\mathbf{w} = \mathbf{0}$ . Algoritmos de otimização funcionam bem melhor em funções diferenciáveis.

Para obtermos o objetivo da margem suave, precisamos introduzir uma *variável de folga*  $\zeta^{(i)} \geq 0$  para cada instância:<sup>4</sup>  $\zeta^{(i)}$  mede quanto a i-ésima instância pode violar a margem. Agora, temos dois objetivos conflitantes: tornar as variáveis de folga o menor possível para reduzir as violações da margem, e tornar  $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$  o menor possível para aumentar a margem. É aqui que o hiperparâmetro  $C$  entra em ação: ele nos permite definir a troca entre esses dois objetivos, que nos dá o problema de otimização restrita na Equação 5-4.

*Equação 5-4. Objetivo do classificador SVM linear de margem suave*

$$\begin{aligned} & \text{minimizar}_{\mathbf{w}, b, \zeta} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{sujeito a} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{e} \quad \zeta^{(i)} \geq 0 \quad \text{para } i = 1, 2, \dots, m \end{aligned}$$

## Programação Quadrática

Os problemas de *margem rígida* e *margem suave* são ambos de otimização quadrática convexa com restrições lineares. Esses problemas são conhecidos como problemas de *programação quadrática* (QP, em inglês). Muitos solucionadores *off-the-shelf* que usam técnicas fora do escopo deste livro estão disponíveis para resolver problemas de programação quadrática.<sup>5</sup> A formulação geral do problema é dada pela Equação 5-5.

*Equação 5-5. Problema da programação quadrática*

$$\begin{aligned} & \text{Minimize}_{\mathbf{p}} \quad \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ & \text{sujeito a} \quad \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \\ & \text{sendo que} \quad \left| \begin{array}{l} \mathbf{p} \text{ é um vetor dimensional } n_p \text{ (} n_p = \text{número de parâmetros}), \\ \mathbf{H} \text{ é uma matriz } n_p \times n_p, \\ \mathbf{f} \text{ é um vetor } n_p\text{-dimensional} \\ \mathbf{A} \text{ é uma matriz } n_c \times n_p \text{ (} n_c = \text{número de restrições}), \\ \mathbf{b} \text{ é um vetor } n_c\text{-dimensional} \end{array} \right. \end{aligned}$$

Note que a expressão  $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$  na verdade define  $n_c$  restrições:  $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$  para  $i = 1, 2, \dots, n_c$ , sendo  $\mathbf{a}^{(i)}$  o vetor que contém os elementos da i-ésima linha de  $\mathbf{A}$  e  $b^{(i)}$  é o i-ésimo elemento de  $\mathbf{b}$ .

<sup>4</sup> Zeta ( $\zeta$ ) é a sexta letra do alfabeto Grego.

<sup>5</sup> Para saber mais sobre a Programação Quadrática, você pode começar lendo Stephen Boyd e Lieven Vandenberghe, Convex Optimization (<http://goo.gl/FGXuLw>) (Cambridge, Reino Unido: Cambridge University Press, 2004) ou assistindo à série de conferências em vídeo de Richard Brown (<http://goo.gl/rTo3Af>).

Você pode verificar facilmente que, se você definir os parâmetros QP da maneira a seguir, obterá o objetivo do classificador SVM linear com margem rígida:

- $n_p = n + 1$ , sendo que  $n$  é o número de características (o  $+1$  é para o termo de polarização);
- $n_c = m$ , sendo que  $m$  é o número de instâncias de treinamento;
- $H$  é a matriz de identidade  $n_p \times n_p$ , exceto com um zero na célula acima à esquerda (para ignorar o termo de polarização);
- $f = 0$ , um vetor  $n_p$ -dimensional cheio de 0s;
- $b = 1$ , um vetor  $n_c$ -dimensional cheio de 1s;
- $a^{(i)} = -t^{(i)} \dot{x}^{(i)}$  sendo que  $\dot{x}^{(i)}$  é igual a  $x^{(i)}$  com uma característica de viés extra  $\dot{x}_0 = 1$ .

Portanto, uma forma de treinar um classificador SVM linear com margem rígida é apenas utilizar um solucionador de QP off-the-shelf passando-lhe os parâmetros anteriores. O vetor resultante  $p$  conterá o termo de polarização  $b = p_0$  e os pesos das características  $w_i = p_i$  para  $i = 1, 2, \dots, m$ . Você também pode utilizar um solucionador QP para resolver o problema da margem suave (veja os exercícios no final do capítulo).

Entretanto, para utilizar o truque do kernel, analisaremos um problema diferente de otimização restrita.

## O Problema Dual

Dado um problema de otimização restrita, conhecido como o *problema primal*, é possível expressar um problema diferente, porém intimamente relacionado, chamado *problema dual*. A solução para o problema dual tipicamente fornece um limite menor para a solução do problema primal, mas em algumas condições pode ter as mesmas soluções do problema primal. Felizmente, o problema da SVM atende essas condições,<sup>6</sup> então você pode optar por resolver o problema primal ou o problema dual, pois ambos terão a mesma solução. A equação 5-6 mostra a forma dual do objetivo da SVM linear (se você estiver interessado em saber como derivar o problema dual do problema primal, consulte o Apêndice C).

*Equação 5-6. Forma dual do objetivo SVM linear*

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ \text{sujeito a} \quad & \alpha^{(i)} \geq 0 \text{ para } i = 1, 2, \dots, m \end{aligned}$$

---

<sup>6</sup> A função objetiva é convexa, e as restrições de desigualdade são funções continuamente diferenciáveis e convexas.

Uma vez encontrado o vetor  $\alpha$  que minimiza esta equação (utilizando um solucionador de QP), você pode calcular  $\hat{\mathbf{w}}$  e  $\hat{b}$  utilizando a Equação 5-7, que minimiza o problema primal.

*Equação 5-7. Da solução dual à solução primal*

$$\begin{aligned}\hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left( t^{(i)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)} \right)\end{aligned}$$

É mais rápido resolver o problema dual do que o primal quando o número de instâncias de treinamento for menor que o número de características. Mais importante, possibilita o truque do kernel, já o primal, não. Então, o que é esse truque do kernel, afinal?

## SVM Kernelizado

Suponha que você deseja aplicar uma transformação polinomial de 2º grau para um conjunto de treinamento bidimensional (como o conjunto de treinamento em formato de luas) e, em seguida, treinar um classificador SVM linear no conjunto transformado. A Equação 5-8 mostra a função de mapeamento polinomial de 2º grau  $\phi$  que você deseja aplicar.

*Equação 5-8. Mapeamento polinomial de segundo grau*

$$\phi(\mathbf{x}) = \phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Observe que o vetor transformado é tridimensional em vez de bidimensional. Agora, vejamos o que acontece com um par de vetores bidimensionais,  $\mathbf{a}$  e  $\mathbf{b}$ , se aplicarmos este mapeamento polinomial de segundo grau e depois calcularmos o produto escalar dos vetores transformados (veja a Equação 5-9).

*Equação 5-9. Truque do Kernel para um mapeamento polinomial de 2º grau*

$$\begin{aligned}\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2\end{aligned}$$

Que tal? O produto escalar dos vetores transformados é igual ao quadrado do produto escalar dos vetores originais:  $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ .

Esta é a visão chave: se você aplicar a transformação  $\phi$  para todas as instâncias de treinamento, então o problema dual (veja Equação 5-6) conterá o produto escalar  $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$ . Mas, se  $\phi$  for a transformação polinomial de segundo grau definida na Equação 5-8, então você pode substituir este produto escalar de vetores transformados simplesmente por  $(\mathbf{x}^{(i)}^T \cdot \mathbf{x}^{(j)})^2$ . Você não precisa realmente transformar as instâncias de treinamento: apenas substitua o produto escalar por seu quadrado na Equação 5-6. O resultado será estritamente o mesmo que seria se você passasse pelo problema de realmente transformar o conjunto de treinamento e então ajustar um algoritmo SVM linear, mas esse truque torna todo o processo muito mais eficiente computacionalmente. Esta é a essência do truque do kernel.

A função  $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$  é chamada de *kernel polinomial de 2º grau*. No Aprendizado de Máquina, um kernel é uma função capaz de calcular o produto escalar  $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$  com base somente nos vetores originais  $\mathbf{a}$  e  $\mathbf{b}$  sem ter que calcular (ou mesmo saber) a transformação  $\phi$ . A equação 5-10 lista alguns dos kernels mais utilizados.

#### *Equação 5-10. Kernels comuns*

$$\text{Linear: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polinomial: } K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$$

$$\text{RBF Gaussiano: } K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$$

$$\text{Sigmóide: } K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \cdot \mathbf{b} + r)$$

### Teorema de Mercer

De acordo com o *Teorema de Mercer*, se uma função  $K(\mathbf{a}, \mathbf{b})$  respeita algumas condições matemáticas chamadas *condições de Mercer* ( $K$  deve ser contínuo, simétrico em seus argumentos, de modo que  $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$ , etc.), então existe uma função  $\phi$  que mapeia  $\mathbf{a}$  e  $\mathbf{b}$  em outro espaço (possivelmente com dimensões muito maiores) tal que  $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$ . Então, você pode utilizar  $K$  como um kernel já que você sabe que  $\phi$  existe, mesmo não sabendo o que  $\phi$  é. No caso do kernel RBF Gaussiano, pode-se mostrar que  $\phi$ , na verdade, mapeia cada instância de treinamento para um espaço de dimensões infinitas, por isso é bom que você não precise realmente executar o mapeamento!

Observe que alguns kernels utilizados com frequência (como o kernel Sigmóide) não respeitam todas as condições de Mercer, mas geralmente funcionam bem na prática.

Ainda há uma extremidade solta que devemos amarrar. A Equação 5-7 mostra como passar da solução dual para a solução primal no caso de um classificador SVM linear, mas

se você aplicar o truque do kernel ficará com equações que incluem  $\phi(x^{(i)})$ . Na verdade,  $w$  deve ter o mesmo número de dimensões que  $\phi(x^{(i)})$ , que pode ser enorme ou mesmo infinito, então você não consegue calculá-lo. Mas como você pode fazer previsões sem conhecer  $w$ ? Bem, a boa notícia é que você pode inserir a fórmula da Equação 5-7 em  $w$  na função de decisão para uma nova instância  $x^{(n)}$  e obterá uma equação apenas com produtos escalares entre vetores de entrada, possibilitando a utilização do truque do kernel mais uma vez (Equação 5-11).

*Equação 5-11. Fazendo previsões com um SVM kernelizado*

$$\begin{aligned} h_{\widehat{W}, \widehat{b}}(\phi(x^{(n)})) &= \widehat{W}^T \cdot \phi(x^{(n)}) + \widehat{b} = \left( \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} \phi(x^{(i)}) \right)^T \cdot \phi(x^{(n)}) + \widehat{b} \\ &= \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} (\phi(x^{(i)})^T \cdot \phi(x^{(n)})) + \widehat{b} \\ &= \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \widehat{\alpha}^{(i)} t^{(i)} K(x^{(i)}, x^{(n)}) + \widehat{b} \end{aligned}$$

Note que, uma vez que  $\alpha^{(i)} \neq 0$  apenas para vetores de suporte, fazer previsões envolve o cálculo do produto escalar do novo vetor de entrada  $x^{(n)}$  com apenas os vetores de suporte e não todas as instâncias de treinamento. Claro, você também precisa calcular o termo de polarização  $b$  utilizando o mesmo truque (Equação 5-12).

*Equação 5-12. Calculando o termo de polarização por meio do truque do kernel*

$$\begin{aligned} \widehat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \widehat{W}^T \cdot \phi(x^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \left( \sum_{j=1}^m \widehat{\alpha}^{(j)} t^{(j)} \phi(x^{(j)}) \right)^T \cdot \phi(x^{(i)}) \right) \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \sum_{\substack{j=1 \\ \widehat{\alpha}^{(j)} > 0}}^m \widehat{\alpha}^{(j)} t^{(j)} K(x^{(i)}, x^{(j)}) \right) \end{aligned}$$

Se você já está começando a ter dores de cabeça, saiba que isto é perfeitamente normal: é um efeito colateral infeliz do truque do kernel.

## SVM Online

Antes de concluir este capítulo, daremos uma rápida olhada nos classificadores de SVM online (lembre-se de que o aprendizado online significa aprender de forma incremental, tipicamente quando chegam novas instâncias).

Um método para minimizar a função de custo na Equação 5-13, que é derivada do problema primal, é utilizar o Gradiente Descendente (por exemplo, utilizando o `SGDClassifier`) para os classificadores SVM lineares. Infelizmente, ele converge muito mais lentamente do que os métodos baseados em QP.

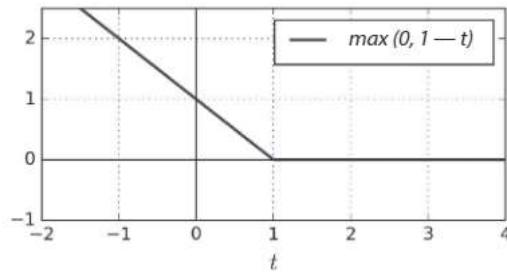
*Equação 5-13. Função de custo do classificador SVM linear*

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))$$

A primeira soma na função de custo levará o modelo a ter um pequeno vetor de peso  $\mathbf{w}$ , aumentando a margem. A segunda soma calcula o total de todas as violações da margem. A violação da margem de uma instância é igual a 0 se estiver localizada fora da via e no lado correto, ou então é proporcional à distância ao lado correto da via. Minimizar esse termo garante que o modelo diminua o tamanho e a quantidade de violações da margem o máximo possível.

### Hinge Loss

A função  $\max(0, 1 - t)$  é chamada de função *hinge loss* [perda de articulação, em tradução livre] (representada abaixo). É igual a 0 quando  $t \geq 1$ . Sua derivada (inclinação) é igual a -1 se  $t < 1$  e 0 se  $t > 1$ . Ela não é diferenciável em  $t = 1$ , mas, assim como para a Regressão Lasso (veja “Regressão Lasso” no Capítulo 4), utilizando qualquer *subderivada* em  $t = 1$ , você ainda pode utilizar o Gradiente Descendente (ou seja, qualquer valor entre -1 e 0).



Também é possível implementar SVM kernelizadas online, por exemplo, utilizando “Incremental and Decremental SVM Learning” (<http://goo.gl/JEqVui>)<sup>7</sup> ou “Fast Kernel Classifiers with Online and Active Learning” (<https://goo.gl/hsoUHA>).<sup>8</sup> No entanto, estes são implementados em Matlab e C++. Para problemas não lineares em larga escala, considere o uso de redes neurais (ver Parte II).

<sup>7</sup> “Incremental and Decremental Support Vector Machine Learning”, G. Cauwenberghs, T. Poggio (2001).

<sup>8</sup> “Fast Kernel Classifiers with Online and Active Learning”, A. Bordes, S. Ertekin, J. Weston, L. Bottou (2005).

## Exercícios

1. Qual é a ideia fundamental por trás das Máquinas de Vetores de Suporte (SVM)?
2. O que é um vetor de suporte?
3. Por que é importante dimensionar as entradas ao utilizar SVM?
4. Um classificador SVM pode produzir uma pontuação de confiança quando classifica uma instância? E quanto a uma probabilidade?
5. Você deve utilizar a forma primal ou dual do problema SVM no treinamento de um modelo em um conjunto de treinamento com milhões de instâncias e centenas de características?
6. Digamos que você treinou um classificador SVM com o kernel RBF. Parece que ele se subajusta ao conjunto de treinamento: você deve aumentar ou diminuir  $\gamma$  (`gamma`)? E quanto ao  $C$ ?
7. Como você deve configurar os parâmetros QP (`H`, `f`, `A`, e `b`) utilizando um solucionador de QP off-the-shelf para resolver o problema do classificador SVM linear de margem suave?
8. Treine um `LinearSVC` em um conjunto de dados linearmente separável. Depois, treine o `SVC` e um `SGDClassifier` no mesmo conjunto de dados. Veja se você consegue fazer com que eles produzam aproximadamente o mesmo modelo.
9. Treine um classificador SVM no conjunto de dados MNIST. Uma vez que os classificadores SVM são classificadores binários, você precisará utilizar um contra todos para classificar todos os 10 dígitos. Ajuste os hiperparâmetros utilizando pequenos conjuntos de validação para acelerar o processo. Qual acurácia você pode alcançar?
10. Treine um regressor SVM no conjunto de dados imobiliários da Califórnia.

As soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 6

# Árvores de Decisão

Como as SVM, as Árvores de Decisão são algoritmos versáteis de Aprendizado de Máquina que podem executar tarefas de classificação, regressão e, até mesmo, tarefas multioutput. São algoritmos muito poderosos capazes de moldar conjuntos complexos de dados. Por exemplo, no Capítulo 2, ajustando-o perfeitamente (na verdade, sobreajustando), você treinou um modelo `DecisionTreeRegressor` no conjunto de dados imobiliários da Califórnia.

As Árvores de Decisão também são os componentes fundamentais das Florestas Aleatórias (veja o Capítulo 7), que estão entre os algoritmos de Aprendizado de Máquina mais poderosos disponíveis atualmente.

Neste capítulo, começaremos discutindo como treinar, visualizar e fazer previsões com as Árvores de Decisão. Então, passaremos pelo algoritmo de treinamento CART utilizado pelo Scikit-Learn e discutiremos como regularizar árvores e utilizá-las para tarefas de regressão. Finalmente, discutiremos algumas das limitações das Árvores de Decisão.

## Treinando e Visualizando uma Árvore de Decisão

Para entender as Árvores de Decisão, vamos construir uma e dar uma olhada em como ela faz previsões. O código a seguir treina um `DecisionTreeClassifier` no conjunto de dados da íris (consulte o Capítulo 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # comprimento e largura das pétalas
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Você pode visualizar a Árvore de Decisão usando o método `export_graphviz()` para gerar como saída um arquivo de definição de grafo chamado `iris_tree.dot`:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

E, então, você pode converter esse arquivo `.dot` em uma variedade de formatos, como PDF ou PNG, utilizando a ferramenta de linha de comando `dot` do pacote `graphviz`.<sup>1</sup> Esta linha de comando converte o arquivo `.dot` para um arquivo de imagem `.png`:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Sua primeira Árvore de Decisão ficará como a da Figura 6-1.

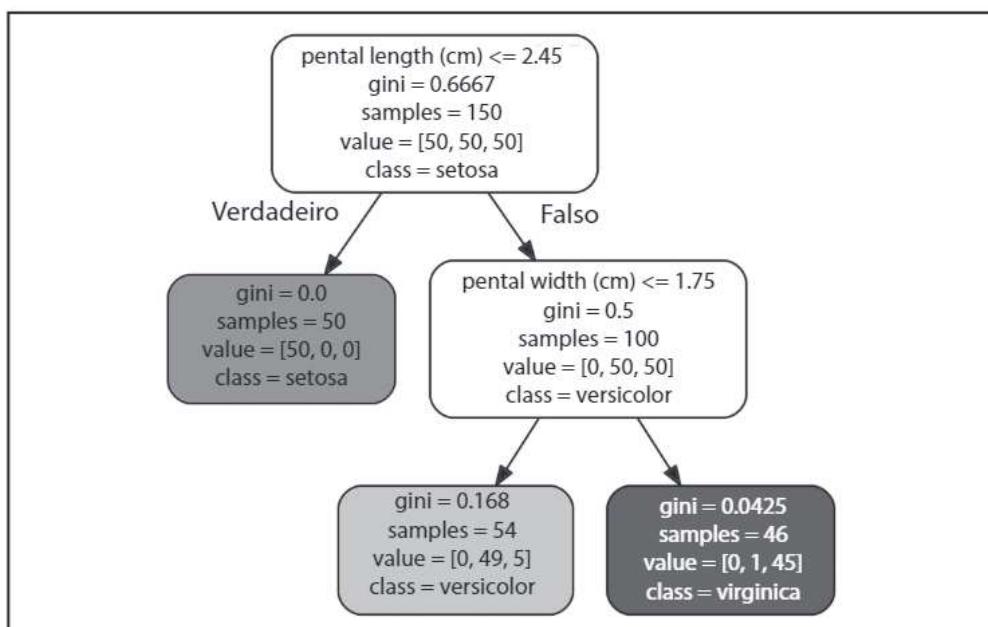


Figura 6-1. Árvore de Decisão da íris

<sup>1</sup> O Graphviz é um pacote de software de visualização de grafos de código aberto, disponível em <http://www.graphviz.org/>.

## Fazendo Previsões

Veja como a árvore representada na Figura 6-1 faz previsões. Suponha que você encontre uma flor da íris e deseje classificá-la. Você começa no *nó da raiz* (profundidade 0, na parte superior): este nó pergunta se o comprimento da pétala da flor é menor do que 2,45cm. Se for, então você se desloca para o nó filho esquerdo da raiz (profundidade 1, esquerda). Neste caso, é um *nó da folha* (isto é, não tem nenhum nó filho), por isso não faz nenhuma pergunta: você pode simplesmente olhar a classe prevista para esse nó e a Árvore de Decisão preverá que sua flor é uma Iris-Setosa (`class=setosa`).

Agora, suponha que você encontre outra flor, mas, desta vez, o comprimento da pétala é maior do que 2,45cm. Você deve se mover para baixo, para o nó filho direito da raiz (profundidade 1, direita), que não é um nó da folha, então ele faz outra pergunta: a largura da pétala é menor que 1,75cm? Se for, então sua flor é provavelmente uma Iris-Versicolor (profundidade 2, esquerda). Caso contrário, é provável que seja uma Iris-Virginica (profundidade 2, direita). É simples assim.



Uma das muitas qualidades das Árvores de Decisão é que elas exigem pouca preparação de dados. Em particular, elas não exigem o escalonamento ou a centralização das características.

O atributo `samples` de um nó conta a quantidade de instâncias de treinamento a que se aplica. Por exemplo, 100 instâncias de treinamento têm um comprimento de pétala maior que 2,45cm (profundidade 1, direita) dentre as quais 54 têm uma largura de pétala menor que 1,75cm (profundidade 2, esquerda). O atributo `value` de um nó lhe diz a quantas instâncias de treinamento de cada classe este nó se aplica: por exemplo, o nó inferior direito se aplica a 0 Iris-Setosa, 1 Iris-Versicolor e 45 Iris-Virginica. Finalmente, o atributo `gini` de um nó mede sua *impureza*: um nó é “puro” ( $gini=0$ ) se todas as instâncias de treinamento pertencem à mesma classe a qual se aplica. Por exemplo, uma vez que o nó esquerdo de profundidade 1 aplica-se apenas às instâncias de treinamento de Iris-Setosa, ele é puro e sua pontuação `gini` é 0. A Equação 6-1 mostra como o algoritmo de treinamento calcula a pontuação  $G_i$  do  $i$ -ésimo nó. Por exemplo, o nó esquerdo de profundidade 2 tem uma pontuação `gini` igual a  $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$ . Outra medida de impureza será discutida brevemente.

*Equação 6-1. Coeficiente de Gini*

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

- $p_{i,k}$  é a média das instâncias da classe  $k$  entre as instâncias de treinamento no nó  $i$ .



O Scikit-Learn utiliza o algoritmo CART, que produz somente *árvores binárias*: os nós sem folhas sempre têm dois filhos (ou seja, as perguntas somente terão respostas sim/não). Entretanto, outros algoritmos, como o ID3, podem produzir Árvores de Decisão com nós que têm mais de dois filhos.

A Figura 6-2 mostra as fronteiras de decisão da Árvore de Decisão. A linha vertical grossa representa a fronteira de decisão do nó raiz (profundidade 0): comprimento da pétala = 2,45cm. Uma vez que a área esquerda é pura (apenas Iris-Setosa), ela não pode ser dividida ainda mais. No entanto, a área direita é impura, então o nódulo direito de profundidade 1 a divide na largura da pétala = 1,75cm (representada pela linha tracejada). Como o `max_depth` foi definido como 2, a Árvore de Decisão para ali. No entanto, se você definir `max_depth` como 3, os dois nós de profundidade 2 adicionarão uma outra fronteira de decisão (representada pelas linhas pontilhadas).

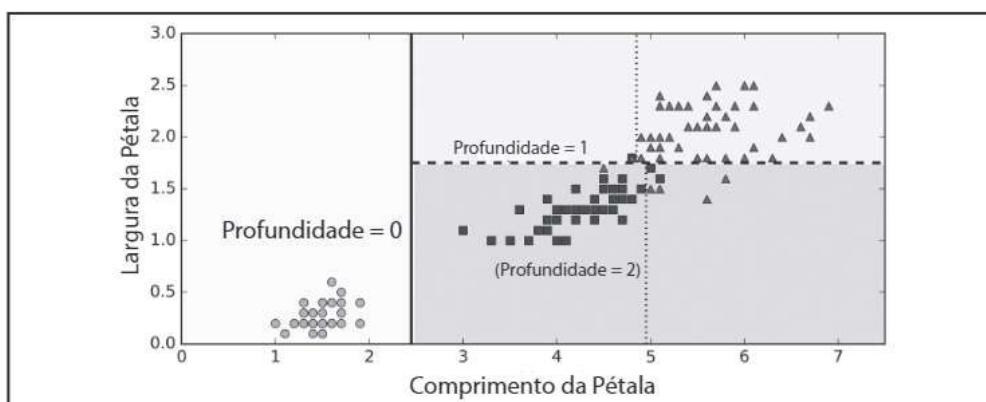


Figura 6-2. Fronteiras de decisão da Árvore de Decisão

### Interpretação do Modelo: Caixa Branca versus Caixa Preta

Como você pode ver, as Árvores de Decisão são bastante intuitivas e é fácil interpretar suas decisões. Esses modelos são geralmente chamados de *modelos de caixa branca*. Em contraste, como veremos, as Florestas Aleatórias ou as redes neurais são, geralmente, consideradas *modelos de caixa preta*. Elas fazem grandes previsões e você pode verificar facilmente os cálculos realizados para a obtenção delas; no entanto, é difícil explicar em termos simples por que essas previsões foram feitas. Por exemplo, se uma rede neural diz que uma pessoa específica aparece em uma imagem, é difícil saber o que realmente contribuiu para essa previsão: o modelo reconheceu os olhos dessa pessoa? Sua boca? Seu nariz? Os sapatos? Ou mesmo o sofá em que ela estava sentada? Por outro lado, as Árvores de Decisão fornecem regras simples de classificação que podem ser aplicadas mesmo manualmente se necessário (por exemplo, para a classificação de uma flor).

## Estimando as Probabilidades de Classes

Uma Árvore de Decisão também pode estimar a probabilidade de uma instância pertencer a uma classe específica  $k$ : primeiro ela atravessa a árvore para encontrar o nó da folha para esta instância e, em seguida, retorna a taxa das instâncias de treinamento da classe  $k$  neste nó. Por exemplo, suponha que ela tenha encontrado uma flor cujas pétalas têm 5cm de comprimento e 1,5cm de largura. O nó da folha correspondente é o nó esquerdo de profundidade 2, portanto a Árvore de Decisão deve exibir as seguintes probabilidades: 0% para Iris-Setosa (0/54), 90,7% para Iris-Versicolor (49/54) e 9,3% para Iris-Virginica (5/54). E, claro, se você pedir que ela preveja a classe, ela deve exibir Iris-Versicolor (classe 1), pois é a que tem a maior probabilidade. Verificaremos isso:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfeito! Repare que as probabilidades estimadas seriam idênticas em qualquer outro lugar no retângulo inferior direito da Figura 6-2 — por exemplo, se as pétalas tivessem 6cm de comprimento e 1,5cm de largura (mesmo que pareça óbvio que provavelmente seria uma Iris-Virginica neste caso).

## O Algoritmo de Treinamento CART

O Scikit-Learn utiliza o algoritmo da *Árvore de Classificação e Regressão* (CART, em inglês) para treinar árvores de decisão (também chamadas de árvores “*em crescimento*”). A ideia é realmente bastante simples: o algoritmo primeiro divide o conjunto de treinamento em dois subconjuntos utilizando uma única característica  $k$  e um limiar  $t_k$  (por exemplo, “comprimento da pétala  $\leq 2,45\text{cm}$ ”). Como ele escolhe  $k$  e  $t_k$ ? Ele busca pelo par  $(k, t_k)$  que produz os subconjuntos mais puros (ponderados pelo tamanho). A função de custo que o algoritmo tenta minimizar é dada pela Equação 6-2.

*Equação 6-2. Função de custo CART para classificação*

$$J(k, t_k) = \frac{m_{\text{esquerda}}}{m} G_{\text{esquerda}} + \frac{m_{\text{direita}}}{m} G_{\text{direita}}$$

sendo que  $\begin{cases} G_{\text{esquerda/direita}} & \text{mede a impureza do subconjunto esquerdo/direito,} \\ m_{\text{esquerda/direita}} & \text{é o número de instâncias no subconjunto esquerdo/direito.} \end{cases}$

Depois de dividir com sucesso o conjunto de treinamento em dois, ele divide os subconjuntos utilizando a mesma lógica, depois os subsubconjuntos e assim por diante, recursivamente. Ele para de recarregar uma vez que atinge a profundidade máxima

(definida pelo hiperparâmetro `max_depth`) ou se não consegue encontrar uma divisão que reduza a impureza. Alguns outros hiperparâmetros (que já serão descritos) controlam as condições adicionais de parada (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` e `max_leaf_nodes`).



Como você pode ver, o algoritmo CART é um *algoritmo ganancioso*: ele busca gananciosamente uma divisão otimizada no nível superior e, em seguida, repete o processo em cada nível. Ele não verifica se a divisão irá ou não resultar na menor impureza possível nos vários níveis abaixo. Um algoritmo ganancioso geralmente produz uma solução razoavelmente boa, mas não é garantia de uma solução ideal.

Infelizmente, encontrar a árvore ideal é conhecido por ser um problema *NP-Completo*:<sup>2</sup> ele requer tempo  $O(\exp(m))$ , tornando o problema intratável mesmo para conjuntos de treinamento muito pequenos. É por isso que devemos nos contentar com uma solução “razoavelmente boa”.

## Complexidade Computacional

Fazer previsões requer percorrer a Árvore de Decisão da raiz até uma folha. Em geral, as árvores de decisão são equilibradas, de modo que percorrê-la requer passar por aproximadamente  $O(\log_2(m))$  nós.<sup>3</sup> Uma vez que cada nó exige apenas a verificação do valor de uma característica, a complexidade geral da previsão será apenas  $O(\log_2(m))$ , independentemente do número de características. Então, as previsões são muito rápidas mesmo se tratando de grandes conjuntos de treinamento.

No entanto, o algoritmo de treinamento compara todas as características (ou menos, se `max_features` estiver definido) em todas as amostras a cada nó. Isto resulta em uma complexidade de treinamento  $O(n \times m \log(m))$ . Ao pré-selecionar os dados para pequenos conjuntos (menos de algumas milhares de instâncias), o Scikit-Learn pode acelerar o treinamento (defina `presort = True`), mas isso diminui significativamente o treinamento para conjuntos maiores.

---

<sup>2</sup> P é o conjunto dos problemas que pode ser resolvido em tempo polinomial. NP é o conjunto de problemas cujas soluções podem ser verificadas em tempo polinomial. Um problema NP-Hard é um problema para o qual qualquer problema NP pode ser reduzido em tempo polinomial. Um problema NP-Completo é tanto o NP quanto o NP-Hard. Uma grande questão matemática aberta é se P = NP, ou não. Se P ≠ NP (o que parece provável), nenhum algoritmo polinomial será encontrado para qualquer problema NP-Completo (exceto, talvez, em um computador quântico).

<sup>3</sup>  $\log_2$  é o logaritmo binário. É igual a  $\log_2(m) = \log(m) / \log(2)$ .

## Coeficiente de Gini ou Entropia?

A medida do *coeficiente de Gini* é aplicada por padrão, mas você pode selecionar a medida de impureza da *entropia* ao configurar o hiperparâmetro `criterion` para “`entropy`”. O conceito de entropia originou-se na termodinâmica como uma medida da desordem molecular: a entropia aproxima-se de zero quando as moléculas ainda estão paradas e bem ordenadas. Mais tarde, se espalhou em uma ampla variedade de campos do conhecimento, incluindo a *Teoria da Informação* de Shannon, na qual mede o conteúdo médio de informação de uma mensagem.<sup>4</sup> A entropia é zero quando todas as mensagens são idênticas. No Aprendizado de Máquina, ela é frequentemente utilizada como uma medida do coeficiente: a entropia de um conjunto é zero quando contém instâncias de apenas uma classe. A Equação 6-3 mostra a definição da entropia do *i*-ésimo nó. Por exemplo, o nó esquerdo de profundidade 2 na Figura 6-1 possui uma entropia igual a  $-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31$

*Equação 6-3. Entropia*

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log(p_{i,k})$$

Então, nesse caso, você deve utilizar o coeficiente de Gini ou a entropia? A verdade é que, na maioria das vezes, não faz muita diferença: elas levam a árvores semelhantes. O coeficiente de Gini é um pouco mais rápido para calcular, então é um bom padrão. No entanto, quando elas diferem, o coeficiente de Gini tende a isolar a classe mais frequente em seu próprio ramo da árvore, enquanto a entropia tende a produzir árvores ligeiramente mais equilibradas.<sup>5</sup>

## Hiperparâmetros de Regularização

As Árvores de Decisão fazem poucos pressupostos sobre os dados de treinamento (ao contrário dos modelos lineares, que obviamente assumem que os dados são lineares, por exemplo). Se for deixada sem restrições, a estrutura da árvore se adaptará aos dados de treinamento muito de perto, provavelmente se sobreajustando. Tal modelo geralmente é chamado de *modelo não paramétrico*, não porque ele não tenha quaisquer parâmetros (geralmente ele têm bastante), mas porque o número de parâmetros não é determinado antes do treinamento, então a estrutura do modelo é livre para ficar próxima aos dados.

<sup>4</sup> Uma redução da entropia geralmente é chamada de *ganho de informação*.

<sup>5</sup> Veja a análise interessante de Sebastian Raschka para obter mais detalhes (<http://goo.gl/UndTrO>).

Em contraste, um *modelo paramétrico*, como um modelo linear, tem um número pré-determinado de parâmetros, então seu grau de liberdade é limitado, reduzindo o risco de sobreajuste (mas aumentando o risco de subajuste).

Para evitar o sobreajuste nos dados de treinamento, você precisa restringir a liberdade da Árvore de Decisão durante este treinamento. Como você sabe, isso é chamado de regularização. Os hiperparâmetros de regularização dependem do algoritmo utilizado, mas geralmente você pode, pelo menos, restringir a profundidade máxima da Árvore de Decisão. No Scikit-Learn, isto é controlado pelo hiperparâmetro `max_depth` (o valor padrão é `None`, que significa ilimitado). Reduzir `max_depth` regularizará o modelo e reduzirá, assim, o risco de sobreajuste.

A classe `DecisionTreeClassifier` tem alguns outros parâmetros que restringem de forma semelhante a forma da Árvore de Decisão: `min_samples_split` (o número mínimo de amostras que um nó deve ter antes que possa ser dividido), `min_samples_leaf` (o número mínimo de amostras que um nó da folha deve ter), `min_weight_fraction_leaf` (o mesmo de `min_samples_leaf`, mas expressa como uma fração do número total de instâncias ponderadas), `max_leaf_nodes` (número máximo de nós da folha) e `max_features` (número máximo de características que são avaliadas para divisão em cada nó). Aumentar os hiperparâmetros `min_*` ou reduzir os hiperparâmetros `max_*` regularizará o modelo.



Outros algoritmos funcionam primeiro treinando a Árvore de Decisão sem restrições, depois *podando* (eliminando) nós desnecessários. Um nó cujos filhos são todos nós da folha é considerado desnecessário caso a melhoria da pureza que ele fornece não seja *estatisticamente significante*. Os testes estatísticos padrão, como o teste  $\chi^2$ , são utilizados para estimar a probabilidade de que a melhora seja puramente o resultado do acaso (que é chamada de *hipótese nula*). Se essa probabilidade, denominada *p-value*, for superior a um determinado limiar (geralmente 5%, controlado por um hiperparâmetro), então o nó é considerado desnecessário e seus filhos são excluídos. A poda continua até que todos os nós desnecessários tenham sido eliminados.

A Figura 6-3 mostra duas Árvores de Decisão treinadas no conjunto de dados de luas (introduzido no Capítulo 5). À esquerda, a Árvore de Decisão é treinada com os hiperparâmetros padrão (ou seja, sem restrições) e, à direita, a Árvore de Decisão é treinada com `min_samples_leaf = 4`. É bastante óbvio que o modelo à esquerda está sobreajustado e o modelo à direita provavelmente generalizará melhor.

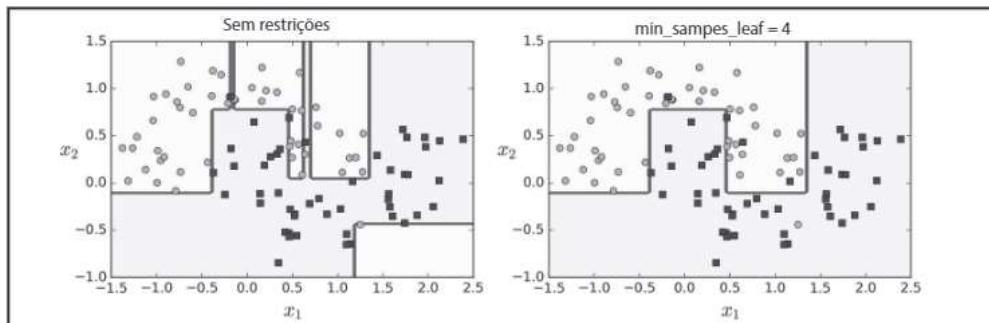


Figura 6-3. Regularização utilizando `min_samples_leaf`

## Regressão

As Árvores de Decisão também desempenham tarefas de regressão. Construiremos uma árvore de regressão utilizando a classe `DecisionTreeRegressor` do Scikit-Learn e treiná-la em um conjunto de dados quadrático confuso com `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

A árvore resultante está representada na Figura 6-4.

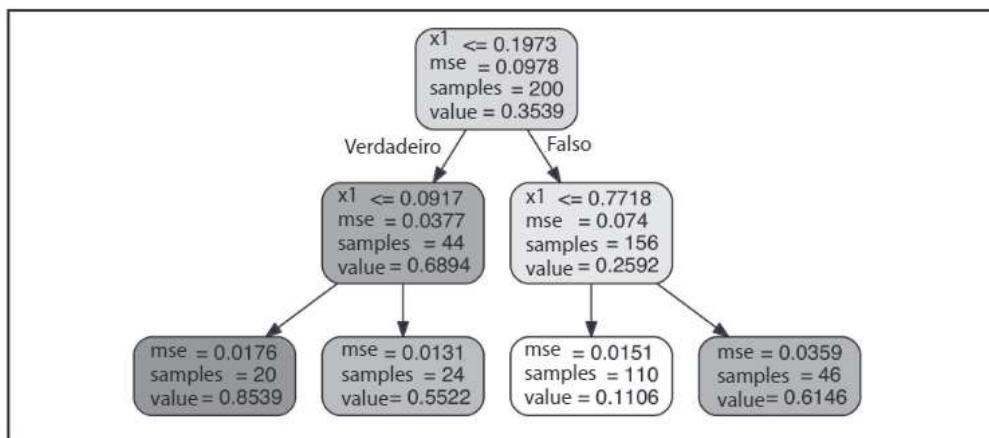


Figura 6-4. Uma Árvore de Decisão para regressão

Esta árvore se parece muito com a árvore de classificação que você criou anteriormente. A principal diferença é que, em vez de prever uma classe em cada nó, ela prevê um valor. Por exemplo, suponha que você queira fazer uma previsão para uma nova instância com

$x_1 = 0,6$ . Você atravessa a árvore começando na raiz e, eventualmente, alcança o nó da folha que prevê um `value=0,1106`. Esta previsão é simplesmente o valor-alvo médio das 110 instâncias de treinamento associadas a este nó da folha. Esta previsão resulta em um erro quadrático médio (MSE) igual a 0,0151 sobre essas 110 instâncias.

As previsões deste modelo estão representadas à esquerda da Figura 6-5. Se você definir `max_depth=3`, obterá as previsões representadas à direita. Observe como o valor previsto para cada região é sempre o valor-alvo médio das instâncias presentes nela. O algoritmo divide cada região de uma forma, o que faz com que a maior parte das instâncias de treinamento se aproxime o máximo possível desse valor previsto.

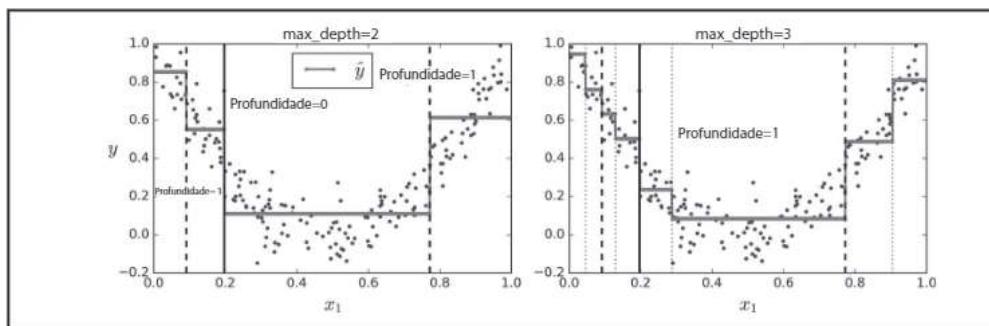


Figura 6-5. Previsões de dois modelos de regressão da Árvore de Decisão

O algoritmo CART funciona mais ou menos da mesma forma, exceto que, em vez de tentar dividir o conjunto de treinamento de forma a minimizar a impureza, ele agora tenta dividi-lo de forma a minimizar o MSE. A Equação 6-4 mostra a função de custo que o algoritmo tenta minimizar.

Equação 6-4. A função CART de custo para regressão

$$J(k, t_k) = \frac{m_{\text{esquerda}}}{m} \text{MSE}_{\text{esquerda}} + \frac{m_{\text{direita}}}{m} \text{MSE}_{\text{direita}} \text{ sendo que } \begin{cases} \text{MSE}_{\text{nó}} = \sum_{i \in \text{nó}} (\hat{y}_{\text{nó}} - y^{(i)})^2 \\ \hat{y}_{\text{nó}} = \frac{1}{m_{\text{nó}}} \sum_{i \in \text{nó}} y^{(i)} \end{cases}$$

Assim como para tarefas de classificação, as Árvores de Decisão são propensas a sobreajustes quando se trata de tarefas de regressão. Sem nenhuma regularização (ou seja, utilizando os hiperparâmetros padrão), você obtém as previsões à esquerda da Figura 6-6. É óbvio que o conjunto de treinamento está muito sobreajustado. Se configurarmos o `min_samples_leaf=10`, resultará em um modelo muito mais razoável, representado à direita da Figura 6-6.

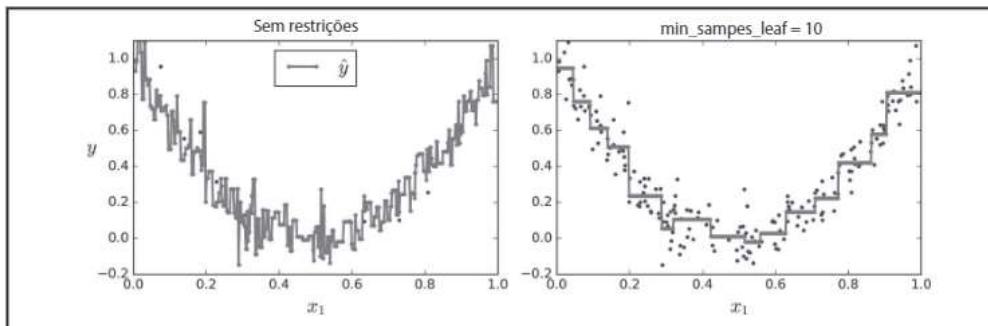


Figura 6-6. Regularizando um regressor da Árvore de Decisão

## Instabilidade

Espero que você esteja convencido de que as Árvores de Decisão têm muitas vantagens: são fáceis de entender, interpretar, usar, e são versáteis e poderosas. No entanto, elas têm algumas limitações. Primeiro, como você pode ter percebido, as Árvores de Decisão adoram fronteiras ortogonais de decisão (todas as divisões são perpendiculares a um eixo), o que as torna sensíveis à rotação do conjunto de treinamento. Por exemplo, a Figura 6-7 mostra um simples conjunto de dados linearmente separável: à esquerda, uma Árvore de Decisão pode dividi-lo facilmente, enquanto à direita, após o conjunto de dados girar em  $45^\circ$ , a fronteira de decisão parece desnecessariamente retorcida. Embora ambas as árvores de decisão se moldem perfeitamente ao conjunto de treinamento, é muito provável que o modelo à direita não generalize bem. Uma maneira de limitar esse problema é utilizar o PCA (consulte o Capítulo 8), o que, muitas vezes, resulta em uma melhor orientação dos dados de treinamento.

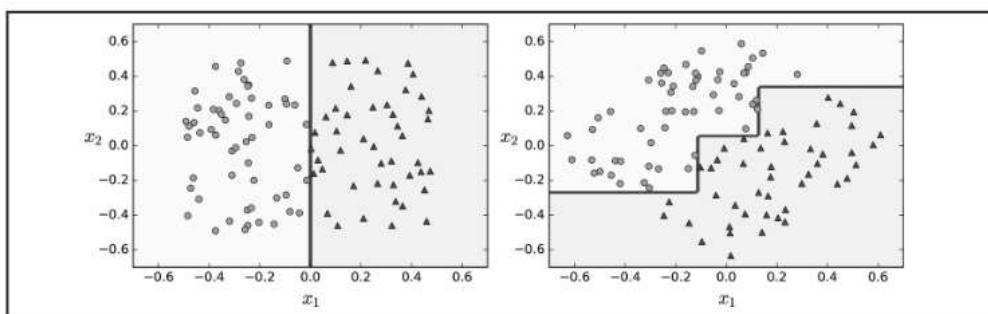


Figura 6-7. Sensibilidade à rotação no conjunto de treinamento

De um modo mais geral, o principal problema com as Árvores de Decisão é que elas são muito sensíveis a pequenas variações nos dados de treinamento. Por exemplo, se você remover a Iris-Versicolor mais larga do conjunto de treinamento da íris (aquela com

pétalas de 4,8cm de comprimento e 1,8cm de largura) e treinar uma nova Árvore de Decisão, pode ser que você obtenha o modelo representado na Figura 6-8. Como você pode ver, é muito diferente da Árvore de Decisão anterior (Figura 6-2). Na verdade, uma vez que o algoritmo de treinamento utilizado pelo Scikit-Learn é estocástico<sup>6</sup>, você pode obter modelos muito diferentes mesmo nos mesmos dados de treinamento (a menos que você defina o hiperparâmetro `random_state`).

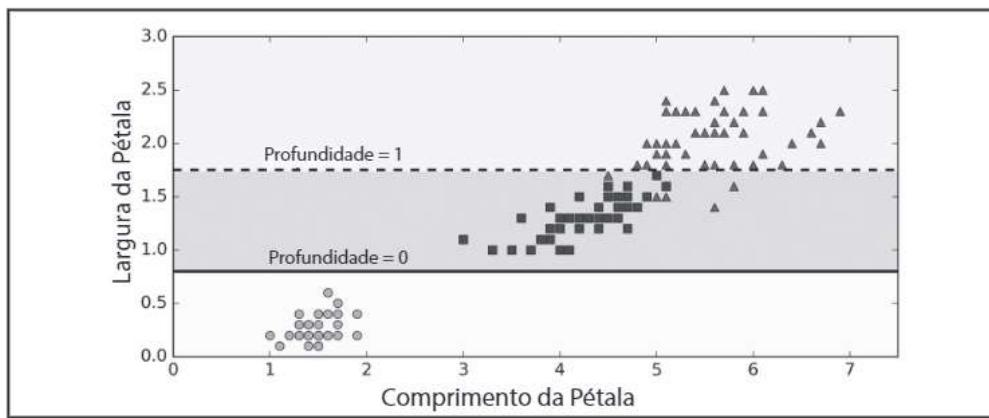


Figura 6-8. Sensibilidade aos detalhes no conjunto de treinamento

As Florestas Aleatórias podem limitar essa instabilidade provendo previsões sobre muitas árvores, como veremos no próximo capítulo.

## Exercícios

1. Qual é a profundidade aproximada de uma Árvore de Decisão treinada (sem restrições) em um conjunto com 1 milhão de instâncias?
2. O coeficiente Gini de um nó geralmente é menor ou maior do que o dos seus pais? Ele é *geralmente* menor/maior, ou *sempre* menor/maior?
3. É uma boa ideia tentar diminuir seu `max_depth` se uma Árvore de Decisão estiver se sobreajustando ao conjunto de treinamento?
4. É uma boa ideia tentar dimensionar as características de entrada se uma Árvore de Decisão estiver se subajustando ao conjunto de treinamento?

---

<sup>6</sup> Seleciona aleatoriamente o conjunto de características a ser avaliado em cada nó.

5. Se treinar uma Árvore de Decisão em um conjunto de treinamento contendo 1 milhão de instâncias demora 1 hora, aproximadamente quanto tempo demorará para treinar outra Árvore de Decisão em um conjunto de treinamento contendo 10 milhões de instâncias?
6. Se o seu conjunto de treinamento contém 100 mil instâncias, a configuração `presort=True` acelerará o treinamento?
7. Treine e ajuste uma Árvore de Decisão para o conjunto de dados de luas.
  - a. Gere um conjunto de dados de luas utilizando `make_moons(n_samples=10000, noise=0.4)`.
  - b. Com a utilização do `train_test_split()`, divida em um conjunto de treinamento e um conjunto de testes.
  - c. Utilize a pesquisa de grade com validação cruzada (com a ajuda da classe `GridSearchCV`) para encontrar bons valores de hiperparâmetros para um `DecisionTreeClassifier`. Dica: tente vários valores para `max_leaf_nodes`.
  - d. Treine-o no conjunto completo de treinamento utilizando estes hiperparâmetros e meça o desempenho do seu modelo no conjunto de teste. Você deve obter aproximadamente 85% a 87% de acurácia.
8. Cultive uma floresta.
  - a. Continuando o exercício anterior, gere mil subconjuntos do conjunto de treinamento, cada um contendo 100 instâncias selecionadas aleatoriamente. Dica: você pode utilizar a classe `ShuffleSplit` do Scikit-Learn para isso.
  - b. Treine uma Árvore de Decisão em cada subconjunto utilizando os melhores valores do hiperparâmetro encontrados acima. Avalie essas mil Árvores de Decisão no conjunto de testes. Uma vez treinadas em conjuntos menores, essas Árvores de Decisão provavelmente terão um desempenho pior do que a primeira, alcançando apenas 80% de acurácia.
  - c. Agora vem a mágica. Gere as previsões das mil Árvores de Decisão e mantenha apenas a previsão mais frequente para cada instância do conjunto de testes (você pode utilizar a função `mode()` do SciPy para isso). Isso lhe dá *previsões dos votos majoritários* sobre o conjunto de testes.
  - d. Avalie estas previsões no conjunto de teste: você deve obter uma acurácia ligeiramente maior que o seu primeiro modelo (cerca de 0,5 a 1,5% a mais). Parabéns, você treinou um classificador de Floresta Aleatória!

As soluções para esses exercícios estão disponíveis no Apêndice A.



## Capítulo 7

# Ensemble Learning e Florestas Aleatórias

Vamos supor que você faça uma pergunta complexa a milhares de pessoas aleatórias e, então, reúna suas respostas. Em muitos casos, você verá que esta resposta agregada é melhor do que a resposta de um especialista, o que é chamado de *sabedoria das multidões*. Da mesma forma, se você agregar as previsões de um conjunto de previsores (como classificadores ou regressores), muitas vezes obterá melhores previsões do que com o melhor previsor individual. Um conjunto de previsores é chamado de *ensemble*; assim, esta técnica é chamada *Ensemble Learning*, e um algoritmo de Ensemble Learning é chamado de *Ensemble method*.

Por exemplo, você pode treinar um conjunto de classificadores de Árvores de Decisão, cada um em um subconjunto aleatório diferente do conjunto de treinamento. Para fazer previsões, devemos obtê-las de todas as árvores individuais e, então, prever a classe que obtém a maioria dos votos (veja o último exercício no Capítulo 6). Esse ensemble de Árvores de Decisão é chamado de Floresta Aleatória e, apesar da sua simplicidade, é um dos mais poderosos algoritmos de Aprendizado de Máquina disponível atualmente.

Além disso, como discutimos no Capítulo 2, você frequentemente utilizará os Ensemble methods perto da conclusão de um projeto, uma vez que já tenha construído alguns bons previsores, com o objetivo de combiná-los em um ainda melhor. De fato, as soluções vencedoras nas competições de Aprendizado de Máquina muitas vezes envolvem vários Ensemble methods (com maior notoriedade nas competições do Prêmio Netflix; <http://netflixprize.com/>).

Neste capítulo discutiremos os Ensemble methods mais populares, incluindo *bagging*, *boosting*, *stacking* e outros. Também exploraremos as Florestas Aleatórias.

## Classificadores de Votação

Suponha que você tenha treinado alguns classificadores, cada um alcançando uma precisão de aproximadamente 80%. Você pode ter um classificador de Regressão Logística, um de SVM, um de Floresta Aleatória, um *K-Nearest Neighbors* e, talvez, mais alguns (veja a Figura 7-1).

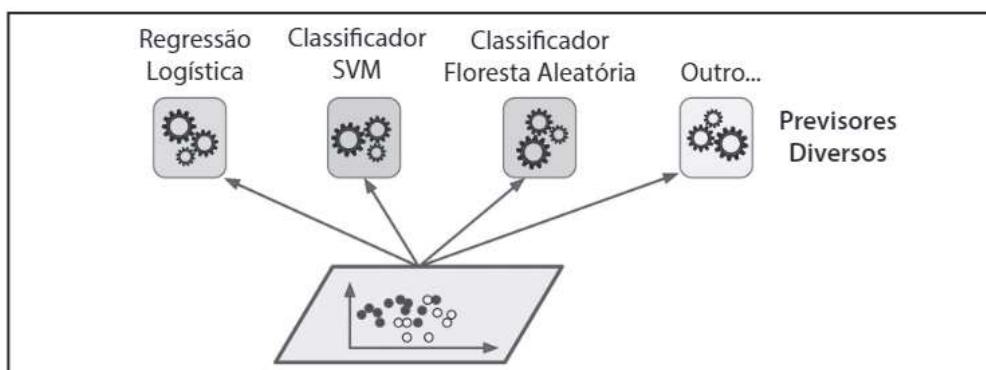


Figura 7-1. Treinando diversos classificadores

Uma maneira bem simples de criar um classificador ainda melhor seria reunir as previsões de cada classificador e prever a classe que obtém a maioria dos votos. Este classificador de votos majoritários é chamado de classificador *hard voting* (veja a Figura 7-2).

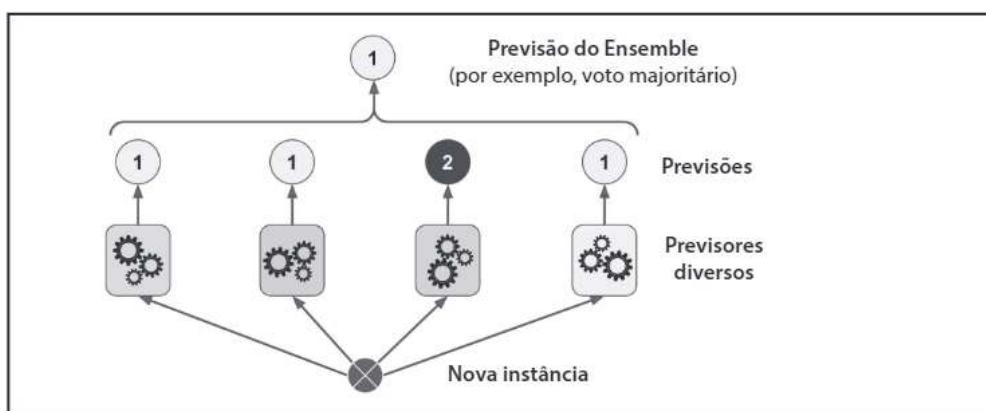


Figura 7-2. Previsões do classificador hard voting

Um tanto surpreendente, este classificador de votação geralmente consegue uma maior precisão do que o melhor classificador no *ensemble*. Na verdade, mesmo que cada classificador seja um *aprendiz fraco* (o que significa que sua classificação é apenas um pouco

melhor do que adivinhações aleatórias), o conjunto ainda pode ser um *forte aprendiz* (alcançando alta acurácia) desde que haja um número suficiente de aprendizes fracos e que sejam suficientemente diferentes.

Como isto é possível? A analogia a seguir pode ajudar a esclarecer esse mistério. Suponha que você tenha uma moeda ligeiramente tendenciosa que tem 51% de chance de dar cara e 49% de chance dar coroa. Se você jogar mil vezes, obterá mais ou menos 510 caras e 490 coroas, portanto, uma maioria de caras. Se você fizer o cálculo, verá que a probabilidade de obter a maioria de caras após mil jogadas é próxima de 75%. Quanto mais você jogar a moeda, maior a probabilidade (por exemplo, com 10 mil lançamentos, a probabilidade sobe mais de 97%). Isto deve-se à *lei dos grandes números*: ao continuar jogando a moeda, a proporção das caras fica cada vez mais próxima da sua probabilidade (51%). A Figura 7-3 mostra 10 séries de lançamentos induzidos de moedas. À medida que o número de lançamentos aumenta, a proporção das caras se aproxima de 51%. Eventualmente, todas as 10 séries terminam tão perto de 51% que sempre estarão consistentemente acima de 50%.

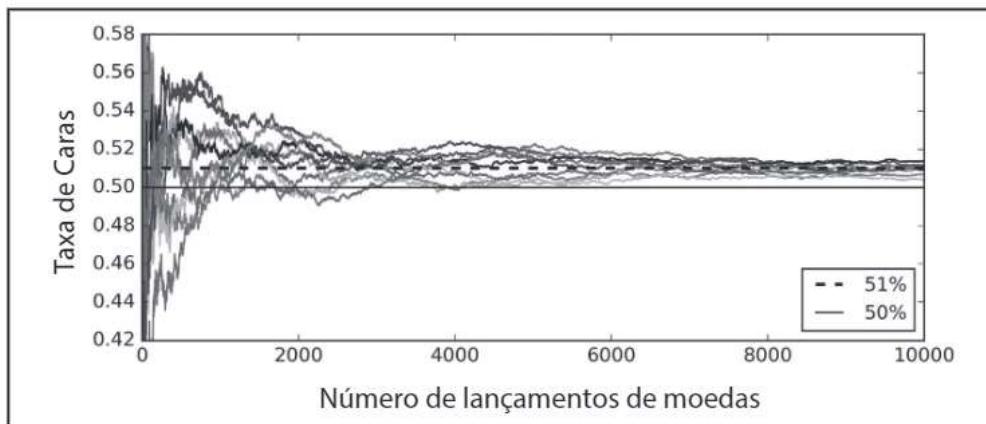


Figura 7-3. A *lei dos grandes números*

Da mesma forma, suponha que você crie um conjunto contendo mil classificadores que estão individualmente corretos em apenas 51% do tempo (um pouco melhor do que a previsão aleatória). Se você prever a classe votada majoritariamente, é possível obter até 75% de acurácia! No entanto, isso só se dá se todos os classificadores forem perfeitamente independentes, gerando erros não correlacionados, o que claramente não é o caso, pois eles são treinados nos mesmos dados. É provável que produzam os mesmos tipos de erros, então haverá muitos votos majoritários para a classe errada, reduzindo a acurácia do ensemble.



Os métodos *Ensemble* funcionam melhor quando os previsores são o mais independente dos outros quanto possível. Uma maneira de obter classificadores diversos seria treiná-los utilizando algoritmos bem diferentes. Isso aumenta a chance de cometeterem tipos de erros muito diferentes, melhorando a acurácia do ensemble.

O código a seguir cria e treina um classificador de votação no Scikit-Learn composto por três classificadores diversos (o conjunto de treinamento é o conjunto de dados de luas introduzido no Capítulo 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Vejamos a acurácia de cada classificador no conjunto de testes:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

Aí está! O classificador de votação supera ligeiramente todos os classificadores individuais.

Se todos os classificadores são capazes de estimar as probabilidades da classe (ou seja, possuem um método `predict_proba()`), então você pode pedir ao Scikit-Learn para prever a classe com a maior probabilidade na média sobre todos os classificadores individuais. Isto é chamado *soft voting*. Muitas vezes, ele consegue um desempenho maior do que o *hard voting*, pois dá mais peso aos votos altamente confiantes. Você só precisa substituir `voting="hard"` por `voting="soft"` e assegurar que todos os classificadores possam estimar as probabilidades da classe. Por padrão, este não é o caso da classe `SVC`, então você precisa configurar seu hiperparâmetro `probability` para `True` (isso fará com que a classe `SVC` utilize a validação cruzada para estimar as probabilidades da

classe, diminuindo o treinamento e adicionando um método `predict_proba()`). Se você modificar o código anterior para utilizar o soft voting, descobrirá que o classificador de votação atingirá mais de 91% de acurácia!

## Bagging e Pasting

Como discutimos, uma forma de obter um conjunto diversificado de classificadores é utilizar algoritmos de treinamento muito diferentes. Outra abordagem seria utilizar o mesmo algoritmo de treinamento para cada previsor, mas treiná-los em diferentes subconjuntos aleatórios do conjunto de treinamento. Quando a amostragem é realizada *com substituição*, este método é chamado *bagging* (<http://goo.gl/o42tml>)<sup>1</sup> (abreviação para *bootstrap aggregating*)<sup>2</sup>. Quando a amostragem é realizada *sem substituição*, é chamado *pasting* (<http://goo.gl/BXm0pm>).<sup>3</sup>

Em outras palavras, tanto bagging quanto pasting permitem que as instâncias de treinamento sejam amostradas várias vezes por meio de múltiplos previsores, mas somente bagging permite que as instâncias de treinamento sejam amostradas diversas vezes pelo mesmo previsor. Este processo de amostragem e treinamento está representado na Figura 7-4.

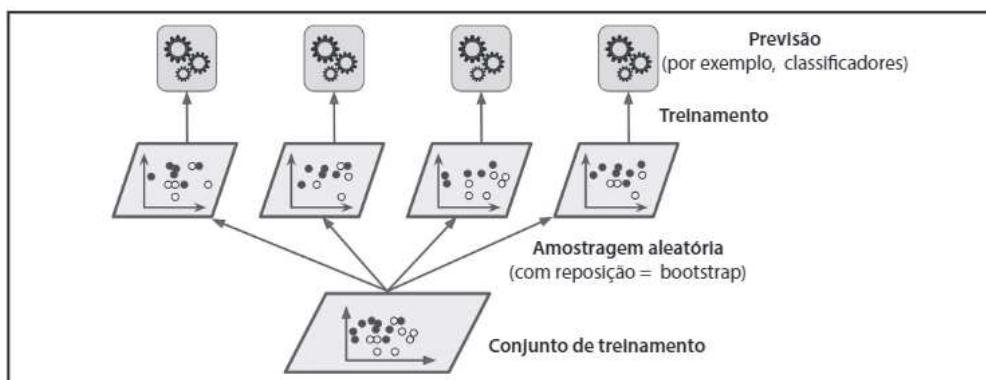


Figura 7-4. Conjunto de treinamento e amostragem pasting/bagging

Uma vez que todos os previsores são treinados, o ensemble pode fazer previsão para uma nova instância simplesmente agregando as previsões de todos os previsores. A função de agregação é tipicamente o *modo estatística* (ou seja, a previsão mais frequente, assim

<sup>1</sup> “Bagging Predictors”, L. Breiman (1996).

<sup>2</sup> Reamostrar com substituição é chamado *bootstrapping* em estatística.

<sup>3</sup> “Pasting small votes for classification in large databases and online”, L. Breiman (1999).

como um classificador hard voting) para a classificação ou a média para a regressão. Cada previsor individual tem um viés mais alto do que se fosse treinado no conjunto original, mas a agregação reduz tanto o viés quanto a variância.<sup>4</sup> Geralmente, o resultado em rede é que o conjunto tem um viés semelhante, mas uma variância inferior do que um previsor único treinado no conjunto de treinamento original.

Como você pode ver na Figura 7-4, os previsores podem ser treinados em paralelo com diferentes núcleos de CPU ou mesmo servidores diferentes. Da mesma forma, as previsões podem ser feitas em paralelo. Esta é uma das razões pelas quais bagging e pasting são métodos tão populares: eles escalonam muito bem.

## Bagging e Pasting no Scikit-Learn

O Scikit-Learn oferece uma API simples tanto para bagging quanto para pasting com a classe `BaggingClassifier` (ou `BaggingRegressor` para a regressão). O código a seguir treina um conjunto de 500 classificadores de Árvores de Decisão,<sup>5</sup> cada um treinado em 100 instâncias de treinamento amostradas aleatoriamente com substituição no conjunto de treinamento (este é um exemplo de bagging, mas, se você quiser utilizar pasting, configure `bootstrap=False`). O parâmetro `n_jobs` informa ao Scikit-Learn o número de núcleos de CPU a serem utilizados para treinamento e previsões (-1 informa ao Scikit-Learn para utilizar todos os núcleos disponíveis):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```



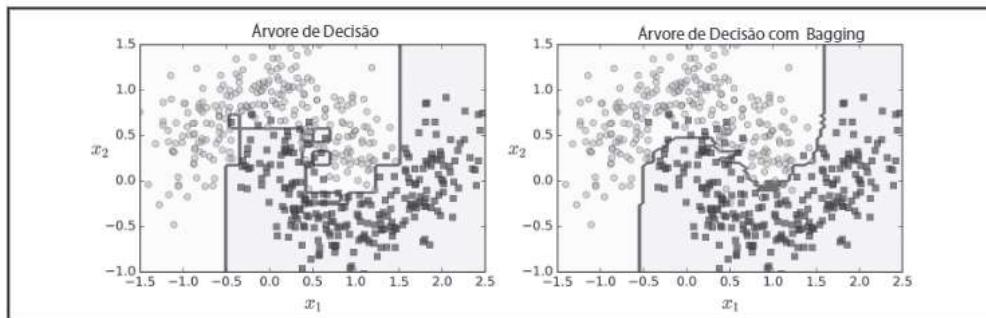
O `BaggingClassifier` executa automaticamente um soft voting em vez de um hard voting se o classificador de base puder estimar as probabilidades da classe (ou seja, se tiver um método `predict_proba()`), como nos classificadores das Árvores de Decisão.

A Figura 7-5 compara a fronteira de decisão de uma única Árvore de Decisão com a fronteira de decisão de um bagging ensemble de 500 árvores (do código anterior), ambos treinados no conjunto de dados em formato de luas. Como você pode ver, as previsões

<sup>4</sup> Viés e variância introduzidos no Capítulo 4.

<sup>5</sup> `max_samples` pode, alternativamente, ser configurado para flutuar entre 0,0 e 1,0, no caso em que o número máximo de instâncias a serem amostradas é igual ao tamanho do conjunto de treinamento vezes `max_samples`.

do ensemble provavelmente generalizarão bem melhor do que as previsões da Árvore de Decisão: o ensemble tem um viés comparável, mas uma variância menor (faz com que o mesmo número de erros ocorra no conjunto de treinamento, mas a fronteira de decisão é menos irregular).



*Figura 7-5. Uma Árvore de Decisão única versus um bagging ensemble de 500 árvores*

O bootstrapping introduz um pouco mais de diversidade nos subconjuntos em que cada previsor é treinado, então bagging acaba com um viés ligeiramente mais alto do que pasting, mas isto também significa que os previsores acabam sendo menos correlatos, portanto a variância do ensemble é reduzida. Em geral, bagging resulta em melhores modelos, o que explica por que em geral é escolhido. Entretanto, se você tiver tempo livre e poder de CPU, utilize a validação cruzada para avaliar tanto o bagging quanto o pasting e selecione o que funcionar melhor.

## Avaliação Out-of-Bag

Com o bagging, algumas instâncias podem ser amostradas muitas vezes em qualquer previsor, enquanto outras podem simplesmente não serem amostradas. Por padrão, um `BaggingClassifier` amostra  $m$  instâncias de treinamento com substituição (`bootstrap=True`), sendo  $m$  o tamanho do conjunto. Isto significa que somente 63% das instâncias de treinamento são amostradas na média para cada previsor.<sup>6</sup> Os 37% restantes das instâncias que não são amostradas são chamadas de instâncias *out-of-bag* (oob). Observe que elas não são os mesmos 37% para todos os previsores.

Como um previsor nunca vê as instâncias oob durante o treinamento, ele pode ser avaliado nessas instâncias sem a necessidade de um conjunto de validação separado ou de validação cruzada. Você pode avaliar o ensemble em si com a média das avaliações oob de cada previsor.

<sup>6</sup> À medida que  $m$  cresce, esta taxa se aproxima de  $1 - \exp(-1) \approx 63,212\%$ .

Você pode configurar `oob_score=True` no Scikit-Learn ao criar um `BaggingClassifier` para requerer uma avaliação `oob` automática após o treinamento. O código a seguir demonstra isto. A pontuação da avaliação resultante está disponível na variável `oob_score`:

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.9013333333333332
```

De acordo com esta avaliação `oob`, este `BaggingClassifier` parece obter uma acurácia de aproximadamente 90,1% no conjunto de teste. Verificaremos:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.9120000000000003
```

Conseguimos 91,2% de acurácia no conjunto de teste — quase lá!

A função de decisão `oob` para cada instância de treinamento também está disponível através da variável `oob_decision_function_`. Neste caso (como o estimador base tem um método `predict_proba()`), a função de decisão retorna as probabilidades da classe para cada instância de treinamento. Por exemplo, a avaliação `oob` estima que a primeira instância de treinamento tem uma probabilidade de 68,25% de pertencer à classe positiva (e 31,75% de pertencer à classe negativa):

```
>>> bag_clf.oob_decision_function_
array([[ 0.31746032,  0.68253968],
       [ 0.34117647,  0.65882353],
       [ 1.        ,  0.        ],
       ...
       [ 1.        ,  0.        ],
       [ 0.03108808,  0.96891192],
       [ 0.57291667,  0.42708333]])
```

## Patches Aleatórios e Subespaços Aleatórios

A classe `BaggingClassifier` também permite a amostragem das características, que é controlada pelos dois hiperparâmetros: `max_features` e `bootstrap_features`. Eles trabalham da mesma forma que `max_samples` e `bootstrap`, mas para a amostragem da característica, não para a da instância. Assim, cada previsor será treinado em um subconjunto aleatório das características da entrada.

Isso é particularmente útil quando você lida com entradas de alta dimensionalidade (como imagens). A amostragem das instâncias de treinamento e das características é chamada

de método *Random Patches* (<http://goo.gl/B2EcM2>).<sup>7</sup> Quando mantemos todas as instâncias de treinamento (por exemplo, `bootstrap = False` e `max_samples = 1.0`), exceto as características de amostragem (ou seja, `bootstrap_features=True` e/ou `max_features` menores que 1,0), são chamadas de método *Random Subspaces* (<http://goo.gl/NPi5vH>).<sup>8</sup>

As características da amostragem resultam em uma maior diversidade dos previsores, negociando um pouco mais de viés por uma menor variância.

## Florestas Aleatórias

Como discutimos, uma Floresta Aleatória (<http://goo.gl/zVOGQI>)<sup>9</sup> é um ensemble de Árvores de Decisão, geralmente treinado pelo método bagging (ou algumas vezes pasting) com `max_samples` ajustada ao tamanho do conjunto de treinamento. Em vez de construir um `BaggingClassifier` e passá-lo por um `DecisionTreeClassifier`, você pode utilizar a classe `RandomForestClassifier`, que é mais conveniente e otimizada para Árvores de Decisão.<sup>10</sup> (igualmente, existe uma classe `RandomForestRegressor` para tarefas de regressão). O código a seguir treina um classificador de Floresta Aleatória com 500 árvores (cada uma limitada a 16 nós no máximo) com a utilização de todos os núcleos de CPU disponíveis:

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

Com poucas exceções, um `RandomForestClassifier` tem todos os hiperparâmetros de um `DecisionTreeClassifier` (para controlar como as árvores se desenvolvem) mais todos os hiperparâmetros de um `BaggingClassifier` para controlar o próprio ensemble.<sup>11</sup>

O algoritmo Floresta Aleatória introduz uma aleatoriedade extra ao desenvolver árvores; em vez de buscar a melhor característica ao dividir um nó (veja o Capítulo 6), ele a busca entre um subconjunto aleatório dessas características, resultando em uma grande diversidade da árvore, que (mais uma vez) troca um alto viés por uma baixa variância,

<sup>7</sup> “Ensembles on Random Patches”, G. Louppe and P. Geurts (2012).

<sup>8</sup> “The random subspace method for constructing decision forests”, Tin Kam Ho (1998).

<sup>9</sup> “Random Decision Forests”, T. Ho (1995).

<sup>10</sup> A classe `BaggingClassifier` permanece útil se você quiser um *bag* de algo em vez de Árvores de Decisão.

<sup>11</sup> Existem algumas exceções notáveis: `splitter` está ausente (forçado para “random”), `presort` está ausente (forçado para `False`), `max_samples` está ausente (forçado para 1.0), e `base_estimator` está ausente (forçado para `DecisionTreeClassifier` com os hiperparâmetros fornecidos).

geralmente produzindo um melhor modelo no geral. O `BaggingClassifier` a seguir é equivalente ao `RandomForestClassifier` anterior:

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),  
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

## Árvores-Extras

Ao desenvolver uma árvore em uma Floresta Aleatória, consideramos somente um subconjunto aleatório para a divisão das características em cada nó (como discutido anteriormente). É possível tornar as árvores ainda mais aleatórias ao utilizarmos também os limiares aleatórios para cada característica em vez de buscar pelo melhor limiar possível (como fazem as Árvores de Decisão regulares).

Uma floresta de árvores aleatórias tão extremas é chamada simplesmente de ensemble de *Árvores Extremamente Aleatorizadas* (<http://goo.gl/RHGEA4>)<sup>12</sup> (ou *Árvores-Extras* na abreviação). Mais uma vez, isto troca mais viés por uma variância menor. Também torna o treinamento das Árvores-Extras mais rápido do que as Florestas Aleatórias regulares, já que encontrar o melhor limiar possível para cada característica em cada nó é uma das tarefas que mais demandam tempo no desenvolvimento de uma árvore.

Você pode criar um classificador Árvores-Extras usando a classe `ExtraTreesClassifier` do Scikit-Learn. Sua API é idêntica à classe `RandomForestClassifier`. Igualmente, a classe `ExtraTreesRegressor` tem a mesma API da classe `RandomForestRegressor`.



A priori, é difícil dizer se um `RandomForestClassifier` terá um desempenho melhor ou pior que um `ExtraTreesClassifier`. Geralmente, a única maneira de saber é tentar com ambos e compará-los utilizando a validação cruzada (e ajustando os hiperparâmetros utilizando a grid search).

## Importância da Característica

Outra grande qualidade das Florestas Aleatórias é que elas facilitam a medição da importância relativa de cada característica. O Scikit-Learn mede a importância de uma característica analisando o quanto os nós da árvore que a utilizam reduzem, na média, a impureza (através de todas as árvores na floresta). Mais precisamente, é uma média ponderada em que o peso de cada nó é igual ao número de amostras treinadas que são associadas a ela (veja o Capítulo 6).

---

<sup>12</sup> “Extremely randomized trees”, P. Geurts, D. Ernst, L. Wehenkel (2005).

O Scikit-Learn calcula automaticamente esta pontuação para cada característica após o treinamento, então dimensiona os resultados para que a soma de todas as importâncias seja igual a 1. Você pode acessar o resultado utilizando a variável `feature_importances_`. Por exemplo, o código a seguir treina um `RandomForestClassifier` no conjunto de dados da íris (introduzido no Capítulo 4) e exibe cada importância desta característica. Parece que as características mais importantes são o comprimento da pétala (44%) e a largura (42%), enquanto, comparativamente, o comprimento e a largura da sépala são menos importantes (11% e 2%, respectivamente).

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Da mesma forma, se você treinar um classificador Floresta Aleatória no conjunto de dados MNIST (introduzido no Capítulo 3) e plotar cada importância do pixel terá a imagem representada na Figura 7-6.

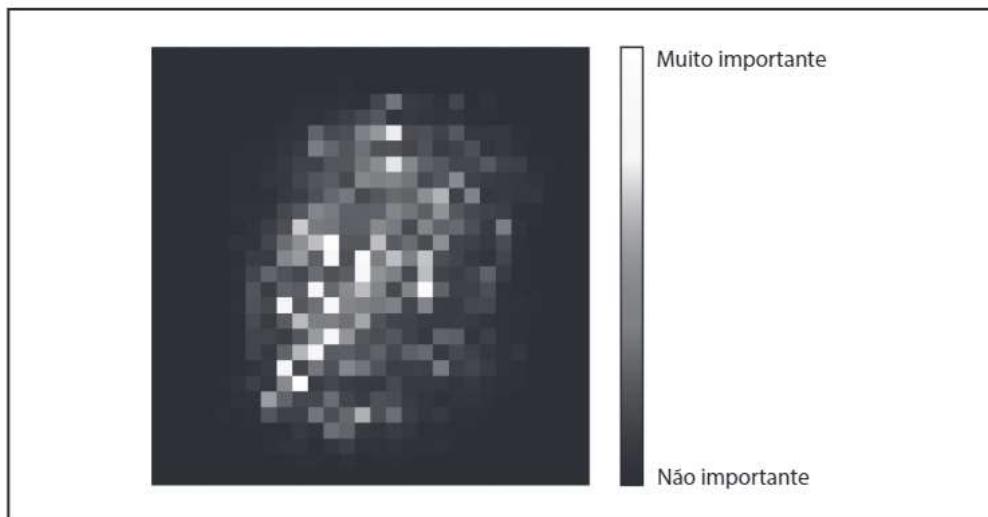


Figura 7-6. Importância do pixel MNIST (de acordo com um classificador Floresta Aleatória)

As Florestas Aleatórias são muito úteis para entender rapidamente quais características realmente importam, principalmente se você precisar executar sua seleção.

## Boosting

*Boosting* (originalmente chamado de *hypothesis boosting*) se refere a qualquer método Ensemble que combina vários aprendizes fracos em um forte. A ideia geral da maioria dos métodos boosting é treinar sequencialmente os previsores, cada um tentando corrigir seu antecessor. Existem muitos métodos boosting disponíveis, mas os mais populares são *AdaBoost* (<http://goo.gl/OIduRW>)<sup>13</sup> (abreviação de *Adaptive Boosting*) e *Gradient Boosting*. Começaremos com o *AdaBoost*.

### AdaBoost

Uma forma de o novo previsor corrigir seu antecessor é prestar um pouco mais de atenção às instâncias de treinamento que seu antecessor subajustou. Isto resulta em novos previsores focando mais e mais *cases difíceis*. Esta técnica é utilizada pelo *AdaBoost*.

Por exemplo, para construir um classificador AdaBoost, treinamos um classificador de primeira base (como uma Árvore de Decisão) e o utilizamos para fazer previsões no conjunto de treinamento. O peso relativo das instâncias de treinamento classificadas erroneamente é aumentado. Um segundo classificador é treinado com a utilização dos pesos atualizados e novamente ele faz previsões no conjunto de treinamento, os pesos são atualizados e assim por diante (veja Figura 7-7).

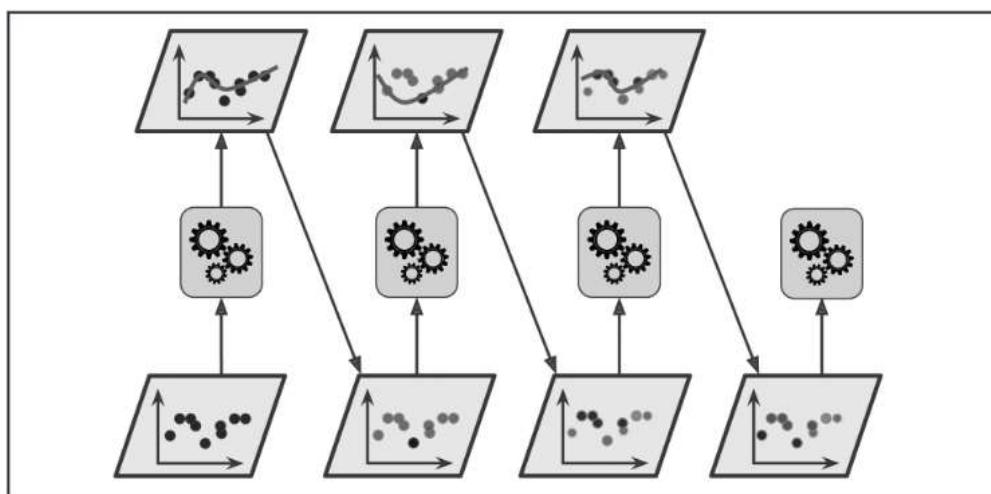


Figura 7-7. Treinamento sequencial AdaBoost com atualizações de peso das instâncias

<sup>13</sup> “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, Yoav Freund, Robert E. Schapire (1997).

A Figura 7-8 mostra as fronteiras de decisão de cinco previsores consecutivos no conjunto de dados em formato de luas (neste exemplo, cada previsor é um classificador SVM altamente regularizado com um kernel RBF)<sup>14</sup>. O primeiro classificador obtém muitas instâncias erradas, o que faz seus pesos serem aumentados. O segundo classificador, entretanto, faz um trabalho melhor nessas instâncias, e assim por diante. A plotagem à direita representa a mesma sequência de previsores, exceto que a taxa de aprendizado é reduzida pela metade (ou seja, os pesos das instâncias classificadas erroneamente são aumentados pela metade a cada iteração). Como você pode ver, esta técnica de aprendizado sequencial tem algumas semelhanças com o Gradiente Descendente, com a exceção de que, em vez de ajustar os parâmetros de um único previsor para minimizar uma função de custo, o AdaBoost adiciona previsores ao ensemble, tornando-o gradualmente melhor.

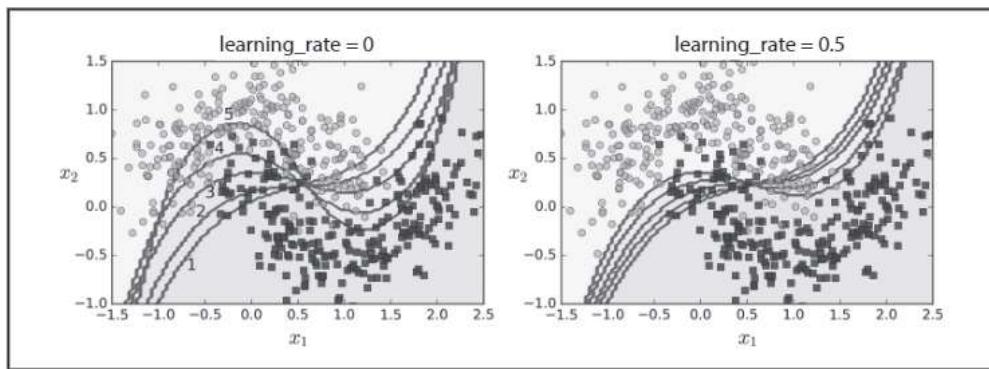


Figura 7-8. Fronteiras de decisão de previsores consecutivos

Quando todos os previsores estão treinados, o ensemble faz previsões muito parecidas com o bagging ou pasting, mas os previsores têm pesos diferentes dependendo da sua precisão geral no conjunto de treinamento ponderado.



Existe uma desvantagem importante para esta técnica de aprendizado sequencial: ela não pode ser paralelizada (ou apenas parcialmente) uma vez que cada previsor só pode ser treinado após o previsor anterior ter sido treinado e avaliado. Como resultado, ele não escalona tão bem quanto o bagging ou o pasting.

Vamos dar uma olhada no algoritmo AdaBoost. Cada peso  $w^{(i)}$  da instância é inicialmente definido em  $\frac{1}{m}$ . Um primeiro previsor é treinado e sua taxa de erro ponderado  $r_1$  é calculada no conjunto de treinamento; veja a Equação 7-1.

---

<sup>14</sup> Apenas para fins ilustrativos. Os SVM geralmente não são bons previsores de base para o AdaBoost, porque são lentos e tendem a ser instáveis com o AdaBoost.

*Equação 7-1. Taxa de erro ponderada do j-ésimo previsor*

$$r_j = \frac{\sum_{i=1}^m w^{(i)}}{\sum_{i=1}^m w^{(i)} \text{ se } \hat{y}_j^{(i)} \neq y^{(i)}} \text{ sendo que } \hat{y}_j^{(i)} \text{ é a } j\text{-ésima previsão do previsor para a } i\text{-ésima instância.}$$

Utilizando a Equação 7-2, o peso  $\alpha_j$  do previsor é calculado, sendo que  $\eta$  é o hiperparâmetro da taxa de aprendizado (o padrão é 1).<sup>15</sup> Quanto mais preciso for o previsor, mais alto será seu peso. Se ele apenas estimar aleatoriamente, então seu peso será próximo de zero. No entanto, se ele errar com mais frequência (ou seja, com menos acurácia do que a estimativa aleatória), então seu peso será negativo.

*Equação 7-2. Peso do previsor*

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Os pesos das instâncias são atualizados usando a Equação 7-3: as instâncias classificadas erroneamente são aumentadas.

*Equação 7-3. Regra da atualização do peso*

para  $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{se } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{se } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Então todos os pesos das instâncias são normalizados (ou seja, divididos por  $\sum_{i=1}^m w^{(i)}$ ).

Finalmente, um novo previsor é treinado usando os pesos atualizados e todo o processo é repetido (o novo peso do previsor é calculado, os pesos das instâncias são atualizados então outro previsor é treinado, e assim por diante). O algoritmo para quando o número de previsões desejado é alcançado, ou quando é encontrado um previsor perfeito.

Para fazer previsões, o AdaBoost simplesmente calcula as previsões de todos os previsores e os pondera usando os pesos  $\alpha_j$  dos previsores. A classe do previsor é aquela que recebe a maioria dos votos ponderados (veja a Equação 7-4).

<sup>15</sup> O algoritmo original AdaBoost não utiliza um hiperparâmetro da taxa de aprendizado.

### Equação 7-4. Previsões AdaBoost

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j \text{ sendo que } N \text{ é o número de previsores.}$$

$\hat{y}_i(\mathbf{x}) = k$

Na verdade, o Scikit-Learn utiliza uma versão multiclasse do AdaBoost chamada *SAMME* (<http://goo.gl/Eji2vR>)<sup>16</sup> (que significa *Stagewise Additive Modeling using a Multi-class Exponential loss function*). O SAMME é equivalente ao AdaBoost quando existem apenas duas classes. Além do mais, se os previsores puderem estimar probabilidades da classe (ou seja, se eles tiverem um método `predict_proba()`), o Scikit-Learn pode utilizar uma variante do SAMME chamada *SAMME.R* (o *R* significa “Real”) que se baseia nas probabilidades da classe em vez de previsões, e geralmente tem um melhor desempenho.

O código a seguir treina um classificador AdaBoost baseado em 200 *Decision Stumps* usando a classe `AdaBoostClassifier` do Scikit-Learn (como esperado, existe também uma classe `AdaBoostRegressor`). Uma *Decision Stump* é uma Árvore de Decisão com `max_depth=1` — em outras palavras, uma árvore composta de um só nó de decisão mais dois nós de folhas. Esta é a estimativa base padrão para a classe `AdaBoostClassifier`:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```



Se o seu ensemble AdaBoost estiver se sobreajustando ao conjunto de treinamento, você pode tentar reduzir o número de estimadores ou regularizar mais fortemente o estimador base.

## Gradient Boosting

Outro algoritmo Boosting bem popular é o *Gradient Boosting* (<http://goo.gl/Ezw4jL>).<sup>17</sup> Assim como o AdaBoost, o Gradient Boosting adiciona previsores sequencialmente a um conjunto, cada um corrigindo seu antecessor. No entanto, em vez de ajustar os pesos da instância a cada iteração como o AdaBoost faz, este método tenta ajustar o novo previsor aos *erros residuais* feitos pelo previsor anterior.

---

<sup>16</sup> Para mais detalhes, veja “Multi-Class AdaBoost”, J. Zhu *et al.* (2006).

<sup>17</sup> Introduzido primeiro em “Arcing the Edge”, L. Breiman (1997).

Analisaremos um exemplo de regressão simples utilizando as Árvores de Decisão como previsores base (claro que o Gradient Boosting também trabalha muito bem com as tarefas de regressão). Isso é chamado *Gradient Tree Boosting*, ou *Gradient Boosted Regression Trees* (GBRT). Primeiro, ajustaremos um `DecisionTreeRegressor` ao conjunto de treinamento (por exemplo, um conjunto de treinamento quadrático ruidoso):

```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Agora treine um segundo `DecisionTreeRegressor` nos erros residuais cometidos pelo primeiro previsor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Em seguida treinaremos um terceiro regressor nos erros residuais cometidos pelo segundo previsor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Agora, temos um ensemble contendo três árvores. Ele pode fazer previsões em uma nova instância adicionando as previsões de todas as árvores:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

A Figura 7-9 representa as previsões dessas três árvores em sua coluna à esquerda e as previsões do ensemble na coluna da direita. Na primeira linha, o ensemble tem apenas uma árvore, então suas previsões são exatamente as mesmas que as previsões da primeira árvore. Na segunda linha, uma nova árvore é treinada nos erros residuais da primeira. À direita, é possível ver que as previsões do ensemble são iguais à soma das previsões das duas primeiras árvores. Da mesma forma, na terceira fila outra árvore é treinada nos erros residuais da segunda árvore. À medida que as árvores são adicionadas ao ensemble, verificamos que suas previsões melhoram gradualmente

Uma maneira mais simples de treinar ensembles GBRT seria com a utilização da classe `GradientBoostingRegressor` do Scikit-Learn. Tal como a classe `RandomForestRegressor`, ela tem hiperparâmetros para controlar o crescimento das Árvores de Decisão (por exemplo, `max_depth`, `min_samples_leaf` e assim por diante), bem como hiperparâmetros para controlar o treinamento do ensemble, como o número de árvores (`n_estimators`). O código a seguir cria o mesmo ensemble que o anterior:

```
from sklearn.ensemble import GradientBoostingRegressor
gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbdt.fit(X, y)
```

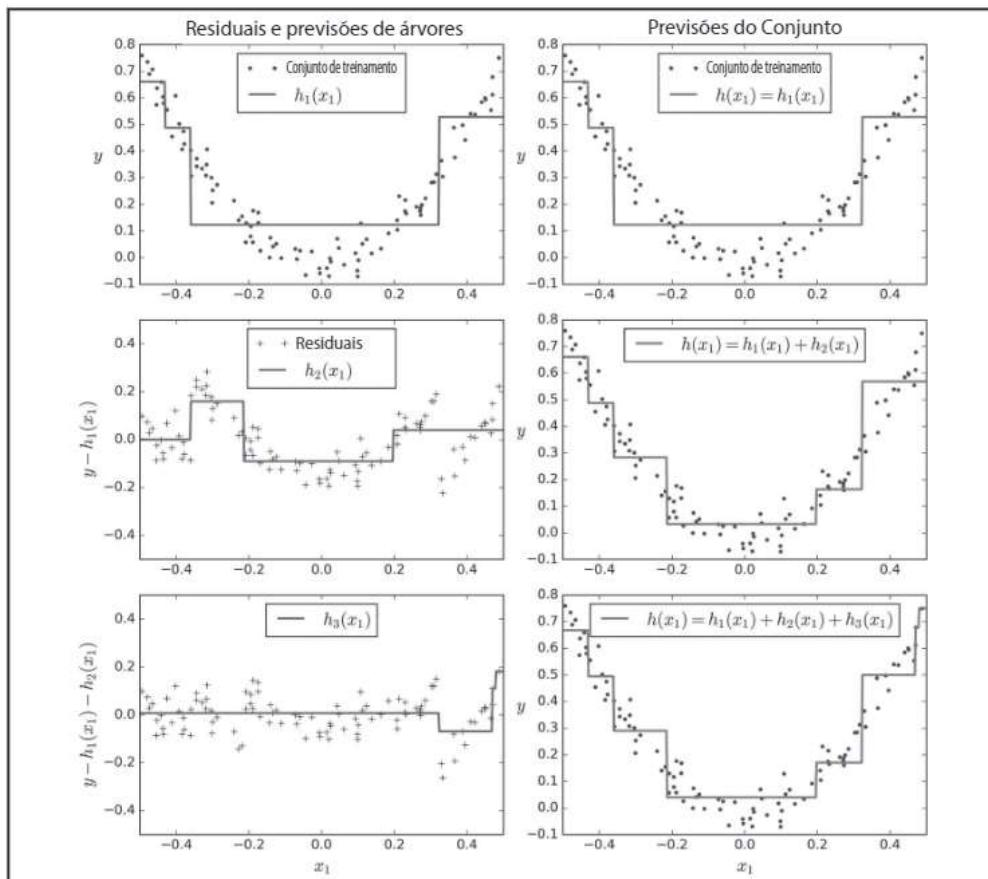
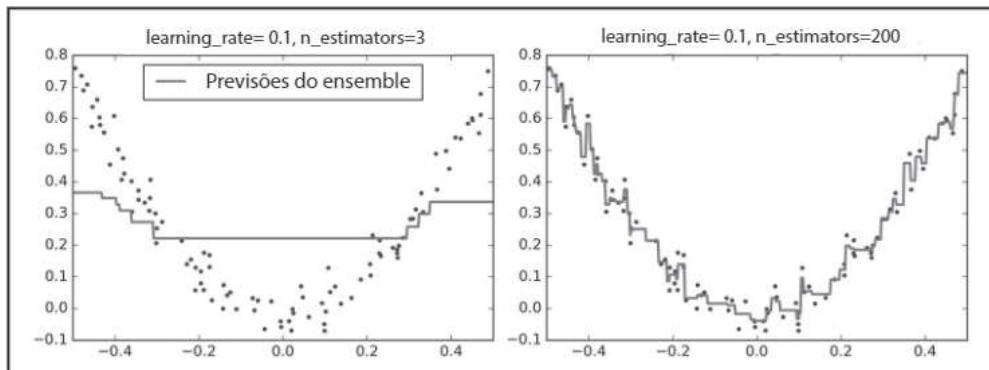


Figura 7-9. Gradient Boosting

O hiperparâmetro `learning_rate` escalona a contribuição de cada árvore. Se você configurá-lo para um valor baixo, como 0,1, precisará de mais árvores no ensemble para se ajustar ao conjunto de treinamento, mas as previsões normalmente generalizarão melhor. Esta é uma técnica de regularização chamada *encolhimento*. A Figura 7-10 mostra dois ensembles GBRT treinados com uma baixa taxa de aprendizado: o do lado esquerdo não possui árvores suficientes para se ajustarem ao conjunto de treinamento, enquanto o do lado direito possui muitas árvores e se sobreajusta ao conjunto de treinamento.



*Figura 7-10. Ensembles GBRT com previsões insuficientes (à esquerda) e demais (à direita)*

Você pode utilizar uma parada antecipada para encontrar o número ideal de árvores (veja o Capítulo 4). Uma maneira simples de implementar isto é com a utilização do método `staged_predict()`: ele retorna um iterador em cada estágio do treinamento sobre as previsões feitas pelo ensemble (com uma árvore, duas árvores, etc.). O código a seguir treina um ensemble GBRT com 120 árvores e mede o erro de validação em cada estágio do treinamento para encontrar o número ideal de árvores e, finalmente treina outro ensemble GBRT:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

Os erros de validação estão representados à esquerda da Figura 7-11, e as previsões do melhor modelo estão representadas à direita.

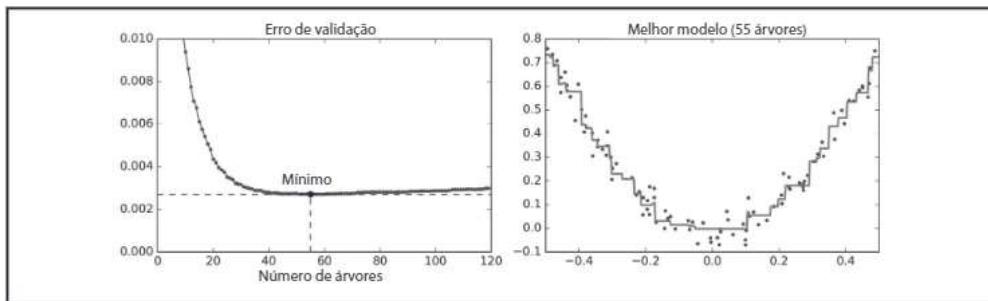


Figura 7-11. Ajustando o número de árvores utilizando uma parada antecipada

Também é possível implementar uma parada antecipada interrompendo antecipadamente o treinamento (em vez de treinar primeiro um grande número de árvores e em seguida olhar para trás para encontrar o número ideal). Você pode fazer isso configurando `warm_start=True`, o que faz com que o Scikit-Learn mantenha as árvores existentes quando recorrer ao método `fit()`, permitindo seu treinamento incremental. O código a seguir deixa de treinar quando o erro de validação não melhora por cinco iterações seguidas:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbdt.n_estimators = n_estimators
    gbdt.fit(X_train, y_train)
    y_pred = gbdt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
    if error_going_up == 5:
        break # parada antecipada
```

A classe `GradientBoostingRegressor` também suporta um hiperparâmetro `subsample`, que especifica a fração das instâncias de treinamento a serem usadas para treinar cada árvore. Por exemplo, se `subsample=0.25`, cada árvore será treinada em 25% das instâncias de treinamento selecionadas aleatoriamente. Como você já deve ter adivinhado, isso troca um viés mais alto por uma variância inferior. Também acelera consideravelmente o treinamento. Esta técnica é chamada de *Stochastic Gradient Boosting*.



É possível utilizar o Gradient Boosting com outras funções de custo. Isso é controlado pelo hiperparâmetro `loss` (consulte a documentação do Scikit-Learn para mais detalhes).

## Stacking

O último Ensemble method que discutiremos neste capítulo é chamado *stacking* (abreviação de *stacked generalization*) (<http://goo.gl/9I2NBw>).<sup>18</sup> Baseia-se em uma ideia simples: em vez de utilizar funções triviais (como hard voting) para agregar as previsões de todos os previsores em um conjunto, por que não treinamos um modelo para agregá-las? A Figura 7-12 mostra esse ensemble executando uma tarefa de regressão em uma nova instância. Cada um dos três previsores abaixo prevê um valor diferente (3,1, 2,7 e 2,9) e, em seguida, o previsor final (chamado de *blender* ou um *meta learner*) escolhe essas previsões como entradas e faz a previsão final (3,0).

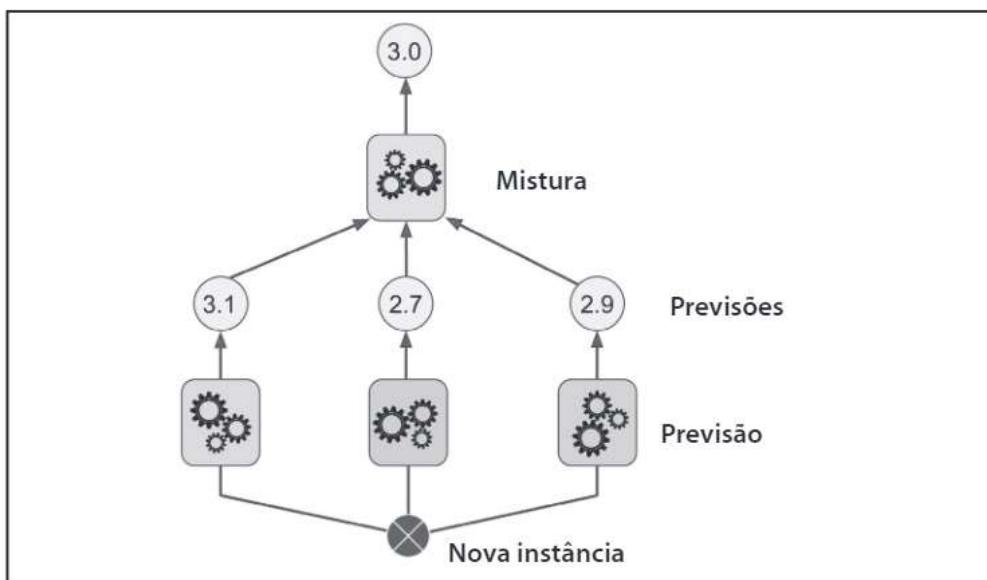


Figura 7-12. Agregando previsões com um previsor blender

Uma abordagem comum é utilizar um conjunto hold-out para treinar o blender.<sup>19</sup> Veremos como isso funciona. Primeiro, o conjunto de treinamento é dividido em dois subconjuntos. O primeiro subconjunto é utilizado para treinar os previsores na primeira camada (veja a Figura 7-13).

<sup>18</sup> "Stacked Generalization", D. Wolpert (1992).

<sup>19</sup> Como alternativa, é possível usar previsões out-of-fold. Em alguns contextos, isso é chamado de *stacking*, enquanto o uso de um conjunto de hold-out é chamado de *blending*. No entanto, para muitas pessoas, esses termos são sinônimos.

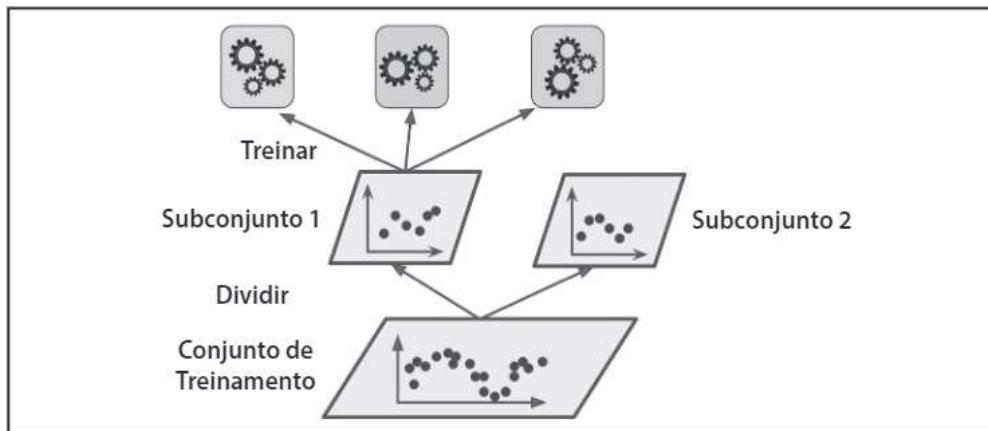


Figura 7-13. Treinando a primeira camada

Em seguida, os previsores da primeira camada são utilizados para fazer previsões no segundo conjunto (hold-out) (veja a Figura 7-14), o que garante que as previsões sejam “limpas”, uma vez que os previsores nunca viram essas instâncias durante o treinamento. Agora, existem três valores previstos para cada instância no conjunto hold-out e utilizando-os como características de entrada, podemos criar um novo conjunto de treinamento (o que torna este novo conjunto tridimensional) e manter os valores de destino. O blender é treinado neste novo conjunto de treinamento, aprendendo a prever o valor do alvo dadas as previsões da primeira camada.

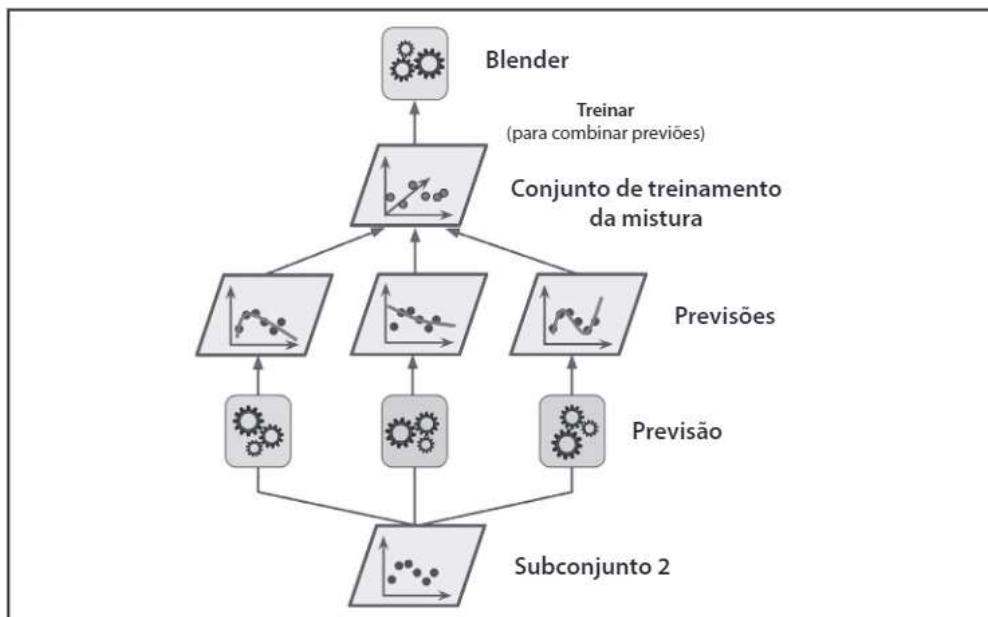


Figura 7-14. Treinando o blender

Na verdade, é possível treinar vários blenders diferentes dessa maneira (por exemplo, em um utilizando Regressão Linear, em outro utilizando Regressão de Floresta Aleatória, e assim por diante), obtendo, desta forma, uma camada completa de blenders. O truque é dividir o conjunto de treinamento em três subconjuntos: o primeiro para treinar a primeira camada, o segundo para criar o conjunto de treinamento usado para treinar a segunda camada (utilizando as previsões feitas pelos previsores da primeira camada), e o terceiro para criar o conjunto de treinamento para treinar a terceira camada (utilizando previsões feitas pelos previsores da segunda camada). Uma vez feito isso, podemos fazer uma previsão para uma nova instância passando sequencialmente por cada camada, como mostrado na Figura 7-15.

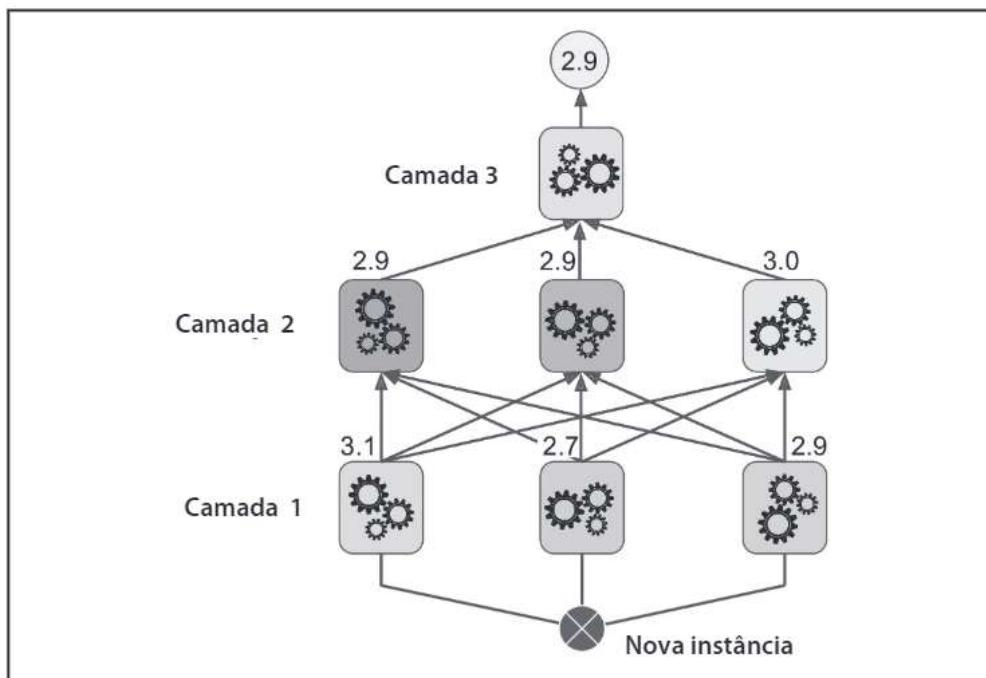


Figura 7-15. Previsões em um ensemble stacking de multicamadas

Infelizmente, o Scikit-Learn não suporta stacking diretamente, mas não é muito difícil introduzir sua própria implementação (veja os seguintes exercícios). Como alternativa, você pode utilizar uma implementação de código aberto como o `brew` (disponível em <https://github.com/viisar/brew>).

## Exercícios

1. Se você treinou cinco modelos diferentes com exatamente os mesmos dados de treinamento e todos conseguem uma precisão de 95%, existe alguma chance de você combinar esses modelos para obter melhores resultados? Em caso afirmativo, como? Se não, por quê?
2. Qual a diferença entre os classificadores de votação hard e soft?
3. É possível acelerar o treinamento de um bagging ensemble distribuindo-o por vários servidores? E com pasting ensembles, boosting ensembles, florestas aleatórias, ou stacking ensembles?
4. Qual é o benefício da avaliação out-of-bag?
5. O que torna as Árvores-Extras mais aleatórias do que as Florestas Aleatórias comuns? Como esta aleatoriedade extra pode ajudar? As Árvores-Extras são mais lentas ou mais rápidas do que as Florestas Aleatórias regulares?
6. Se o seu ensemble AdaBoost se subajusta aos dados de treinamento, quais hiperparâmetros você deve ajustar e como?
7. Se o seu ensemble Gradient Boosting se sobreajusta ao conjunto de treinamento, você deve aumentar ou diminuir a taxa de aprendizado?
8. Carregue os dados MNIST (introduzido no Capítulo 3) e o divida em um conjunto de treinamento, um conjunto de validação e um conjunto de teste (por exemplo, utilize 40 mil instâncias para treinamento, 10 mil para validação e 10 mil para teste). Em seguida, treine vários classificadores como um classificador Floresta Aleatória, um classificador Árvores-extra e um SVM. Em seguida, utilizando um classificador de votação soft ou hard, tente combiná-los em um ensemble que supere todos no conjunto de validação. Uma vez tendo encontrado o ensemble, teste-o no conjunto de teste. Qual é a melhoria de desempenho em comparação com os classificadores individuais?
9. Execute os classificadores individuais do exercício anterior para fazer previsões no conjunto de validação e crie um novo conjunto de treinamento com as previsões resultantes: cada instância de treinamento é um vetor que contém o conjunto de previsões de todos os seus classificadores para uma imagem e o alvo é a classe da imagem. Parabéns, você acabou de treinar um blender, e, junto com os classificadores, eles formam um stacking ensemble! Agora, avaliaremos o conjunto no conjunto de testes. Para cada imagem no conjunto de teste, faça previsões com todos os seus classificadores, então forneça as previsões ao blender para obter as previsões do ensemble. Como ela se compara ao classificador de votação que você treinou anteriormente?

As soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 8

# Redução da Dimensionalidade

Muitos problemas de Aprendizado de Máquina envolvem milhares ou mesmo milhões de características para cada instância de treinamento. Como veremos, isso não só torna o treinamento extremamente lento, mas também torna muito mais difícil encontrar uma boa solução. Este problema muitas vezes é referido como a *maldição da dimensionalidade*.

Felizmente, é possível transformar um problema insolúvel em um problema tratável no mundo real ao reduzir consideravelmente o número de características abordadas. Por exemplo, considere as imagens do MNIST (introduzidas no Capítulo 3): os pixels nas bordas da imagem são quase sempre brancos, então você pode descartá-los completamente do conjunto de treinamento sem perder muita informação. A Figura 7-6 confirma que esses pixels não têm importância alguma para a tarefa de classificação. Além disso, dois pixels vizinhos são muitas vezes altamente correlacionados: se você mesclá-los em um único pixel (por exemplo, tomando a média das duas intensidades dos pixels), não perderá muita informação.



Ao reduzir a dimensionalidade, perdemos alguma informação (assim como a compressão de uma imagem para JPEG pode degradar sua qualidade), por isso, embora acelere o treinamento, também pode fazer com que seu sistema funcione um pouco pior. Isso também torna seus pipelines um pouco mais complexos e, portanto, mais difíceis de manter. Você deve primeiro tentar treinar seu sistema com os dados originais antes de considerar a redução da dimensionalidade caso o treinamento esteja muito lento. Em alguns casos, no entanto, reduzir a dimensionalidade dos dados de treinamento poderá filtrar algum ruído e detalhes desnecessários, e resultar em melhor desempenho (mas em geral isso não acontecerá; apenas acelerará o treinamento).

Além de acelerar o treinamento, a redução da dimensionalidade também é extremamente útil para visualização de dados (ou *DataViz*). Reduzir o número de dimensões para duas (ou três) permite plotar em um gráfico um conjunto de treinamento de alta dimensão, e

muitas vezes obter alguns insights importantes por meio da detecção visual de padrões, como os clusters.

Neste capítulo, discutiremos a maldição da dimensionalidade e teremos uma noção do que se passa no espaço da alta dimensão. Apresentaremos as duas abordagens principais para a redução da dimensionalidade (projeção e Manifold Learning) e passaremos por três das técnicas mais populares da redução de dimensionalidade: PCA, Kernel PCA e LLE.

## A Maldição da Dimensionalidade

Estamos tão acostumados a viver em três dimensões<sup>1</sup> que nossa intuição falha quando tentamos imaginar um espaço de alta dimensão. Nossa mente tem uma dificuldade extrema de projetar até mesmo um hipercubo básico 4D (veja a Figura 8-1), que dirá uma elipsóide de 200 dimensões dobrada em um espaço de mil dimensões.

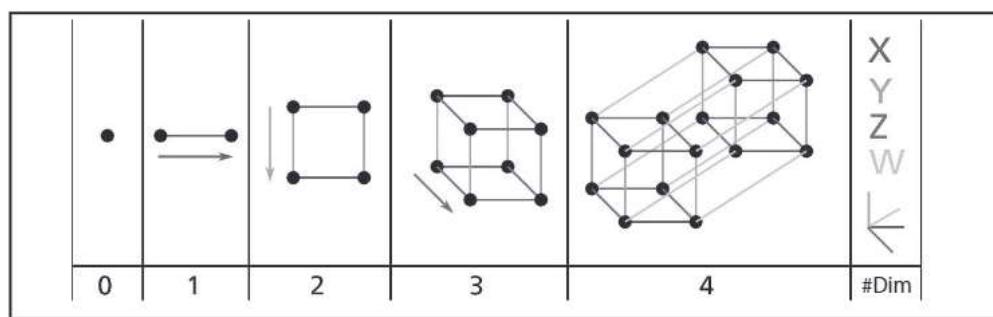


Figura 8-1. Ponto, segmento, quadrado, cubo e tesserato (hipercubos 0D a 4D)<sup>2</sup>

Acontece que muitas coisas se comportam de forma bem diferente no espaço da alta dimensão. Por exemplo, se você escolher um ponto aleatório em um quadrado unitário (um quadrado  $1 \times 1$ ), ele terá apenas 0,4% de chance de estar localizado a menos de 0,001 de uma borda (em outras palavras, é muito improvável que um ponto aleatório seja “extremo” ao longo de qualquer dimensão). Mas, em um hipercubo de 10 mil dimensões (um cubo  $1 \times 1 \times \dots \times 1$ , com dez mil 1s), esta probabilidade é maior que 99,999999%. A maioria dos pontos em um hipercubo de alta dimensão está muito próxima da borda.<sup>3</sup>

1 Bem, quatro dimensões se você contar o tempo, e algumas mais se você concordar com a teoria das cordas.

2 Assista a um tesserato rotativo projetado no espaço 3D em <http://goo.gl/OM7ktJ>. Imagem da Wikipédia disponibilizada pelo usuário NerdBoy1392 (Creative Commons BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)). Reproduzido em <https://en.wikipedia.org/wiki/Tesseract>.

3 Fato divertido: qualquer um que você conheça provavelmente é um extremista em pelo menos uma dimensão (por exemplo, quanto açúcar eles colocam em seu café), se você considerar dimensões suficientes.

Eis uma diferença mais problemática: se você escolher dois pontos aleatoriamente em um quadrado, a distância entre esses dois pontos será, na média, aproximadamente 0,52. Se você escolher dois pontos aleatórios em um cubo 3D, a distância média será de aproximadamente 0,66. Mas e quanto a dois pontos escolhidos aleatoriamente em um hipercubo de 1 milhão de dimensões? Bem, a distância média, acredite ou não, será de mais ou menos 408,25 (aproximadamente  $\sqrt{1.000.000/6!}$  ), o que é bastante contraintuitivo: como dois pontos podem ser tão distantes quando ambos ficam dentro do mesmo hipercubo? Esse fato implica que os conjuntos de dados de alta dimensão correm o risco de serem muito escassos: a maioria das instâncias de treinamento provavelmente estará longe uma da outra. Claro, isso também significa que uma nova instância provavelmente estará longe de qualquer outra instância de treinamento, tornando as previsões muito menos confiáveis do que nas dimensões inferiores, uma vez que serão baseadas em extrapolações muito maiores. Em suma, quanto mais dimensões o conjunto de treinamento tiver, maior o risco de sobreajustá-lo.

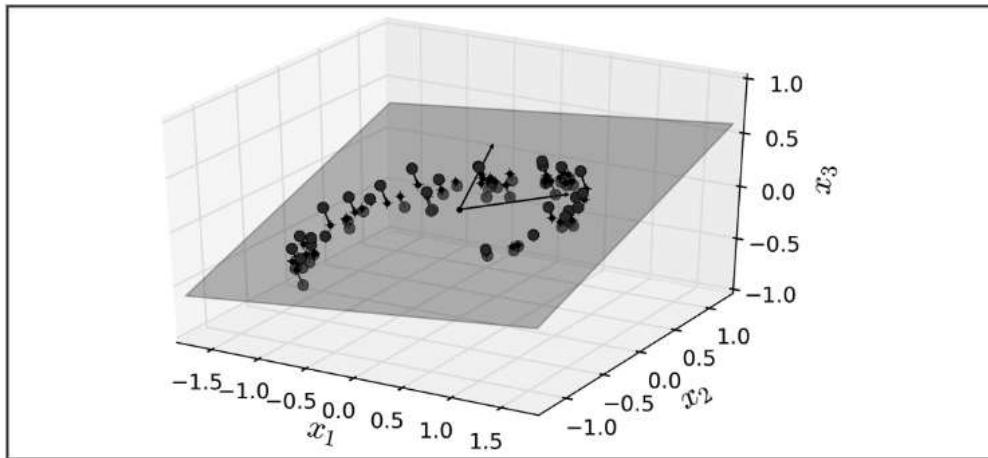
Na teoria, uma solução para a maldição da dimensionalidade poderia ser aumentar o tamanho do conjunto de treinamento para atingir uma densidade suficiente de instâncias. Infelizmente, na prática, o número de instâncias de treinamento necessárias para atingir uma determinada densidade cresce exponencialmente com o número de dimensões. Com apenas 100 características (muito menos do que no problema da MNIST), você precisaria de mais instâncias de treinamento do que os átomos no universo observável para que essas instâncias estivessem a 0,1 de cada uma em média, assumindo que estivessem espalhadas uniformemente em todas as dimensões.

## Principais Abordagens para a Redução da Dimensionalidade

Antes de mergulharmos em algoritmos específicos de redução da dimensionalidade, veremos as duas principais abordagens para reduzi-la: Projeção e Manifold Learning.

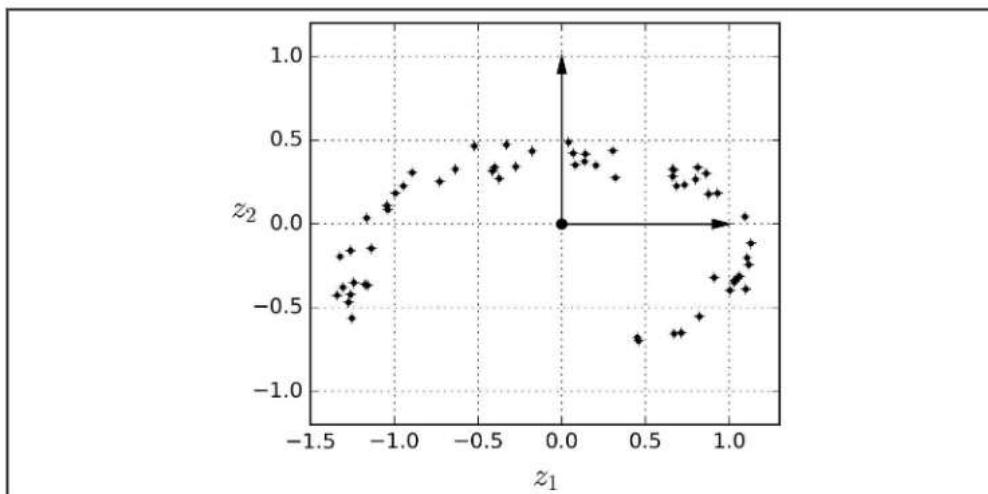
### Projeção

Na maioria dos problemas do mundo real, as instâncias de treinamento *não* se espalham uniformemente em todas as dimensões. Muitas características são quase constantes, enquanto outras estão altamente correlacionadas (como discutido anteriormente no MNIST). Como resultado, todas as instâncias de treinamento estão realmente dentro (ou perto) de um *subespaço* de dimensão bem menor no espaço de alta dimensão, mas como isto parece muito abstrato, vejamos um exemplo. Na Figura 8-2, podemos ver um conjunto de dados 3D representado pelos círculos.



*Figura 8-2. Um conjunto de dados 3D perto de um subespaço 2D*

Observe que todas as instâncias de treinamento estão próximas de um plano: este é um subespaço (2D) de dimensões inferiores em um espaço de alta dimensão (3D). Agora, se projetarmos perpendicularmente cada instância de treinamento a este subespaço (como representado pelas linhas curtas que conectam as instâncias ao plano), obtemos o novo conjunto de dados 2D mostrado na Figura 8-3. Sim! Acabamos de reduzir a dimensionalidade do conjunto de dados de 3D para 2D. Observe que os eixos correspondem às novas características  $z_1$  e  $z_2$  (as coordenadas das projeções no plano).



*Figura 8-3. O novo conjunto de dados 2D após a projeção*

No entanto, a projeção nem sempre é a melhor abordagem para a redução da dimensionalidade. Em muitos casos, o subespaço pode torcer e girar como no famoso conjunto de dados em rolo suíço representado na Figura 8-4.

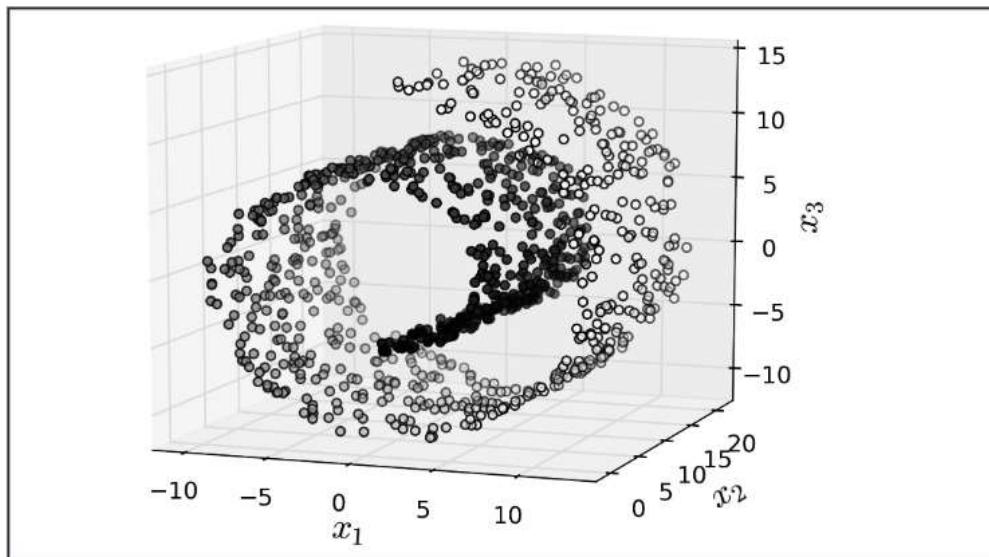


Figura 8-4. Conjunto de dados do rolo suíço

Simplesmente projetar em um plano (por exemplo, descartando  $x_3$ ) comprimiria diferentes camadas do rolo suíço, como mostrado à esquerda na Figura 8-5. No entanto, o que você realmente quer é desenrolar o rolo para obter o conjunto de dados 2D à direita na Figura 8-5.

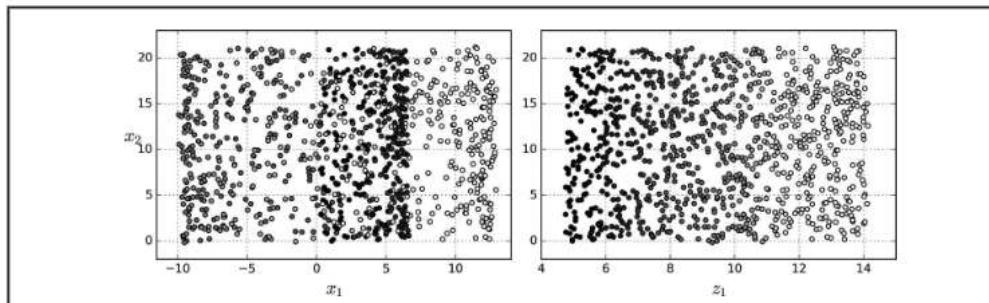


Figura 8-5. Esmagar o rolo ao projetá-lo em um plano (esquerda) versus desenrolar o rolo suíço (direita)

## Manifold Learning

O rolo suíço é um exemplo de um *manifold* 2D. Simplificando, um manifold 2D é uma forma 2D que pode ser dobrada e torcida em um espaço de dimensões maiores. Geralmente, um manifold  $d$ -dimensional é uma parte de um espaço  $n$ -dimensional (em que  $d < n$ ) que se assemelha localmente a um hiperplano  $d$ -dimensional. No caso do rolo suíço,  $d = 2$  e  $n = 3$ : ele se assemelha localmente a um plano 2D, mas é enrolado na terceira dimensão.

Muitos algoritmos da redução de dimensionalidade funcionam modelando-se o *manifold* onde estão as instâncias de treinamento; isso é chamado *Manifold Learning*. Baseia-se na *manifold assumption*, também chamada de *manifold hypothesis*, que sustenta que a maioria dos conjuntos de dados de alta dimensão do mundo real está próxima de um manifold muito mais baixo. Esta suposição frequentemente é observada empiricamente.

Mais uma vez, pense no conjunto de dados do MNIST: todas as imagens manuscritas dos dígitos têm algumas semelhanças. Elas são feitas de linhas conectadas, as bordas são brancas, elas estão mais ou menos centradas, e assim por diante. Se você as gerou aleatoriamente, apenas uma fração minúscula delas seria semelhante a dígitos escritos a mão. Em outras palavras, os graus de liberdade disponíveis quando se tenta criar uma imagem numérica são dramaticamente inferiores aos graus de liberdade que você teria se pudesse gerar outra imagem qualquer. Essas restrições tendem a comprimir o conjunto de dados em um manifold de dimensão inferior.

A *manifold assumption* é muitas vezes acompanhada por outra suposição implícita: a tarefa em questão (por exemplo, classificação ou regressão) será mais simples se for expressa em um espaço de dimensão inferior do manifold. Por exemplo, na linha superior da Figura 8-6, o rolo suíço é dividido em duas classes: no espaço 3D (à esquerda), a fronteira de decisão seria bastante complexa, mas no espaço manifold 2D desenrolado (à direita), a fronteira de decisão é uma linha reta simples.

No entanto, essa suposição nem sempre é válida. Por exemplo, na linha inferior da Figura 8-6, a fronteira de decisão está localizada em  $x_1 = 5$ . Esta fronteira de decisão parece muito simples no espaço 3D original (um plano vertical), mas parece mais complexa no manifold desenrolado (uma coleção de quatro segmentos de linha independentes).

Em resumo, se você reduzir a dimensionalidade em seu conjunto de treinamento antes de treinar um modelo, ele definitivamente acelerará o treinamento, mas nem sempre poderá levar a uma solução melhor ou mais simples; tudo depende do conjunto de dados.

Espero que você tenha agora uma boa noção do que é a maldição da dimensionalidade e como seus algoritmos de redução da dimensionalidade podem lutar contra ela, especialmente quando a *manifold assumption* se sustenta. O restante deste capítulo aborda alguns dos algoritmos mais populares.

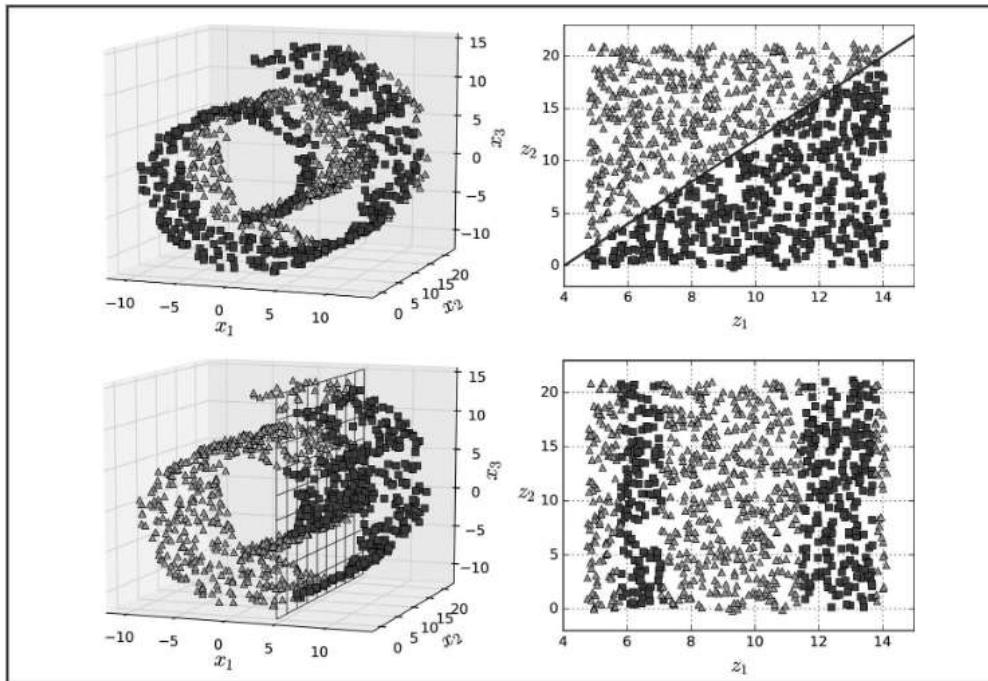


Figura 8-6. A fronteira de decisão nem sempre é mais simples com dimensões inferiores

## PCA

A Análise dos Componentes Principais (PCA) é, de longe, o algoritmo de redução de dimensionalidade mais popular. Primeiro, ele identifica o hiperplano que se encontra mais próximo dos dados e, então, projeta os dados sobre ele.

### Preservando a Variância

Antes de projetar o conjunto de treinamento em um hiperplano de dimensões inferiores, é preciso escolher o hiperplano correto. Por exemplo, um simples conjunto de dados 2D é representado à esquerda da Figura 8-7 juntamente com três eixos diferentes (ou seja, hiperplanos unidimensionais). À direita está o resultado da projeção do conjunto de dados em cada um desses eixos. Como você pode ver, a projeção na linha sólida preserva a variância máxima, enquanto a projeção na linha pontilhada preserva pouca variância e a projeção na linha tracejada preserva uma variância intermediária.

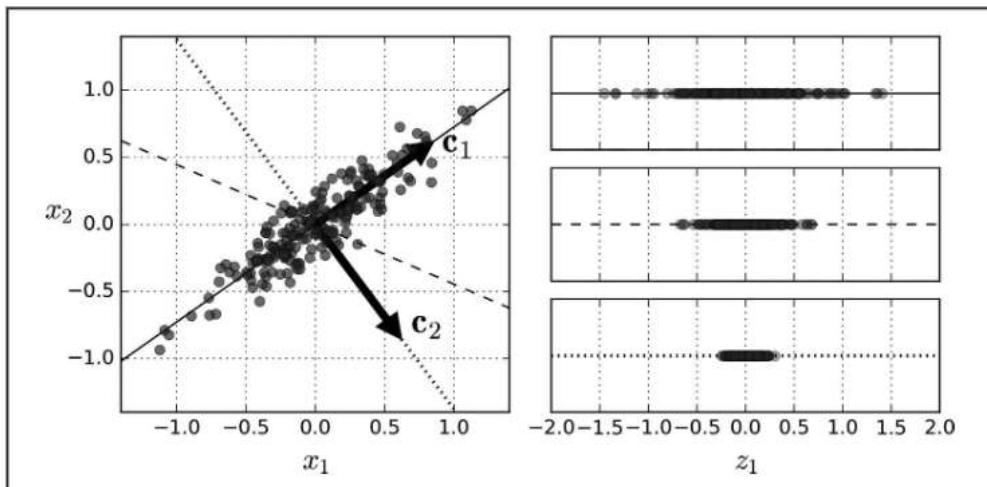


Figura 8-7. Selecionando o subespaço no qual projetar

Parece razoável selecionar o eixo que preserva a quantidade máxima de variância, pois provavelmente ela perderá menos informações do que as outras projeções. O fato de o eixo minimizar a distância quadrática média entre o conjunto original de dados e sua projeção nesse eixo é outra maneira de justificar essa escolha. Esta é a ideia simples por trás do PCA (<http://goo.gl/gbNoID>).<sup>4</sup>

## Componentes Principais

O PCA identifica o eixo que representa a maior quantidade de variância no conjunto de treinamento. É a linha sólida na Figura 8-7. Ela também encontra um segundo eixo ortogonal ao primeiro, que representa a maior quantidade remanescente de variância. Neste exemplo 2D, não há escolha: é a linha pontilhada. Se fosse um conjunto de dados de dimensão mais alta, o PCA também encontraria um terceiro eixo ortogonal aos dois eixos anteriores, e um quarto, um quinto, e assim por diante — tantos eixos quanto o número de dimensões no conjunto de dados.

O vetor da unidade que define o  $i^{\text{ésimo}}$  eixo é chamado de  $i^{\text{ésimo}} \text{ componente principal (PC)}$ . Na Figura 8-7, o  $1^{\text{o}}$  PC é  $c_1$  e o  $2^{\text{o}}$  PC é  $c_2$ . Na Figura 8-2, os dois primeiros PC estão representados no plano pelas setas ortogonais e o terceiro PC seria ortogonal ao plano (apontando para cima ou para baixo).

<sup>4</sup> “Onlines and Planes of Closest Fit to Systems of Points in, K. Pearson (1901).



A direção dos componentes principais não é estável: se você perturbar ligeiramente o conjunto de treinamento e executar o PCA novamente, alguns dos novos PC podem apontar na direção oposta dos PC originais. No entanto, eles ainda permanecem nos mesmos eixos. Em alguns casos, um par de PC pode até girar ou trocar, mas o plano que eles definem geralmente permanece o mesmo.

Então, como é possível encontrar os principais componentes de um conjunto de treinamento? Felizmente, existe uma técnica padrão de fatoração da matriz chamada *Decomposição em Valores Singulares* (SVD, em inglês) que pode decompor a matriz  $X$  do conjunto de treinamento em um produto escalar de três matrizes  $U \cdot \Sigma \cdot V^T$ , sendo que  $V$  contém todos os componentes principais que buscamos, como mostrado na Equação 8-1.

*Equação 8-1. Matriz dos componentes principais*

$$V = \begin{pmatrix} | & | & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & | \end{pmatrix}$$

O seguinte código Python utiliza a função `svd()` do NumPy para obter todos os componentes principais do conjunto de treinamento e, em seguida, extraí os dois primeiros PC:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```



O PCA assume que o conjunto de dados está centrado em torno da origem. Como veremos, as classes PCA do Scikit-Learn cuidam de centrar os dados para você. No entanto, se implementarmos o PCA por conta própria (como no exemplo anterior), ou se utilizarmos outras bibliotecas, não podemos esquecer de centralizar os dados primeiro.

## Projetando para d Dimensões

Assim que você identificar todos os componentes principais, pode reduzir a dimensionalidade do conjunto de dados a  $d$  dimensões projetando no hiperplano definido pelos  $d$  primeiros componentes principais. Selecionar este hiperplano garante que a projeção preservará a maior quantidade possível de variância. Por exemplo, na Figura 8-2, o conjunto de dados 3D é projetado no plano 2D definido pelos dois primeiros componentes principais, preservando uma grande parte da variância do conjunto de dados. Como resultado, a projeção 2D se parece muito com o conjunto de dados 3D original.

Para projetar o conjunto de treinamento no hiperplano, simplesmente calcule o produto escalar da matriz  $X$  do conjunto de treinamento pela matriz  $W_d$ , definida como a matriz que contém os primeiros componentes principais  $d$  (isto é, a matriz composta das primeiras colunas  $d$  de  $V$ ), como mostrado na Equação 8-2.

*Equação 8-2. Projetando o conjunto de treinamento para d dimensões*

$$X_{d\text{-proj}} = X \cdot W_d$$

O seguinte código Python projeta o conjunto de treinamento no plano definido pelos dois primeiros componentes principais:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

Aí está! Agora você sabe como reduzir a dimensionalidade de qualquer conjunto de dados para qualquer número de dimensões preservando a maior quantidade possível de variância.

## Utilizando o Scikit-Learn

A classe PCA do Scikit-Learn implementa o PCA com a utilização da decomposição SVD como fizemos antes. O código a seguir aplica o PCA para reduzir a dimensionalidade a duas dimensões no conjunto de dados (observe que ela automaticamente cuida da centralização dos dados):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

Após ajustar o transformador `PCA` ao conjunto de dados utilizando a variável `components_`, acesse os componentes principais (observe que ela contém os PC como vetores horizontais; então, por exemplo, o primeiro componente principal é igual a `pca.components_.T[:, 0]`).

## Taxa de Variância Explícada

Outra informação útil é a *taxa de variância explicada* de cada componente principal, disponível na variável `explained_variance_ratio_`. Ela indica a proporção da variância do conjunto de dados que se encontra ao longo do eixo de cada componente principal. Por exemplo, vejamos as taxas de variância explicadas dos dois primeiros componentes do conjunto de dados 3D representado na Figura 8-2:

```
>>> pca.explained_variance_ratio_
array([ 0.84248607,  0.14631839])
```

Isso indica que 84,2% da variância do conjunto de dados está ao longo do primeiro eixo e 14,6% situa-se ao longo do segundo eixo. Isso deixa menos de 1,2% para o terceiro eixo, por isso é razoável supor que ele provavelmente carrega pouca informação.

## Escolhendo o Número Certo de Dimensões

Em vez de escolher arbitrariamente o número de dimensões a serem reduzidas, é preferível escolher o número de dimensões que adicionam uma porção suficientemente grande de variância (por exemplo, 95%). A menos que, claro, você esteja reduzindo a dimensionalidade para a visualização de dados — nesse caso, reduza a dimensionalidade para 2 ou 3.

O código a seguir calcula o PCA sem reduzir a dimensionalidade e, em seguida, calcula o número mínimo de dimensões necessárias para preservar 95% da variância no conjunto de treinamento:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

Você poderia ajustar `n_components=d` e executar novamente o PCA. No entanto, existe uma opção bem melhor: em vez de especificar o número de componentes principais que você quer preservar, ajuste `n_components` para flutuar entre 0.0 e 1.0, indicando a taxa de variância que você quer preservar:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Outra opção seria plotar a variância explicada como uma função do número de dimensões (simplesmente plote `cumsum`, veja a Figura 8-8). Geralmente haverá um cotovelo na curva no qual a variância explicada rapidamente deixa de crescer. Você pode pensar nisso como a dimensionalidade intrínseca do conjunto de dados. Neste caso, você verá que a redução da dimensionalidade para cerca de 100 dimensões não perderia muito da variância explicada.

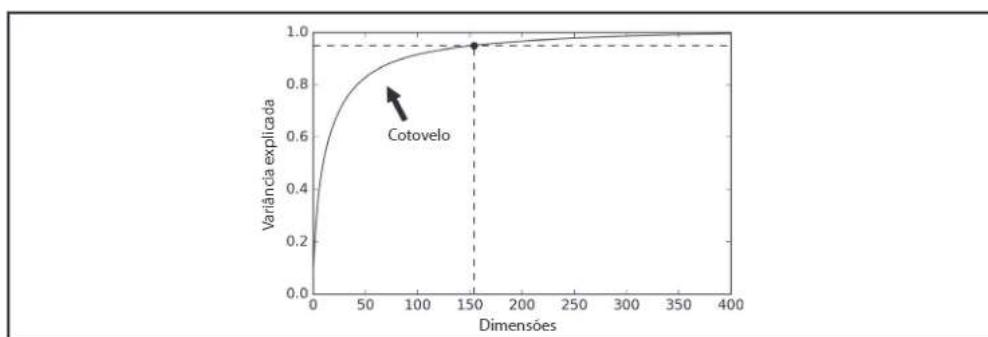


Figura 8-8. Variação explicada como uma função do número de dimensões

## PCA para a Compressão

É óbvio que, após a redução da dimensionalidade, o conjunto de treinamento ocupa muito menos espaço. Por exemplo, tente aplicar o PCA no conjunto de dados MNIST preservando 95% de sua variação. Você verá que cada instância terá pouco mais de 150 características em vez das 784 originais. Assim, enquanto a maior parte da variância é preservada, o conjunto de dados está agora com menos de 20% do seu tamanho original! Esta é uma taxa de compressão razoável e você verá como isso pode acelerar tremenda-mente um algoritmo de classificação (como o classificador SVM).

Também é possível descomprimir o conjunto de dados reduzido de volta para 784 dimensões aplicando a transformação inversa da projeção PCA. Claro que isso não trará de volta os dados originais, uma vez que a projeção perde um pouco de informação (dentro da variância de 5% que foi descartada), mas provavelmente será bastante próxima dos dados originais. A distância quadrática média entre os dados originais e os dados reconstruídos (comprimidos e depois descomprimidos) é chamada de *erro de reconstrução*. Por exemplo, o código a seguir comprime o conjunto de dados MNIST para 154 dimensões, depois utiliza o método `inverse_transform()` para descomprimi-lo de volta para 784 dimensões. A Figura 8-9 mostra alguns dígitos do conjunto de treinamento original (à esquerda) e os dígitos correspondentes após a compressão e a descompressão. Veja que há uma ligeira perda de qualidade da imagem, mas os dígitos ainda estão intactos.

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

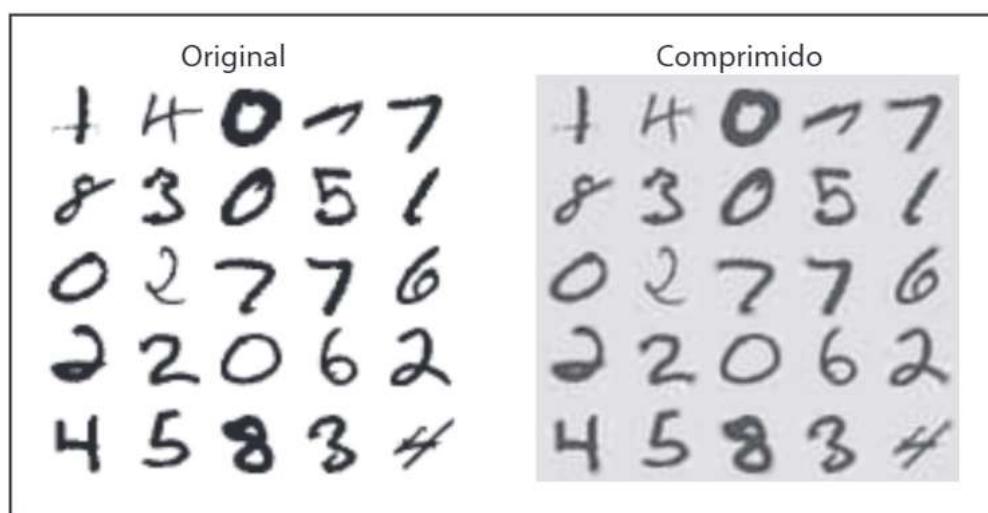


Figura 8-9. Compressão do MNIST preservando 95% da variância

A transformação inversa é mostrada na Equação 8-3.

*Equação 8-3. Transformação inversa do PCA de volta ao número original de dimensões*

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \cdot \mathbf{W}_d^T$$

## PCA Incremental

Um problema com a implementação do PCA anterior é que ele exige que todo o conjunto de treinamento caiba na memória para que o algoritmo SVD seja executado. Felizmente, foram desenvolvidos os algoritmos *PCA Incremental* (IPCA): você pode dividir o conjunto de treinamento em minilotes e alimentar um algoritmo de IPCA com um minilote por vez. Isso é útil para grandes conjuntos de treinamento e também para aplicar o PCA online (ou seja, em tempo real, à medida que chegam novas instâncias).

O código a seguir divide o conjunto de dados MNIST em 100 minilotes (utilizando a função do NumPy `array_split()`), os fornece à classe `IncrementalPCA` do Scikit-Learn (<http://goo.gl/FmdhUP>)<sup>5</sup> e reduz a dimensionalidade do conjunto de dados MNIST para 154 dimensões (como antes). Chame o método `partial_fit()` em cada minilote em vez do método `fit()` para todo o conjunto de treinamento:

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

Alternativamente, você pode utilizar a classe `memmap` do NumPy, que permite a manipulação de um grande array armazenado em um arquivo binário no disco como se estivesse inteiramente na memória; a classe carrega na memória somente os dados de que precisa, quando precisa. Como a classe `IncrementalPCA` utiliza apenas uma pequena parte do array em um dado momento, o uso da memória permanece sob controle. Isto possibilita chamar ao método usual `fit()`, como podemos ver no código a seguir:

```
X_mm = np.memmap(filename, dtype="float32", mode="readonly", shape=(m, n))

batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

---

<sup>5</sup> O Scikit-Learn utiliza o algoritmo descrito em “Incremental Learning for Robust Visual Tracking”, D. Ross *et al.* (2007).

## PCA Randomizado

O Scikit-Learn oferece ainda outra opção para executar o PCA, o algoritmo chamado *PCA Randomizado*. Este é um algoritmo estocástico que rapidamente encontra uma aproximação dos primeiros componentes principais  $d$ . Sua complexidade computacional é  $O(m \times d^2) + O(d^3)$ , em vez de  $O(m \times n^2) + O(n^3)$ , portanto é dramaticamente mais rápida do que os algoritmos anteriores quando  $d$  é muito menor do que  $n$ .

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

## Kernel PCA

No Capítulo 5, discutimos o truque do kernel, uma técnica matemática que implicitamente mapeia instâncias em um espaço de alta dimensão (chamado de *espaço de características*), permitindo a classificação não linear e a regressão com as Máquinas de Vetores de Suporte. Lembre-se de que uma fronteira de decisão linear em uma característica de espaço de alta dimensão corresponde a uma fronteira complexa de decisão não linear no *espaço original*.

Acontece que o mesmo truque pode ser aplicado ao PCA, possibilitando a realização de projeções não lineares complexas para a redução da dimensionalidade. Isso é chamado de *Kernel PCA* (kPCA) (<http://goo.gl/5lQT5Q>).<sup>6</sup> Ele frequentemente é eficaz na preservação de grupos de instâncias após a projeção, ou às vezes até mesmo para desenrolar conjuntos de dados que estão próximos a um manifold retorcido.

Por exemplo, o código a seguir utiliza a classe `KernelPCA` do Scikit-Learn para realizar o kPCA com o kernel *RBF* (veja o Capítulo 5 para mais detalhes sobre o kernel *RBF* e os outros kernels):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

A Figura 8-10 mostra o rolo suíço reduzido a duas dimensões com a utilização de um kernel linear (equivalente a simplesmente utilizar a classe `PCA`), um kernel *RBF* e um *kernel sigmoidal* (Logística).

---

<sup>6</sup> “Kernel Principal Component Analysis,” B. Schölkopf, A. Smola, K. Müller (1999).

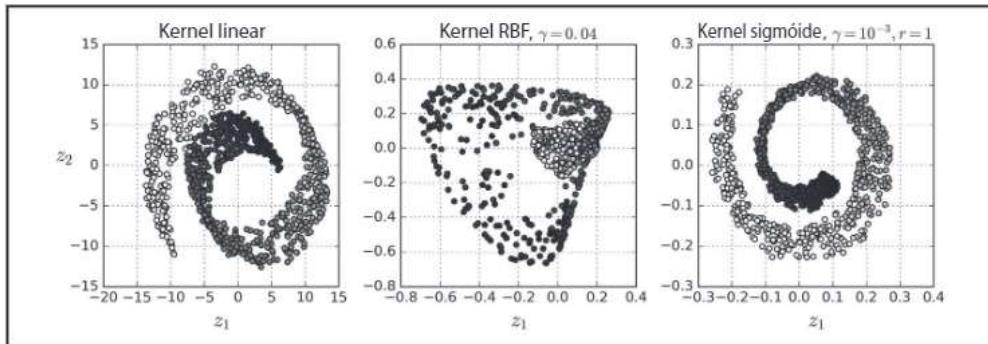


Figura 8-10. Rolo suíço reduzido para 2D utilizando kPCA com vários kernels

## Selecionando um Kernel e Ajustando Hiperparâmetros

Como o kPCA é um algoritmo de aprendizado não supervisionado, não existe uma medida de desempenho óbvia para ajudá-lo a selecionar os melhores valores de kernel e hiperparâmetros. No entanto, a redução da dimensionalidade muitas vezes é um passo de preparação para uma tarefa de aprendizado supervisionado (por exemplo, a classificação), então você pode simplesmente usar a grid search para selecionar o kernel e os hiperparâmetros que conduzam ao melhor desempenho dessa tarefa. Por exemplo, o código a seguir cria um pipeline de duas etapas, primeiro reduzindo a dimensionalidade para duas dimensões com a utilização do kPCA, e em seguida aplicando a Regressão Logística para a classificação. Então, ele utiliza o Grid SearchCV para encontrar o melhor valor do kernel e de gama para o kPCA a fim de obter a melhor acurácia de classificação ao final do pipeline:

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("k pca", KernelPCA(n_components=2)),
    ("log reg", LogisticRegression())
])

param_grid = [
    {"k pca__gamma": np.linspace(0.03, 0.05, 10),
     "k pca__kernel": ["rbf", "sigmoid"]}
]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Os melhores kernel e hiperparâmetros estão disponíveis pela variável `best_params_`:

```
>>> print(grid_search.best_params_)
{'kPCA_gamma': 0.04333333333333335, 'kPCA_kernel': 'rbf'}
```

Outra abordagem, desta vez inteiramente não supervisionada, é a seleção do kernel e dos hiperparâmetros que produzem o menor erro de reconstrução. No entanto, a reconstrução não é tão fácil quanto com o PCA linear. Veja o porquê. A Figura 8-11 mostra o conjunto de dados 3D original do rolo suíço (superior esquerdo) e o conjunto de dados 2D resultante utilizando um Kernel RBF após o kPCA ser aplicado (canto superior direito). Graças ao truque do kernel, isto é matematicamente equivalente ao mapeamento do conjunto de treinamento para um espaço de características de dimensão infinita (à direita, abaixo) com a utilização do *mapa de características*  $\phi$ , em seguida projetando o conjunto de treinamento transformado para 2D com o PCA linear. Observe que, se pudéssemos inverter o passo do PCA linear para uma determinada instância no espaço reduzido, o ponto reconstruído ficaria no espaço de característica e não no espaço original (por exemplo, como o representado por um x no diagrama). Uma vez que o espaço de característica é de dimensões infinitas, não podemos calcular o ponto reconstruído e, portanto, não podemos calcular o verdadeiro erro de reconstrução. Felizmente, é possível encontrar um ponto no espaço original que possa mapear perto do ponto reconstruído, o que é chamado de reconstrução da *pré-imagem*. Depois de ter esta pré-imagem, podemos medir sua distância ao quadrado para a instância original. Podemos, então, selecionar o kernel e os hiperparâmetros que minimizam este erro de reconstrução da pré-imagem.

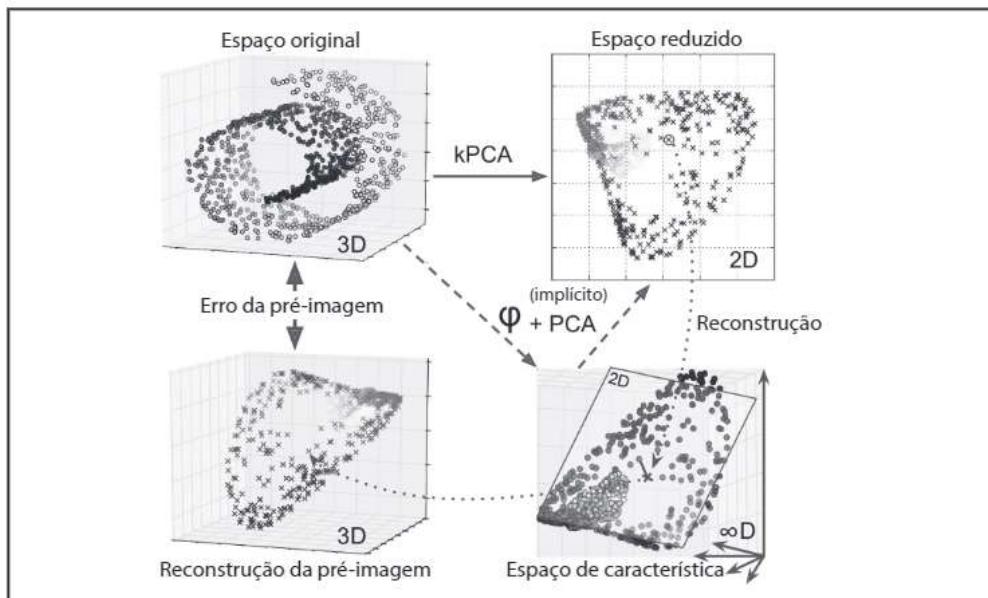


Figura 8-11. Kernel PCA e o erro de reconstrução da pré-imagem

Você pode estar se perguntando como realizar essa reconstrução. Uma solução seria treinar um modelo de regressão supervisionado com as instâncias projetadas como o conjunto de treinamento e as instâncias originais como alvos. O Scikit-Learn fará isso automaticamente se você configurar `fit_inverse_transform=True`, conforme mostrado no código a seguir:<sup>7</sup>

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                     fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```



Por padrão, `fit_inverse_transform=False` e `KernelPCA` não possuem o método `inverse_transform()`. Este método somente é criado quando você configura `fit_inverse_transform=True`.

Você pode, então, calcular o erro de reconstrução da pré-imagem:

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

Agora, você pode utilizar a grid search com validação cruzada para encontrar o kernel e os hiperparâmetros que minimizem esse erro de reconstrução da pré-imagem.

## LLE

*Locally Linear Embedding* (LLE)<sup>8</sup> (<https://goo.gl/iA9bns>) é outra técnica muito poderosa de *redução da dimensionalidade não linear* (NLDR, em inglês). É uma técnica de Manifold Learning que não depende de projeções como os algoritmos anteriores. Em poucas palavras, o LLE trabalha primeiro medindo como cada instância de treinamento se relaciona linearmente com suas vizinhas mais próximas (c.n., em inglês) e, em seguida, procura uma representação de dimensão inferior do conjunto de treinamento na qual esses relacionamentos locais são melhor preservados (mais detalhes em breve). Isso faz com que seja particularmente bom para desenrolar manifolds torcidos, especialmente quando não há muito ruído.

Por exemplo, o código a seguir utiliza a classe `LocallyLinearEmbedding` do Scikit-Learn para desenrolar o rolo suíço. O conjunto de dados 2D resultante é mostrado na Figura 8-12. Como você pode ver, o rolo suíço é completamente desenrolado e as distâncias

<sup>7</sup> O Scikit-Learn utiliza o algoritmo baseado na Regressão de Ridge do Kernel descrito em Gokhan H. Bakır, Jason Weston e Bernhard Scholkopf, “Learning to Find Pre-images” (<http://goo.gl/d0ydY6>) (Tubingen, Alemanha: Max Planck Institute for Biological Cybernetics, 2004).<sup>i</sup>

<sup>8</sup> “Nonlinear Dimensionality Reduction by Locally Linear Embedding”, S. Roweis, L. Saul (2000).

entre instâncias são bem preservadas localmente. No entanto, em uma escala maior, as distâncias não são preservadas: a parte esquerda desenrolada do rolo suíço é comprimida, enquanto a parte direita está esticada. Porém, o LLE fez um ótimo trabalho na modelagem do manifold.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

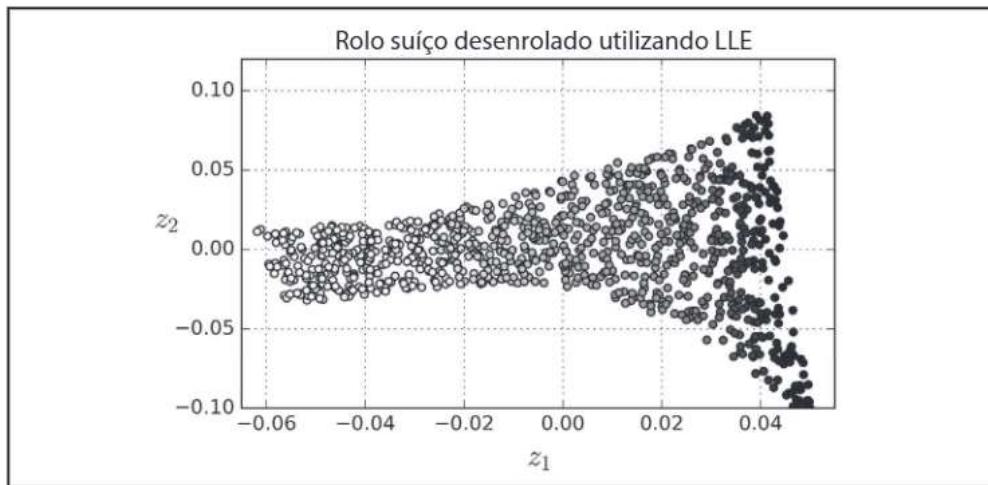


Figura 8-12. Rolo suíço desenrolado com a utilização do LLE

Veja como funciona o LLE: primeiro, para cada instância de treinamento  $\mathbf{x}^{(i)}$ , o algoritmo identifica seus  $k$ -nearest neighbors mais próximos (no código anterior  $k = 10$ ) e tenta reconstruir  $\mathbf{x}^{(i)}$  como uma função linear desses vizinhos. Mais especificamente, ele encontra os pesos  $w_{i,j}$  tais que a distância quadrada entre  $\mathbf{x}^{(i)}$  e  $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$  seja a menor possível, assumindo que  $w_{i,j} = 0$  se  $\mathbf{x}^{(j)}$  não seja um dos  $k$ -nearest neighbors de  $\mathbf{x}^{(i)}$ . Assim, o primeiro passo do LLE é o problema da otimização restrita descrito na Equação 8-4 em que  $\mathbf{W}$  é a matriz de peso contendo todos os pesos  $w_{i,j}$ . A segunda restrição simplesmente normaliza os pesos para cada instância de treinamento  $\mathbf{x}^{(i)}$ .

*Equação 8-4. LLE passo 1: modelagem linear de relacionamentos locais*

$$\begin{aligned} \widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2 \\ \text{sujeito a } \begin{cases} w_{i,j} = 0 & \text{se } \mathbf{x}^{(j)} \text{ não é um dos } k \text{ c.n. de } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{para } i = 1, 2, \dots, m \end{cases} \end{aligned}$$

Após este passo, a matriz de peso  $\mathbf{W}$  (que contém os pesos  $w_{i,j}$ ) codifica as relações lineares locais entre as instâncias de treinamento. Agora, o segundo passo consiste em mapear as instâncias de treinamento em um espaço  $d$ -dimensional (em que  $d < n$ ) preservando essas relações locais ao máximo. Se  $\mathbf{z}^{(i)}$  é a imagem de  $\mathbf{x}^{(i)}$  neste espaço  $d$ -dimensional, então queremos que a distância ao quadrado entre  $\mathbf{z}^{(i)}$  e  $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$  seja a menor possível. Essa ideia leva ao problema da otimização sem restrições descrito na Equação 8-5. Parece muito semelhante ao primeiro passo, mas, em vez de manter as instâncias corrigidas e encontrar os pesos ideais, estamos fazendo o inverso: mantendo os pesos fixos e encontrando a posição ideal das imagens das instâncias no espaço de dimensão inferior. Observe que  $\mathbf{Z}$  é a matriz que contém todo  $\mathbf{z}^{(i)}$ .

*Equação 8-5. LLE passo 2: reduz a dimensionalidade enquanto preserva os relacionamentos*

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{z}^{(i)} - \sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

As implementações do LLE do Scikit-Learn têm a seguinte complexidade computacional:  $O(m \log(m)n \log(k))$  para encontrar o *k-nearest neighbors*,  $O(mnk^3)$  para otimizar os pesos e  $O(dm^2)$  para a construção das representações de dimensão inferior. Infelizmente, o  $m^2$  no último termo faz com que este algoritmo escalone mal em conjuntos de dados muito grandes.

## Outras Técnicas de Redução da Dimensionalidade

Existem muitas outras técnicas de redução da dimensionalidade, várias das quais estão disponíveis no Scikit-Learn. Veja algumas das mais populares:

- *Escalonamento Multidimensional* (MDS, em inglês) reduz a dimensionalidade enquanto tenta preservar as distâncias entre as instâncias (veja a Figura 8-13).
- *Isomap* cria um grafo conectando cada instância aos seus *k-nearest neighbors*, então reduz a dimensionalidade enquanto tenta preservar as *distâncias geodésicas*<sup>9</sup> entre as instâncias.
- *t-Distributed Stochastic Neighbor Embedding* (t-SNE) reduz a dimensionalidade ao tentar manter as instâncias semelhantes próximas e instâncias de diferentes tipos separadas. É utilizado principalmente para a visualização, em particular para clusters de instâncias em um espaço de alta dimensão (por exemplo, para visualizar as imagens MNIST em 2D).

---

<sup>9</sup> A distância geodésica entre dois nós em um grafo é o número de nós no caminho mais curto entre esses nós.

- Análise Discriminante Linear (LDA, em inglês) é, na verdade, um algoritmo de classificação, mas ele assimila os eixos mais discriminativos entre as classes durante o treinamento de forma que esses eixos possam, então, ser utilizados na definição de um hiperplano no qual serão projetados os dados. O benefício é que a projeção manterá as classes o mais distantes possível, então o LDA é uma boa técnica para reduzir a dimensionalidade antes de executar outro algoritmo de classificação, como um classificador SVM.

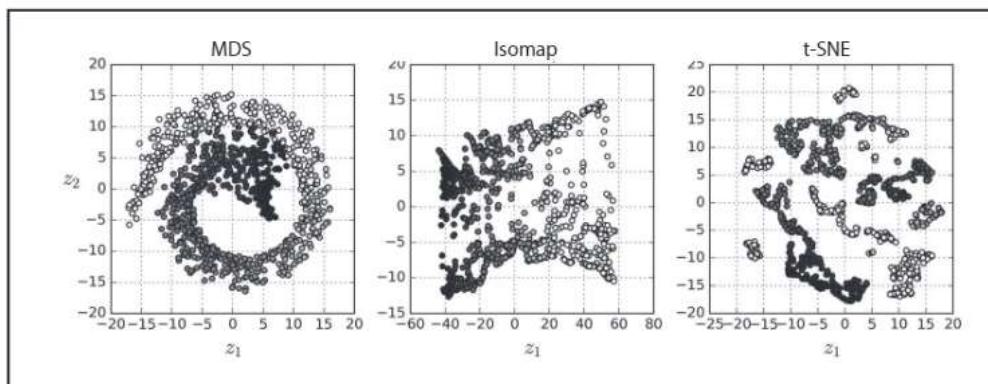


Figura 8-13. Reduzindo o rolo suíço para 2D com a utilização de várias técnicas

## Exercícios

1. Quais são as principais motivações para reduzir a dimensionalidade de um conjunto de dados? Quais são as principais desvantagens?
2. O que é a maldição da dimensionalidade?
3. Uma vez reduzida a dimensionalidade de um conjunto de dados, é possível reverter a operação? Em caso afirmativo, como? Se não, por quê?
4. O PCA pode ser utilizado para reduzir a dimensionalidade de um conjunto de dados altamente não linear?
5. Suponha que você execute um PCA em um conjunto de dados de mil dimensões, definindo a taxa de variância explicada em 95%. Quantas dimensões o conjunto de dados resultante terá?
6. Em que casos você usaria o PCA normal, PCA incremental, PCA randomizado ou Kernel PCA?
7. Como você pode avaliar o desempenho de um algoritmo de redução da dimensionalidade em seu conjunto de dados?
8. Faz algum sentido juntar dois algoritmos diferentes de redução da dimensionalidade?

9. Carregue o conjunto de dados MNIST (introduzido no Capítulo 3) e o divida em um conjunto de treinamento e um conjunto de teste (pegue as primeiras 60 mil instâncias para treinamento e os 10 mil restantes para teste). Treine um classificador Floresta Aleatória no conjunto de dados, cronometre quanto tempo ele demora, e avalie o modelo resultante no conjunto de teste. Em seguida, utilize o PCA para reduzir a dimensionalidade no conjunto de dados com uma taxa de variância explicada de 95%. Treine um novo classificador Floresta Aleatória no conjunto de dados reduzido e veja quanto tempo demorou. O treinamento foi muito mais rápido? Em seguida, avalie o classificador no conjunto de testes: como ele se compara ao classificador anterior?
10. Utilize o t-SNE para reduzir o conjunto de dados MNIST para duas dimensões e plotar o resultado com a utilização do Matplotlib. Você pode utilizar um gráfico de dispersão com dez cores diferentes para representar a classe de destino de cada imagem. Alternativamente, escreva dígitos coloridos no local de cada instância ou até mesmo plote versões reduzidas das próprias imagens dos dígitos (se plotar todos os dígitos sua visualização será muito confusa, então desenhe uma amostra aleatória ou plote uma instância apenas se nenhuma outra já tiver sido plotada a uma distância próxima). Você deve obter uma boa visualização com grupos de dígitos bem separados. Tente utilizar outros algoritmos de redução da dimensionalidade como o PCA, LLE ou MDS e compare as visualizações resultantes.

Soluções para estes exercícios estão disponíveis no Apêndice A.



## Parte II

# Redes Neurais e Aprendizado Profundo



## Capítulo 9

# Em Pleno Funcionamento com o TensorFlow

O *TensorFlow* é uma poderosa biblioteca de software para cálculo numérico de código aberto especialmente adequada e ajustada para o Aprendizado de Máquina em larga escala. Seu princípio básico é simples: primeiro, você define um grafo de cálculos para executar em Python (por exemplo, o da Figura 9-1) e, em seguida, o *TensorFlow* pega esse grafo e o executa de forma eficiente utilizando um código C++ otimizado.

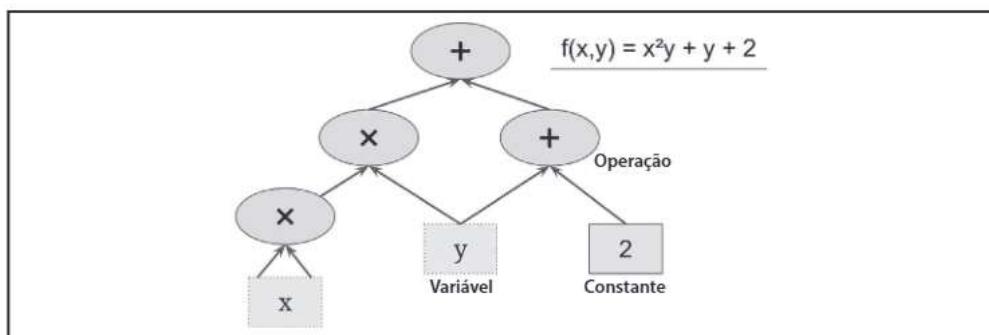


Figura 9-1. Um grafo de cálculo simples

Mas o mais importante é ser possível dividir o grafo em vários pedaços e executá-los em paralelo em várias CPUs ou GPUs (como mostrado na Figura 9-2). O *TensorFlow* também suporta a computação distribuída para que você possa treinar redes neurais colossais em gigantescos conjuntos de treinamento dividindo os cálculos por centenas de servidores em um período de tempo razoável (veja o Capítulo 12). O *TensorFlow* pode treinar uma rede com milhões de parâmetros em um conjunto de treinamento composto por bilhões de instâncias com milhões de características cada. Isso não deve ser uma surpresa, já que o *TensorFlow* foi desenvolvido pela equipe do Google Brain e alimenta muitos dos serviços em grande escala do Google, como Google Cloud Speech, Google Fotos e o Google Search.

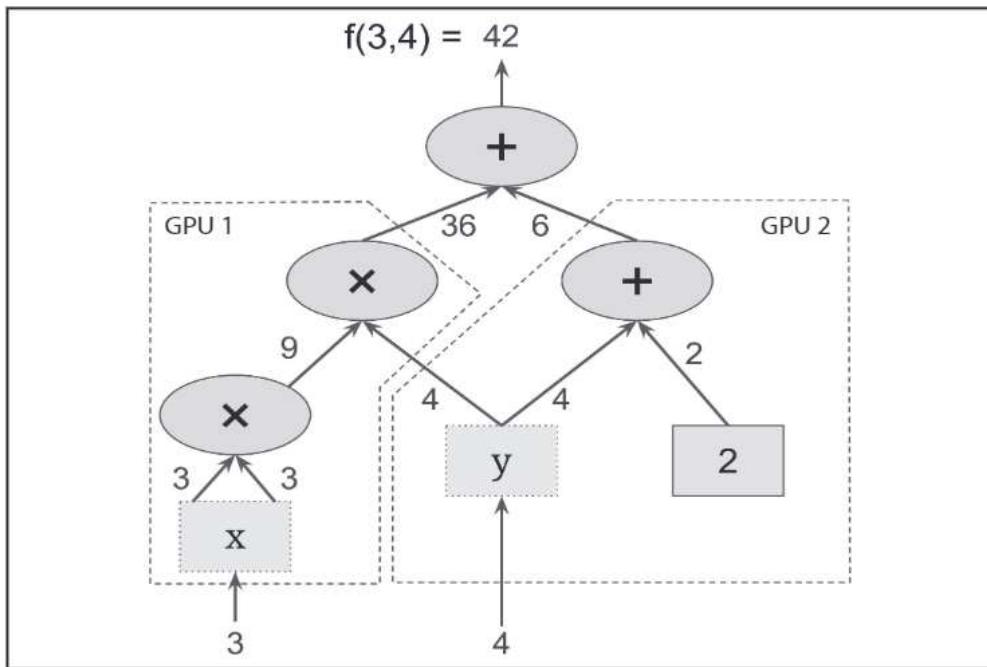


Figura 9-2. Computação paralela em várias CPUs/GPUs/servidores

Quando o código do TensorFlow foi aberto em novembro de 2015, já existiam muitas bibliotecas populares de código aberto para o Aprendizado Profundo (a Tabela 9-1 lista algumas), e, para ser sincero, a maioria dos recursos do TensorFlow já existiam em uma ou outra biblioteca. No entanto, seu design limpo, a escalabilidade, a flexibilidade<sup>1</sup> e uma ótima documentação (sem mencionar o nome do Google) rapidamente o alçaram ao topo da lista. Em suma, o TensorFlow foi projetado para ser flexível, escalável e pronto para produção, e os frameworks existentes, sem dúvida, atingiram apenas dois dos três itens citados. Estes são alguns destaques do TensorFlow:

- Ele roda não apenas no Windows, Linux e MacOS, mas também em dispositivos móveis, incluindo iOS e Android;
- Ele fornece uma API Python muito simples chamada *TF.Learn*<sup>2</sup> (`tensorflow.contrib.learn`) compatível com o Scikit-Learn. Como você verá, com apenas algumas linhas de código ele poderá ser utilizado para treinar vários tipos de redes neurais. Anteriormente, era um projeto independente chamado *Scikit Flow* (ou *skflow*);

1 O TensorFlow não se limita às redes neurais ou mesmo ao Aprendizado de Máquina; você poderia executar simulações de física quântica se quisesse.

2 Não confundir com a biblioteca TFLearn, que é um projeto independente.

- Ele também fornece outra API simples chamada *TF-slim* (`tensorflow.contrib.slim`) para simplificar a construção, o treinamento e a avaliação de redes neurais;
- Várias outras APIs de alto nível foram construídas independentemente com base no TensorFlow, como o *Keras* (<http://keras.io>) (disponível em `tensorflow.contrib.keras`) ou o *Pretty Tensor* (<https://github.com/google/prettytensor/>);
- Sua principal API do Python oferece muito mais flexibilidade (ao custo da maior complexidade) para criar vários tipos de cálculos, incluindo qualquer arquitetura de rede neural em que você possa pensar;
- Ele inclui implementações C++ altamente eficientes de muitas operações do AM, particularmente aquelas necessárias para construir redes neurais. Há também uma API C++ para definir suas próprias operações de alto desempenho;
- Ele fornece vários nós de otimização avançados para procurar por parâmetros que minimizam uma função de custo. Usá-los é muito fácil, uma vez que o TensorFlow cuida automaticamente do cálculo dos gradientes das funções que você define. Isto é chamado *diferenciação automática* (ou *autodiff*).
- Ele também vem com uma grande ferramenta de visualização chamada *TensorBoard* que permite navegar pelo grafo de computação, ver curvas de aprendizado e muito mais;
- O Google também lançou um serviço em nuvem para rodar os grafos do TensorFlow (<https://cloud.google.com/ml>);
- Por último, mas não menos importante, ele tem uma equipe dedicada de desenvolvedores apaixonados e colaborativos, e uma comunidade crescente, que contribuem para melhorá-lo. É um dos projetos de código aberto mais populares no GitHub, e cada vez mais estão sendo construídos grandes projetos a partir dele (para exemplos, confira as páginas de recursos em <https://www.tensorflow.org/>, ou <https://github.com/jtoy/awesome-tensorflow>). Para questões técnicas, você deve usar <http://stackoverflow.com/> e marcar sua pergunta com “`tensorflow`”. No GitHub, você pode arquivar erros e solicitações de recursos. Para discussões gerais, junte-se ao grupo do Google (<http://goo.gl/N7kRF9>).

Neste capítulo, analisaremos os conceitos básicos do TensorFlow, desde a instalação até a criação, execução, gravação e visualização de grafos de cálculos simples. É importante o domínio desses princípios antes de construir sua primeira rede neural (o que faremos no próximo capítulo).

*Tabela 9-1. Bibliotecas de Aprendizado Profundo de código aberto (não é uma lista completa)*

Biblioteca	API	Plataformas	Iniciado por	Ano
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Y. Jia, UC Berkeley (BVLC)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	A. Gibson, J. Patterson	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014
MXNet	Python, C++, others	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	Universidade de Montreal	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	R. Collobert, K. Kavukcuoglu, C. Farabet	2002

## Instalação

Vamos começar! Supondo que, seguindo as instruções de instalação no Capítulo 2, você instalou o Jupyter e o Scikit-Learn, você pode simplesmente utilizar o pip para instalar o TensorFlow. Criou-se um ambiente isolado utilizando o virtualenv, primeiro você precisa ativá-lo:

```
$ cd $ML_PATH          # O diretório de seu AM (por ex., $HOME/ml)
$ source env/bin/activate
```

Em seguida, instale o TensorFlow (se você não estiver utilizando um virtualenv, precisará de direitos de administrador ou adicionar a opção `--user`):

```
$ pip3 install --upgrade tensorflow
```



Para o suporte a GPU, você precisa instalar o `tensorflow-gpu` em vez do `tensorflow`. Consulte o Capítulo 12 para obter mais detalhes.

Para testar sua instalação, digite o comando abaixo, que deve exibir a versão do TensorFlow que você instalou.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)'
1.3.0
```

## Criando Seu Primeiro Grafo e o Executando em uma Sessão

O código a seguir cria o grafo representado na Figura 9-1:

```
import tensorflow as tf

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

E isso é tudo por enquanto! O mais importante a se entender é que este código na verdade não realiza cálculo algum, embora pareça fazê-lo (especialmente a última linha). Ele apenas cria um grafo de cálculo. Na verdade, mesmo as variáveis ainda não foram inicializadas. Para avaliar este grafo, você precisa abrir uma *sessão* do TensorFlow e utilizá-la para inicializar as variáveis e avaliar *f*. Uma sessão do TensorFlow trata de colocar as operações em *dispositivos* como CPUs e GPUs e executá-las, e contém todos os valores das variáveis.<sup>3</sup> O código a seguir cria uma sessão, inicializa as variáveis, as avalia, e então *f* encerra a sessão (o que libera recursos):

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Ter que repetir *sess.run()* todas as vezes é um pouco complicado, mas felizmente há uma forma melhor:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

A sessão é configurada como padrão dentro do bloco *with*. Chamar *x.initializer.run()* é equivalente a chamar *tf.get\_default\_session().run(x.initializer)*, assim como *f.eval()* é equivalente a chamar *tf.get\_default\_session().run(f)*, facilitando a leitura do código. Além do mais, a sessão é encerrada automaticamente ao final do bloco.

Você pode utilizar a função *global\_variables\_initializer()* em vez de executar manualmente o inicializador para cada variável. Observe que ela não executa a inicialização imediatamente, mas cria um nó no grafo que inicializará todas as variáveis quando esta for executada:

---

<sup>3</sup> No TensorFlow distribuído os valores das variáveis são armazenados nos servidores em vez de na sessão, como veremos no Capítulo 12.

```

init = tf.global_variables_initializer() # prepara um nó init

with tf.Session() as sess:
    init.run() # na verdade, inicializa todas as variáveis
    result = f.eval()

```

Dentro do Jupyter, ou dentro de um *shell* do Python, você pode preferir criar uma `InteractiveSession`. Sua única diferença em relação a uma `Session` regular é que, quando é criada, ela automaticamente se configura como uma sessão padrão, então você não precisa de um bloco `with` (mas é necessário encerrar a sessão manualmente ao terminar):

```

>>> sess = tf.InteractiveSession()
>>> init.run()
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()

```

Um programa TensorFlow normalmente é dividido em duas partes: a primeira cria um gráfico de cálculo (isto é chamado de *fase de construção*), e a segunda o executa (esta é a *fase de execução*). A fase de construção geralmente constrói um grafo de cálculo que representa o modelo AM e os cálculos necessários para treiná-lo. A fase de execução geralmente executa um loop que avalia repetidamente um passo do treinamento (por exemplo, um passo por minilote), melhorando gradualmente os parâmetros do modelo. Veremos um exemplo em breve.

## Gerenciando Grafos

Qualquer nó que você crie é adicionado automaticamente ao grafo padrão:

```

>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True

```

Na maioria dos casos isso é bom, mas, às vezes, pode ser que você queira gerenciar vários grafos independentes, o que pode ser feito por meio da criação de um novo `Graph` tornando-o temporariamente o grafo padrão dentro de um bloco `with`:

```

>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False

```



No Jupyter (ou em uma shell do Python), é comum executar os mesmos comandos mais de uma vez durante a fase de experimentação. Como resultado, você pode acabar com um grafo padrão contendo muitos nós duplicados. Uma solução é reiniciar o kernel Jupyter (ou o shell do Python), mas uma solução mais conveniente é resetar o grafo padrão executando `tf.reset_default_graph()`.

## Ciclo de Vida de um Valor do Nô

Quando você avalia um nó, o TensorFlow determina automaticamente o conjunto de nós dependentes e os avalia primeiro. Por exemplo, considere o código a seguir:

```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

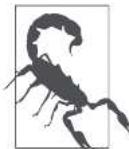
with tf.Session() as sess:
    print(y.eval()) # 10
    print(z.eval()) # 15
```

Primeiro, este código define um grafo muito simples. Em seguida, ele inicia uma sessão e executa o grafo para avaliar `y`: o TensorFlow automaticamente detecta que `y` depende de `x`, que depende de `w`, então ele primeiro avalia `w`, então `x`, depois `y`, e retorna o valor de `y`. Finalmente, o código executa o grafo para avaliar `z`. Mais uma vez, o TensorFlow detecta que ele deve primeiro avaliar `w` e `x`. É importante notar que ele *não* reutilizará o resultado das avaliações prévias de `w` e `x`. Em suma, o código anterior avalia duas vezes `w` e `x`.

Todos os valores do nó são descartados entre as execuções dos grafos, exceto os valores das variáveis que são mantidas na sessão pela execução dos grafos (queues e readers também mantêm algum estado, como veremos no Capítulo 12). Uma variável começa sua vida quando seu inicializador é executado e termina quando a sessão é encerrada.

Se você quiser avaliar `y` e `z` eficientemente, sem avaliar `w` e `x` duas vezes como no código anterior, deve pedir ao TensorFlow para avaliar tanto `y` quanto `z` em apenas uma execução do grafo, como mostrado no código a seguir:

```
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
    print(y_val) # 10
    print(z_val) # 15
```



Em um processo único do TensorFlow, as múltiplas sessões não compartilham de nenhum estado, mesmo se reutilizarem grafos similares (cada sessão teria sua própria cópia de cada variável). No TensorFlow distribuído (ver Capítulo 12), o estado da variável é armazenado nos servidores, não nas sessões, então as múltiplas sessões podem compartilhar as mesmas variáveis.

## Regressão Linear com o TensorFlow

As operações do TensorFlow (também chamadas de *ops*) podem pegar quaisquer números de entradas e produzir quaisquer números de saídas. Por exemplo, as operações de adição e multiplicação tomam duas entradas e produzem uma saída. Constantes e variáveis não tomam nenhuma entrada (elas são chamadas de *source ops*). As entradas e saídas são arrays multidimensionais chamados *tensores* (daí o nome “fluxo do tensor”, tradução literal de *tensor flow*). Assim como os arrays NumPy, os tensores têm um tipo e uma forma. Na verdade, os tensores da API do Python são simplesmente representados por arrays NumPy. Eles geralmente contêm floats, mas você também pode utilizá-los para carregar strings (arrays de bytes arbitrários).

Nos exemplos vistos, os tensores apenas continham um único valor escalar, mas você pode, obviamente, executar cálculos em arrays de qualquer forma. Por exemplo, o código a seguir manipula arrays 2D para executar a Regressão Linear no conjunto de dados imobiliários da Califórnia (introduzido no Capítulo 2). Começa pela busca do conjunto de dados; então adiciona uma característica de entrada de viés extra ( $x_0 = 1$ ) a todas as instâncias de treinamento utilizando o NumPy para que possa ser executado imediatamente; então cria dois nós constantes do TensorFlow, `X` e `y`, para segurar estes dados e os alvos,<sup>4</sup> e utiliza algumas das operações da matriz fornecidas pelo TensorFlow para definir `theta`. Essas funções de matriz — `transpose()`, `matmul()` e `matrix_inverse()` — são autoexplicativas, mas, como de costume, elas não realizam nenhum cálculo imediatamente; em vez disso, elas criam nós no grafo que irão executá-las quando este for executado. Você deve reconhecer que a definição de `theta` corresponde ao Método dos Mínimos Quadrados ( $\theta = (X^T \cdot X)^{-1} \cdot X^T \cdot y$ ; veja o Capítulo 4). Finalmente, o código cria uma sessão e a utiliza para avaliar `theta`.

---

<sup>4</sup> Note que `housing.target` é um array 1D, mas precisamos remodelá-lo para um vetor de coluna para calcular `theta`. Recorde que a função `reshape()` do NumPy aceita `-1` (significando “unspecified”) para uma das dimensões: essa dimensão será calculada com base no comprimento do array e nas demais dimensões.

```

import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()

```

O principal benefício deste código em relação ao Método dos Mínimos Quadrados diretamente no NumPy é que o TensorFlow executará isso automaticamente no seu cartão GPU, se você tiver um (desde que tenha instalado o TensorFlow com suporte a GPU, é claro, veja o Capítulo 12 para mais detalhes).

## Implementando o Gradiente Descendente

Tentaremos usar o Gradiente Descendente em Lote (introduzido no Capítulo 4) em vez do Método dos Mínimos Quadrados. Primeiro, faremos isso calculando manualmente os gradientes, então utilizaremos o recurso autodiff para permitir que o TensorFlow calcule os gradientes automaticamente e, finalmente, utilizaremos alguns otimizadores inusitados do TensorFlow.



Ao utilizar o Gradiente Descendente, lembre-se que é importante primeiro normalizar os vetores das características de entrada, ou o treinamento pode ficar muito mais lento. Você pode fazer isso utilizando o TensorFlow, NumPy, StandardScaler do Scikit-Learn, ou qualquer outra solução que você preferir. O código a seguir assume que esta normalização já foi feita.

## Calculando Manualmente os Gradiêntes

O código a seguir deve ser bem autoexplicativo, exceto por alguns novos elementos:

- A função `random_uniform()` cria um nó no grafo que gerará um tensor contendo valores aleatórios, dado sua forma e faixa de valor, bem parecido com a função `rand()` do NumPy;
- A função `assign()` cria um nó que atribuirá um novo valor a uma variável. Neste caso, ela implementa o passo  $\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$  do Gradiente Descendente em Lote;

- O loop principal executa o passo de treinamento repetidamente (`n_epochs` vezes), e a cada 100 iterações imprime o Erro Médio Quadrado atual (`mse`). Você deve ver o MSE diminuir a cada iteração.

```

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

    best_theta = theta.eval()

```

## Utilizando o autodiff

O código anterior funciona bem, mas requer uma derivação matemática dos gradientes da função de custo (MSE). No caso da Regressão Linear, é razoavelmente fácil, mas provocaria muitas dores de cabeça se você tivesse que fazer isso com redes neurais profundas: seria tedioso e propenso a erros. Você poderia utilizar a *diferenciação simbólica* para encontrar automaticamente as equações para as derivadas parciais, mas o código resultante não seria necessariamente muito eficiente.

Para entender o porquê, considere a função  $f(x) = \exp(\exp(\exp(x)))$ . Se você conhece cálculo, pode descobrir sua derivada  $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$ . Se você codificar separadamente  $f(x)$  e  $f'(x)$  exatamente como elas aparecem, seu código não será tão eficiente como poderia ser. Uma solução mais eficiente seria escrever uma função que calcule primeiro  $\exp(x)$ , depois  $\exp(\exp(x))$ , depois  $\exp(\exp(\exp(x)))$ , e retorne todas as três. Isto lhe dá diretamente  $f(x)$  (o terceiro termo), e, se você precisar da derivada, pode apenas multiplicar os três termos e pronto. Com uma naive approach, você precisaria chamar a função `exp` nove vezes para o cálculo de  $f(x)$  e  $f'(x)$ . Com esta abordagem, você precisa chamá-la apenas três vezes.

Isto piora quando sua função for definida por algum código arbitrário. Você consegue encontrar a equação (ou o código) para calcular as derivadas parciais da seguinte função? Dica: nem tente.

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

Felizmente, o recurso autodiff do TensorFlow vem ao resgate: ele pode calcular automaticamente e eficientemente os gradientes para você. Simplesmente substitua a linha `gradients = ...` no código do Gradiente Descendente na sessão anterior pela seguinte linha e o código continuará a funcionar bem:

```
gradients = tf.gradients(mse, [theta])[0]
```

A função `gradients()` pega uma op (neste caso `mse`) e uma lista de variáveis (neste caso apenas `theta`), e cria uma lista de ops (uma por variável) para calcular os gradientes de op com relação a cada variável. Assim, o nó `gradients` calculará o vetor gradiente do MSE com relação à `theta`.

Existem quatro abordagens principais para o cálculo automático de gradientes. Elas estão resumidas na Tabela 9-2. O TensorFlow utiliza *reverse-mode autodiff*, que é perfeito (eficiente e preciso) quando existem muitas entradas e poucas saídas, como geralmente é o caso nas redes neurais. Ele calcula todas as derivadas parciais das saídas em relação a todas as entradas em apenas  $n_{outputs} + 1$  travessias em grafo.

*Tabela 9-2. Principais soluções para calcular gradientes automaticamente*

Técnica	Nº de travessias em grafo para calcular todos os gradientes	Acurácia	Superta código arbitrário	Comentário
Diferenciação Numérica	$n_{inputs} + 1$	Baixa	Sim	Implementação trivial
Diferenciação Simbólica	N/A	Alta	Não	Constrói um grafo muito diferente
Modo Avançado autodiff	$n_{inputs}$	Alta	Sim	Utiliza <i>dual numbers</i>
Modo reverso autodiff	$n_{outputs} + 1$	Alto	Sim	Implementado pelo TensorFlow

Se você se interessa em saber como funciona esta mágica, verifique o Apêndice D.

## Utilizando um Otimizador

O TensorFlow calcula os gradientes para você. Mas fica ainda mais fácil: ele também fornece uma série de otimizadores inusitados, incluindo um otimizador de Gradiente Descendente. Você pode simplesmente substituir as linhas anteriores `gradients = ...` e `training_op = ...` pelo código a seguir e, mais uma vez, tudo funcionará bem:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

Se você deseja utilizar um tipo diferente de otimizador, basta alterar uma linha. Por exemplo, utilize um otimizador momentum (que, muitas vezes, converge bem mais rápido do que o Gradiente Descendente, veja o Capítulo 11) definindo o otimizador assim:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                      momentum=0.9)
```

## Fornecendo Dados ao Algoritmo de Treinamento

Tentaremos modificar o código anterior para implementar o Gradiente Descendente em Minilote. Para isso, precisamos de uma forma de substituir `X` e `y` em cada iteração com o próximo minilote. A maneira mais simples de fazer isso é utilizar nós placeholders. Esses nós são especiais porque eles, na verdade, não executam nenhum cálculo, apenas exibem os dados que você os manda exibir em tempo de execução. Eles geralmente são utilizados durante o treinamento para passar os dados de treinamento para o TensorFlow. Se você não especificar um valor para um placeholder em tempo de execução, terá uma exceção.

Para criar um nó placeholder, você precisa chamar a função `placeholder()` e especificar o tipo de saída dos dados do tensor. Como opção, você também pode especificar sua forma se quiser reforçá-la. Especificar `None` para uma dimensão significa “qualquer tamanho”. Por exemplo, o código a seguir cria um nó placeholder `A` e também um nó `B = A + 5`. Quando avaliamos `B`, passamos um método `feed_dict` para o método `eval()`, que especifica o valor de `A`. Observe que `A` deve ter classificação 2 (ou seja, ele deve ser bidimensional) e deve ter três colunas (ou então uma exceção é levantada), mas pode ter qualquer número de linhas.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
 [12. 13. 14.]]
```



Você pode fornecer o resultado de *qualquer* operações, não apenas placeholders. Neste caso, o TensorFlow não tenta avaliar essas operações; ele utiliza os valores que você fornece.

Para implementar o Gradiente Descendente em Minilote, só precisamos ajustar ligeiramente o código existente. Primeiro, altere a definição de `X` e `y` na fase de construção para torná-los nós placeholders:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Em seguida, defina o tamanho do lote e calcule o número total de lotes:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

Finalmente, na fase de execução, pegue os minilotes um a um e forneça o valor de `X` e `y` por meio do parâmetro `feed_dict` ao avaliar um nó que dependa deles.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # carrega os dados do disco
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
best_theta = theta.eval()
```



Não precisamos passar o valor de `X` e `y` ao avaliar `theta` já que ela não depende de nenhum deles.

## Salvando e Restaurando Modelos

Depois de ter treinado seu modelo, você deve salvar seus parâmetros no disco para que possa voltar a eles sempre que quiser utilizá-los em outro programa, compará-los com outros modelos e assim por diante. Além disso, é importante salvar pontos de verificação em intervalos regulares durante o treinamento, de modo que, se o seu computador travar durante o treinamento, você poderá continuar a partir do último ponto de verificação em vez de começar de novo.

É muito fácil salvar e restaurar um modelo no TensorFlow. Basta criar um nó `Saver` no final da fase de construção (após a criação de todos os nós de variáveis); em seguida, chamar seu método `save()` sempre que quiser salvar o modelo, passando a sessão e o caminho do arquivo do ponto de verificação na fase de execução:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")
        sess.run(training_op)

        best_theta = theta.eval()
        save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Restaurar um modelo é fácil: como anteriormente, você cria um `Saver` ao final da fase de construção, mas, utilizando o nó `init` no início da fase de execução, você chama o método `restore()` do objeto `Saver` em vez de inicializar as variáveis:

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
[...]
```

Por padrão, um `Saver` salva e restaura todas as variáveis com seu próprio nome, mas, se você precisar de mais controle, pode especificar quais as variáveis a serem salvadas ou restauradas e quais nomes utilizar. Por exemplo, o `Saver` a seguir vai salvar ou restaurar somente a variável `theta` com o nome `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

Por padrão, o método `save()` também salva a estrutura do grafo em um segundo arquivo com o mesmo nome mais a extensão `.meta`. Você pode carregar esta estrutura do grafo utilizando `tf.train.import_meta_graph()`, adicionando o grafo ao grafo padrão e retornando uma instância `Saver` que você pode utilizar para restaurar o estado do grafo (ou seja, os valores das variáveis):

```
saver = tf.train.import_meta_graph("/tmp/my_model_final.ckpt.meta")

with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
[...]
```

Isso permite que você restaure completamente um modelo salvo, incluindo a estrutura do grafo e os valores das variáveis sem ter que procurar o código que o construiu.

## Visualização do Grafo e Curvas de Treinamento com o TensorBoard

Agora temos um grafo de cálculo que treina um modelo de Regressão Linear utilizando o Gradiente Descendente em Minilotes, e estamos salvando pontos de verificação em intervalos regulares. Soa sofisticado, não? No entanto, ainda confiamos na função `print()` para visualizar o progresso durante o treinamento. Existe um jeito melhor: digite `TensorBoard`, que exibirá visualizações interativas agradáveis dessas estatísticas em seu navegador (por exemplo, curvas de aprendizado) se você fornecer algumas estatísticas de treinamento. Você também pode fornecer a definição do grafo e ele lhe dará uma ótima interface para navegação, o que é muito útil para identificar erros no grafo, encontrar pontos de estrangulamento e assim por diante.

O primeiro passo é ajustar um pouco seu programa para que ele escreva a definição do grafo e algumas estatísticas de treinamento — por exemplo, o erro de treinamento (MSE) — para um diretório de logs que o `TensorBoard` lerá. Você precisa utilizar um diretório de log diferente a cada vez que executa o programa senão o `TensorBoard` mesclará estatísticas a cada execução diferente, o que bagunçará as visualizações. A solução mais simples é incluir um timestamp no nome do diretório do log. Adicione o código a seguir no início do programa:

```
from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}-{}{}".format(root_logdir, now)
```

A seguir, adicione o código abaixo no final da fase de construção:

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

A primeira linha cria um nó no grafo que avaliará o valor MSE e o escreverá em uma string de log binário chamada de `summary`, compatível com o `TensorBoard`. A segunda linha cria um `FileWriter` que utilizaremos para escrever resumos nos arquivos de log no diretório de logs. O primeiro parâmetro indica o caminho do diretório de log (neste caso, algo como `tf_logs/run-20160906091959/`, relativo ao diretório atual). O segundo parâmetro (opcional) é o grafo que você visualizará. Na criação, o `FileWriter` cria o diretório de log caso ele ainda não exista (e seus diretórios pais, se necessário), e escreve a definição do grafo em um arquivo de log binário chamado de `events file`.

Depois, durante o treinamento, você precisa atualizar a fase de execução para avaliar regularmente o nó `mse_summary` (por exemplo, a cada 10 minilotes). Isso produzirá um

resumo do que você pode, então, escrever no arquivo de eventos usando o `file_writer`. Veja o código atualizado:

```
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
    [...]
```

Evite registrar estatísticas de treinamento em cada etapa de treinamento pois isso reduzirá sua velocidade significativamente.



Finalmente, encerre o `FileWriter` ao final do programa:

```
file_writer.close()
```

Agora, execute-o: ele criará o diretório de log e escreverá um arquivo de eventos nele contendo a definição do grafo e os valores MSE. Abra uma shell e vá até seu diretório de trabalho e digite `ls -l tf_logs/run*` para listar o conteúdo do diretório de logs:

```
$ cd $ML_PATH          # Seu diretório do AM (por ex., $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep 6 11:10 events.out.tfevents.1472553182.mymac
```

Se você executar o programa uma segunda vez, você deve encontrar um segundo diretório no diretório `tf_logs/`:

```
$ ls -l tf_logs/
total 0
drwxr-xr-x 3 ageron staff 102 Sep 6 10:07 run-20160906091959
drwxr-xr-x 3 ageron staff 102 Sep 6 10:22 run-20160906092202
```

Muito bem! Agora é hora de abrir o servidor do TensorBoard. Você precisa ativar o seu ambiente virtualenv caso tenha criado um, então iniciar o servidor executando o comando `tensor_board` apontando para o diretório raiz do log, e assim iniciando o servidor web TensorBoard pela porta 6006 (que é “goog” escrito de cabeça para baixo):

```
$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard on port 6006
(You can navigate to http://0.0.0.0:6006)
```

A seguir, abra um navegador e digite `http://0.0.0.0:6006/` (ou `http://localhost:6006/`). Bem vindo ao TensorBoard! Na aba *Events*, você deve ver o MSE à direita. Ao clicar nele, verá uma plotagem do MSE durante o treinamento para ambas as execuções (Figura 9-3). Você

pode marcar ou desmarcar as execuções que queira visualizar, dar um zoom in ou out, passar o cursor sobre a curva para obter detalhes, e assim por diante.

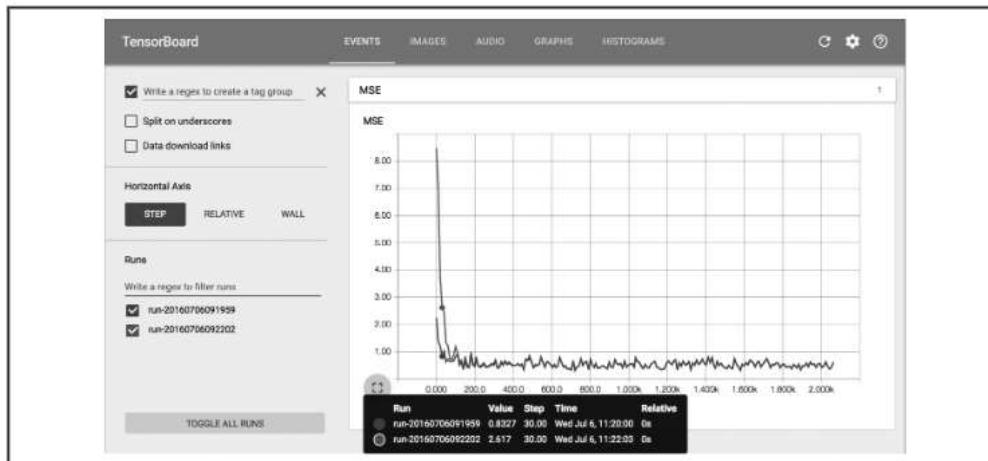


Figura 9-3. Visualizando estatísticas de treinamento utilizando o TensorBoard

Agora, clique na aba Graphs. Você deve ver o grafo mostrado na Figura 9-4.

Para reduzir a confusão, os nós que têm muitas *bordas* (ou seja, conexões para outros nós) são separados em uma área auxiliar à direita (você pode mover um nó para frente e para trás entre o grafo principal e a área auxiliar clicando nele com o botão direito do mouse). Algumas partes do grafo também são colapsadas por padrão. Por exemplo, tente passar o mouse pelo nó `gradients`, clique no ícone  $\oplus$  para expandir este subgrafo. A seguir, neste subgrafo, tente expandir o subgrafo `mse_grad`.

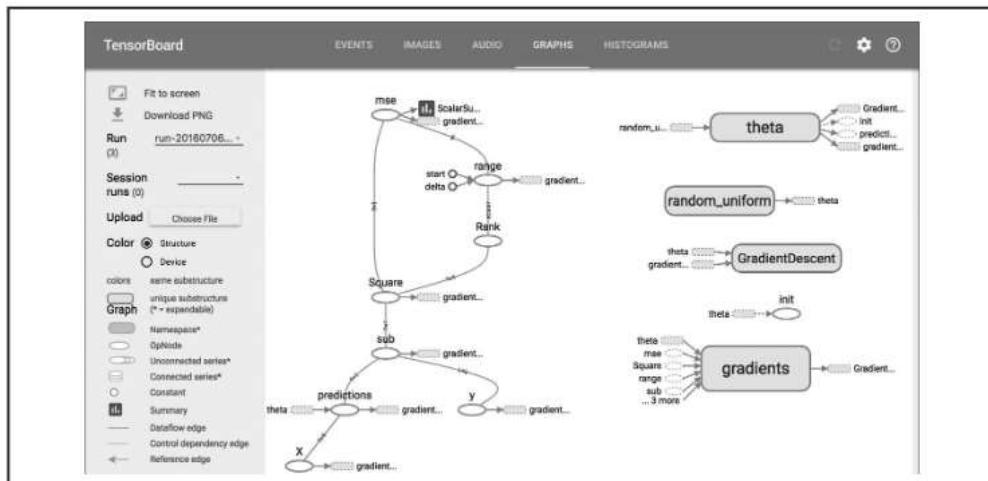


Figura 9-4. Visualizando o grafo com a utilização do TensorBoard



Se quiser dar uma olhada no grafo diretamente dentro do Jupyter, utilize a função `show_graph()` disponível no notebook para este capítulo. Ela foi escrita originalmente por A. Mordvintsev em seu grande tutorial de notebook (<http://goo.gl/EtCWUc>). Outra opção é instalar a ferramenta de debug do TensorFlow de E. Jang (<https://github.com/ericjang/tdb>) que inclui uma extensão do Jupyter para a visualização de grafos (e mais).

## Escopos do Nome

O grafo pode facilmente ficar confuso com os milhares de nós ao lidar com modelos mais complexos como as redes neurais. Para evitar isso, crie *escopos do nome* para agrupar nós relacionados. Por exemplo, vamos modificar o código anterior para definir o ops `error` e `mse` dentro de um escopo do nome chamado “`loss`”:

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

O nome de cada op definido dentro do escopo agora é prefixado com “`loss/`”:

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

Os nós `mse` e `error` aparecem agora dentro do escopo de nomes `loss` no TensorBoard, que aparece colapsado por padrão (Figura 9-5).

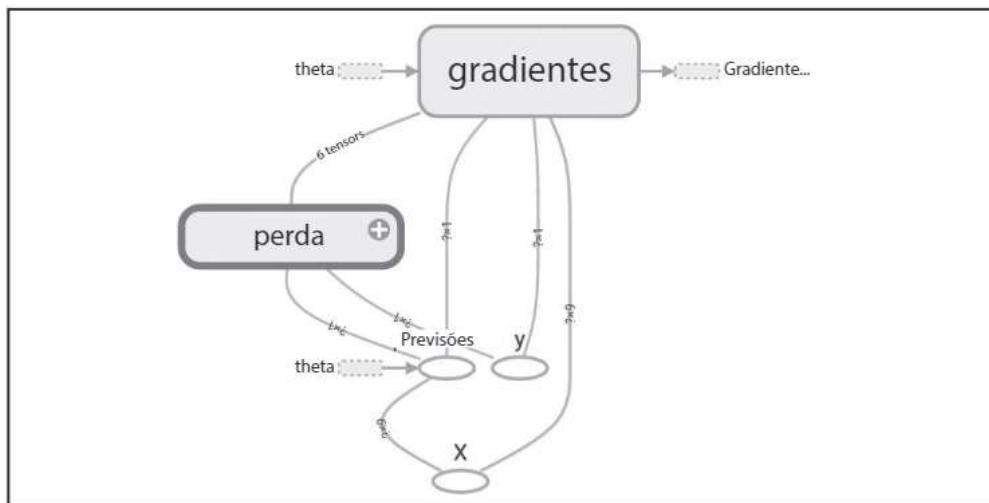


Figura 9-5. Um escopo do nome colapsado no TensorBoard

## Modularidade

Suponha que você queira criar um grafo que adicione a saída de duas *unidades lineares retificadas* (ReLU). Um ReLU calcula uma função linear das entradas e exibe o resultado se for positivo, e 0 caso contrário, como mostrado na Equação 9-1.

*Equação 9-1. Unidade linear retificada*

$$h_{w,b}(X) = \max(X \cdot w + b, 0)$$

O código a seguir cumpre esta função, mas é bem repetitivo:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```

Além de ser de difícil manutenção, este código repetitivo é propenso a erros (na verdade, este código contém um erro de cópia e cola, você viu?). Seria ainda pior se você quisesse adicionar mais algumas ReLUs. Felizmente, o TensorFlow permite que você evite repetições: simplesmente criando uma função para desenvolver uma ReLU. O código a seguir cria cinco ReLUs e exibe sua soma (observe que `add_n()` cria uma operação que calculará a soma de uma lista de tensores):

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Observe que, quando você cria um nó, o TensorFlow verifica se seu nome já existe e, se existir, ele anexa um sublinhado seguido de um índice para tornar o nome exclusivo. Portanto, a primeira ReLU contém nós denominados “weights”, “bias”, “z” e “relu”

(mais muitos outros nós com seu nome padrão, como “`MatMul`”); a segunda ReLU contém nós denominados “`weights_1`”, “`bias_1`”, e assim por diante; a terceira ReLU contém nós denominados “`weights_2`”, “`bias_2`”, e assim por diante. O TensorBoard identifica estas séries e as recolhe para reduzir a bagunça (como você pode ver na Figura 9-6).

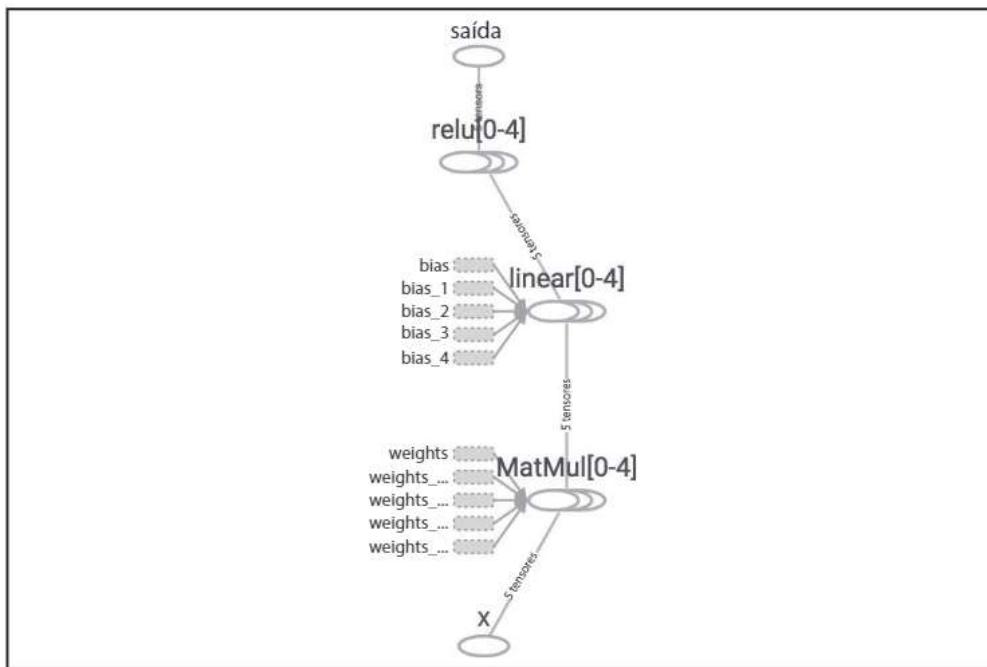


Figura 9-6. Séries de nós colapsados

Utilizando escopos do nome, o grafo fica muito mais limpo. Basta mover todo o conteúdo da função `relu()` dentro de um escopo do nome. A Figura 9-7 mostra o grafo resultante. Observe que o TensorFlow também dá nomes exclusivos aos escopos do nome anexando `_1`, `_2`, e assim por diante.

```

def relu(X):
    with tf.name_scope("relu"):
        [...]

```

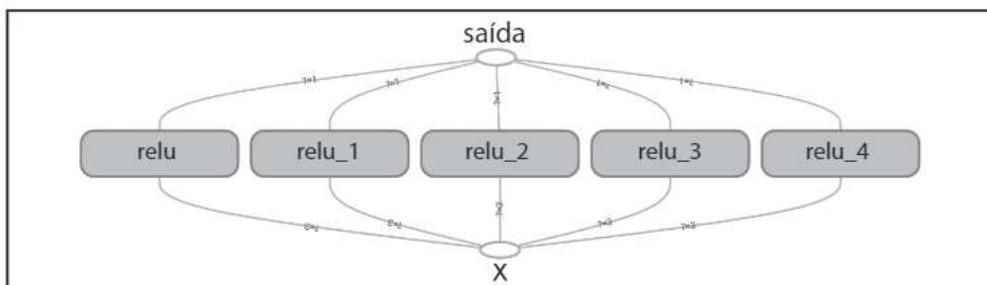


Figura 9-7. Um grafo mais claro utilizando unidades com escopos do nome

## Compartilhando Variáveis

Se você quiser compartilhar uma variável entre vários componentes do seu grafo, uma opção simples é criá-la primeiro e depois passá-la como um parâmetro para as funções que a necessitam. Por exemplo, suponha que você deseja controlar o limiar da ReLU (atualmente codificado para 0) utilizando uma variável `threshold` compartilhada para todas as ReLUs. Você poderia simplesmente criar essa variável primeiro e depois passá-la para a função `relu()`:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Isso funciona bem: agora você pode controlar o limiar para todas as ReLUs utilizando a variável `threshold`. No entanto, se houver muitos parâmetros compartilhados como esse, será doloroso ter que passá-las como parâmetros o tempo todo. Muitas pessoas criam um dicionário de Python em seu modelo contendo todas as variáveis e as passam para todas as funções. Outras criam uma classe para cada módulo (por exemplo, uma classe `ReLU` utilizando variáveis de classe para lidar com o parâmetro compartilhado). Outra opção seria definir a variável compartilhada como um atributo da função `relu()` na primeira chamada, dessa maneira:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
        return tf.maximum(z, relu.threshold, name="max")
```

O TensorFlow oferece outra opção, o que pode levar a um código ligeiramente mais limpo e mais modular do que as soluções anteriores.<sup>5</sup> É um pouco complicado entender essa solução no início, mas, como é muito utilizada no TensorFlow, vale a pena entrar em detalhes. A ideia é utilizar a função `get_variable()` para criar a variável compartilhada se ela ainda não existir, ou reutilizá-la se ela já existe. O comportamento desejado (criando ou reutilizando) é controlado por um atributo do `variable_scope()`. Por exemplo, o código a seguir criará uma variável chamada “`relu/threshold`” (como um escalar, já que `shape=()`, e utilizando `0.0` como o valor inicial):

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
```

<sup>5</sup> Criar uma classe `ReLU` é indiscutivelmente a opção mais limpa, mas é bem pesado.

Note que, se a variável já tiver sido criada por uma chamada anterior a `get_variable()`, esse código criará uma exceção. Este comportamento impede a reutilização de variáveis por engano. Se você quiser reutilizar uma variável, você precisa explicitamente dizer isso definindo o atributo `reuse` da variável para `True` (assim não é necessário especificar a forma ou o inicializador):

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

Este código buscará a variável “`relu/threshold`” existente ou criará uma exceção se esta não existir, ou se não foi criada utilizando `get_variable()`. Como alternativa, você pode definir o atributo `reuse` como `True` dentro do bloco chamando o método do escopo `reuse_variables()`:

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```



Uma vez que `reuse` for definido para `True`, ele não pode ser ajustado novamente para `False` dentro do bloco. Além disso, se você definir outros escopos de variáveis dentro deste bloco, eles herdarão automaticamente `reuse=True`. Por fim, apenas variáveis criadas por `get_variable()` podem ser reutilizadas desta forma.

Agora você tem todas as peças que precisa para fazer a função `relu()` acessar a variável `threshold` sem ter que fazê-la passar por um parâmetro:

```
def relu(x):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold") # reusa variável existente
        [...]
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"): # cria a variável
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    relus = [relu(X) for relu_index in range(5)]
    output = tf.add_n(relus, name="output")
```

Este código primeiro define a função `relu()`, então cria a variável `relu/threshold` (como um escalar que será inicializado mais tarde para `0.0`) e constrói cinco ReLUs chamando a função `relu()`. A função `relu()` reutiliza a variável `relu/threshold` e cria os outros nós ReLU.



As variáveis criadas utilizando `get_variable()` são sempre nomeadas utilizando o nome de seus `variable_scope` como um prefixo (por exemplo, “`relu/threshold`”), mas, para todos os outros nós (incluindo as variáveis criadas com `tf.Variable()`), o escopo da variável atua como um novo escopo de nome. Principalmente se um escopo do nome com um nome idêntico já tiver sido criado, então um sufixo é adicionado para tornar o nome único. Por exemplo, todos os nós criados no código anterior (exceto a variável `threshold`) têm um nome prefixado com “`relu_1/`” para “`relu_5/`”, como mostrado na Figura 9-8.

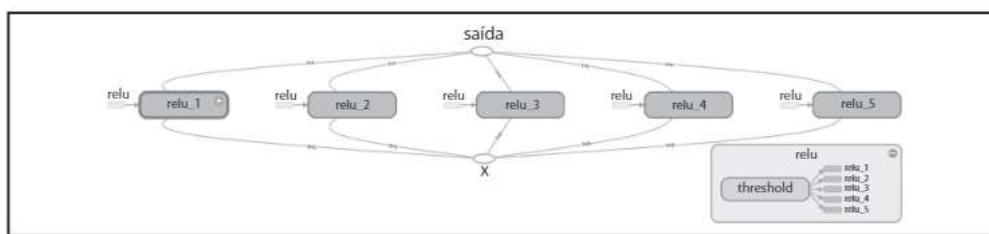


Figura 9-8. Cinco ReLUs compartilhando a variável `threshold`

É um tanto triste que a variável `threshold` deva ser definida fora da função `relu()`, na qual reside todo o resto do código ReLU. Para corrigir isso, o código a seguir cria a variável `threshold` dentro da função `relu()` na primeira chamada e, em seguida, a reutiliza nas chamadas subsequentes. Agora, a função `relu()` não tem que se preocupar com escopos de nomes ou variáveis compartilhadas: ela apenas chama `get_variable()`, que criará ou reutilizará a variável `threshold` (ela não precisa saber qual é o caso). O restante do código chama `relu()` por cinco vezes, certificando-se de definir para `reuse=False` na primeira chamada e `reuse=True` para as outras chamadas.

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
```

O grafo resultante é ligeiramente diferente do anterior, uma vez que a variável compartilhada vive dentro da primeira ReLU (veja a Figura 9-9).

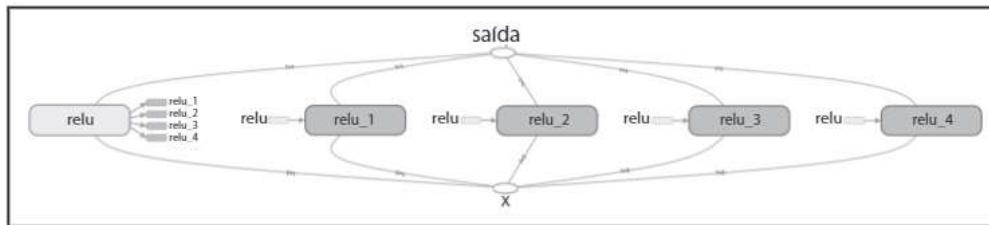


Figura 9-9. Cinco ReLUs compartilhando a variável threshold

Concluímos aqui esta introdução ao TensorFlow. Discutiremos tópicos mais avançados à medida que passarmos pelos próximos capítulos, em particular operações relacionadas a redes neurais profundas, convolutivas e recorrentes, bem como expandir o TensorFlow com a utilização do multithreading, queues, múltiplas GPUs e vários servidores.

## Exercícios

- Quais são os principais benefícios de se criar um grafo de cálculo em vez de executar diretamente os cálculos? Quais são as principais desvantagens?
- A declaração `a_val = a.eval(session=sess)` é equivalente a `a_val = sess.run(a)`?
- A declaração `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` é equivalente a `a_val, b_val = sess.run([a, b])`?
- É possível executar dois grafos na mesma sessão?
- Se você criar um grafo `g` contendo uma variável `w`, então iniciar dois segmentos e abrir uma sessão em cada, ambos utilizando o mesmo grafo `g`, cada sessão terá sua própria cópia da variável `w` ou esta será compartilhada?
- Quando uma variável é inicializada? Quando é destruída?
- Qual é a diferença entre um placeholder e uma variável?
- O que acontece quando você executa o grafo para avaliar uma operação que depende de um placeholder, mas você não fornece seu valor? O que acontece se a operação não depender do placeholder?
- Quando você executa um grafo, você pode fornecer o valor de saída de qualquer operação ou apenas o valor dos placeholders?

10. Como você pode definir uma variável para qualquer valor desejado (durante a fase de execução)?
11. Quantas vezes o autodiff de modo reverso precisa percorrer o gráfico para calcular os gradientes da função de custo em relação a dez variáveis? E quanto ao autodiff de modo avançado? E a diferenciação simbólica?
12. Implemente a Regressão Logística com o Gradiente Descendente com Minilotes utilizando o TensorFlow. Treine-o e o avalie no conjunto de dados em formato de luas (introduzido no Capítulo 5). Tente adicionar todas as características extras:
  - Defina o grafo dentro da função `logistic_regression()`, que pode ser facilmente reutilizada;
  - Salve pontos de verificação com a utilização de um `Saver` em intervalos regulares durante o treinamento e salve o modelo final ao término do treinamento;
  - Restaure o último ponto de verificação após a inicialização caso o treinamento seja interrompido;
  - Defina o grafo com a utilização de escopos do nome para que pareça bem no TensorBoard;
  - Adicione resumos para visualizar as curvas de aprendizado no TensorBoard;
  - Tente ajustar alguns hiperparâmetros, como a taxa de aprendizado ou o tamanho do minilote e veja a forma da curva de aprendizado.

As soluções para estes exercícios estão disponíveis no Apêndice A



## Capítulo 10

# Introdução às Redes Neurais Artificiais

As aves nos inspiraram a voar, a planta bardana inspirou a criação do velcro e a natureza inspirou muitas outras invenções. Parece lógico, então, olhar para a arquitetura do cérebro para obter a inspiração sobre como construir uma máquina inteligente. Esta é a ideia-chave que inspirou as *redes neurais artificiais* (RNA). No entanto, embora os aviões tenham sido inspirados por pássaros, eles não têm que bater suas asas. Igualmente, as RNA gradualmente têm se tornado bem diferentes de seus primos biológicos. Alguns pesquisadores até argumentam que devemos descartar completamente a analogia biológica (por exemplo, ao dizer “unidades” em vez de “neurônios”) para não restringir nossa criatividade com relação a sistemas biologicamente plausíveis.<sup>1</sup>

As RNA estão no cerne do Aprendizado Profundo. Elas são versáteis, poderosas e escaláveis, tornando-as ideais para lidar com grandes tarefas altamente complexas do Aprendizado de Máquina, como classificar bilhões de imagens (por exemplo, as do Google), alimentar serviços de reconhecimento da fala (por exemplo, a Siri da Apple), recomendar os melhores vídeos para centenas de milhões de usuários todos os dias (como o YouTube), ou aprender a derrotar o campeão mundial no jogo Go, examinando milhões de jogos passados e depois jogando contra si mesmo (AlphaGo da DeepMind).

Neste capítulo, introduziremos redes neurais artificiais começando por um breve tour pelas primeiras arquiteturas RNA. Em seguida, apresentaremos as *Perceptrons Multicamadas* (MLPs, do inglês) e como implementá-las com a utilização do TensorFlow para resolver o problema da classificação de dígito do MNIST (introduzido no Capítulo 3).

---

<sup>1</sup> Você pode ter o melhor dos dois mundos encarando abertamente inspirações biológicas sem ter medo de criar modelos biologicamente não realistas, desde que funcionem bem.

## De Neurônios Biológicos a Neurônios Artificiais

Surpreendentemente, as RNA existem há muito tempo: elas foram introduzidas pela primeira vez em 1943 pelo neurofisiologista Warren McCulloch e pelo matemático Walter Pitts. Em seu artigo de referência (<https://goo.gl/Ul4mxW>),<sup>2</sup> “Logical Calculus of Ideas Immanent in Nervous Activity” [“Cálculo lógico de ideias inerentes na atividade nervosa”, em tradução livre], McCulloch e Pitts apresentaram um modelo computacional simplificado utilizando *lógica proposicional* de como os neurônios biológicos podem trabalhar juntos em cérebros de animais na realização de cálculos complexos. Esta foi a primeira arquitetura de rede neural artificial. Desde então, muitas outras foram inventadas, como veremos.

Os primeiros sucessos das RNA até a década de 1960 levaram à crença generalizada de que, em breve, estaríamos conversando com máquinas verdadeiramente inteligentes. Quando ficou claro que essa promessa não seria cumprida (pelo menos por um bom tempo), o financiamento minguou e as RNA entraram em um longo e obscuro período. No início dos anos 1980, à medida que novas arquiteturas de rede foram inventadas e melhores técnicas de treinamento foram desenvolvidas, houve um ressurgimento do interesse nas RNA. Mas, na década de 1990, as técnicas avançadas do Aprendizado de Máquina, como Máquinas de Vetores de Suporte (veja o Capítulo 5), foram favorecidas pela maioria dos pesquisadores, pois pareciam oferecer melhores resultados e bases teóricas mais fortes. Finalmente, estamos testemunhando mais uma onda de interesse nas RNA. Essa onda morrerá como as anteriores? Existem algumas boas razões para acreditar que esta é diferente e terá um impacto muito mais profundo em nossas vidas:

- Temos, atualmente, uma grande quantidade de dados disponível para treinar redes neurais, e as RNA superam com frequência outras técnicas de AM em problemas muito grandes e complexos;
- O tremendo aumento do poder de computação desde a década de 1990 agora permite treinar grandes redes neurais em um período razoável de tempo. Isto se deve em parte à Lei de Moore, mas também graças à indústria dos jogos, que tem produzido poderosos GPU aos milhões;
- Os algoritmos de treinamento foram melhorados. Para ser sincero, eles são apenas um pouco diferentes dos usados na década de 1990, mas esses ajustes relativamente pequenos têm um enorme impacto positivo;
- Algumas limitações teóricas das RNA tornaram-se benignas na prática. Por exemplo, muitas pessoas pensavam que os algoritmos de treinamento RNA estavam condenados porque provavelmente poderiam ficar presos em um local

<sup>2</sup> “A Logical Calculus of Ideas Immanent in Nervous Activity”, W. McCulloch e W. Pitts (1943).

optimum, mas resultou que, na prática, isso é bastante raro (ou quando é o caso, geralmente são muito próximos do global optimum);

- As RNA parecem ter entrado em um círculo virtuoso de financiamento e progresso. Produtos surpreendentes baseados em RNA estão regularmente nas manchetes, que atraem cada vez mais atenção e financiamento, resultando em mais progressos e produtos ainda mais incríveis.

## Neurônios Biológicos

Antes de discutirmos neurônios artificiais, daremos uma rápida olhada em um neurônio biológico (representado na Figura 10-1). É uma célula de aparência incomum encontrada principalmente no córtex cerebral animal (por exemplo, seu cérebro), composto de um *corpo celular* contendo o núcleo e a maioria dos componentes complexos da célula e muitas extensões de ramificação chamadas *dendritos*, além de uma extensão muito longa chamada de *axônio*. O comprimento do axônio pode ser apenas algumas vezes mais longo do que o corpo da célula, ou até dezenas de milhares de vezes maior. Perto de sua extremidade, o axônio divide-se em muitos ramos chamados de *telodendros*, e na ponta desses ramos estão estruturas minúsculas chamadas *terminais sinápticos* (ou simplesmente *sinapses*) que estão conectadas aos dendritos (ou diretamente ao corpo celular) de outros neurônios. Os neurônios biológicos recebem curtos impulsos elétricos de outros neurônios através dessas sinapses chamadas *sinais*. Quando um neurônio recebe um número suficiente de sinais de outros neurônios em alguns milissegundos, ele dispara seus próprios sinais.

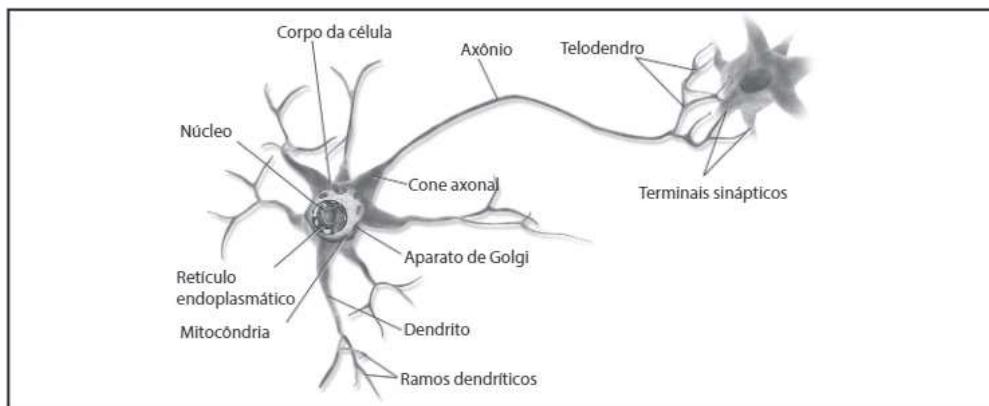


Figura 10-1. Neurônio Biológico<sup>3</sup>

<sup>3</sup> Imagem por Bruce Blaus (Creative Commons 3.0 (<https://creativecommons.org/licenses/by/3.0/>)). Reproduzida de <https://en.wikipedia.org/wiki/Neuron>.

Assim sendo, os neurônios biológicos individuais parecem se comportar de uma forma bastante simples, mas estão organizados em uma vasta rede de bilhões de neurônios, cada um normalmente conectado a milhares de outros. Cálculos altamente complexos podem ser realizados por uma vasta rede de neurônios bem simples, muito parecida com um formigueiro complexo que surge dos esforços combinados de simples formigas. A arquitetura das redes neurais biológicas (RNB)<sup>4</sup> ainda é objeto de pesquisa ativa, mas algumas partes do cérebro foram mapeadas e parece que os neurônios muitas vezes são organizados em camadas consecutivas, como mostrado na Figura 10-2.

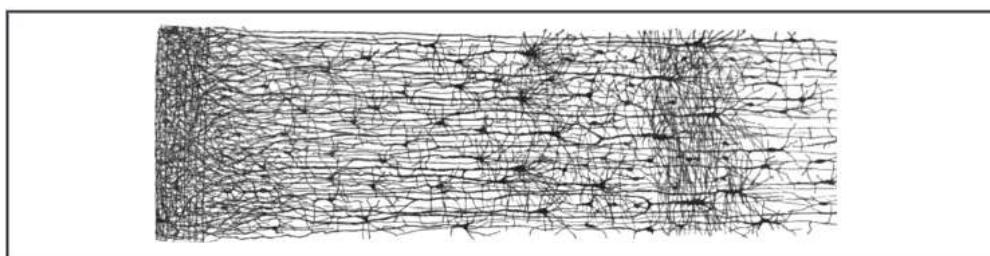


Figura 10-2. Múltiplas camadas em uma rede neural biológica (córtex humano)<sup>5</sup>

## Cálculos Lógicos com Neurônios

Warren McCulloch e Walter Pitts propuseram um modelo simples de neurônio biológico, mais tarde conhecido como *neurônio artificial*: este possui uma ou mais entradas binárias (ligado/desligado) e uma saída binária. O neurônio artificial ativa sua saída quando mais do que certo número de suas entradas estão ativas. McCulloch e Pitts demonstraram que mesmo com um modelo tão simplificado é possível construir uma rede de neurônios artificiais que calcula qualquer proposta lógica desejada. Por exemplo, construiremos algumas RNA que realizam vários cálculos lógicos (veja a Figura 10-3) assumindo que um neurônio seja ativado quando pelo menos duas de suas entradas estiverem ativas.

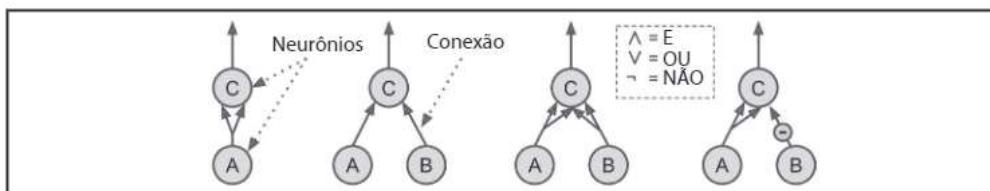


Figura 10-3. RNA realizando cálculos lógicos simples

<sup>4</sup> No contexto do Aprendizado de Máquina, a frase “redes neurais” geralmente se refere às RNA, não às RNB.

<sup>5</sup> Desenho de uma laminação cortical por S. Ramon y Cajal (domínio público). Reproduzido de [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex).

- A primeira rede à esquerda é a função de identidade: se o neurônio A estiver ativado, o neurônio C também será ativado (uma vez que ele recebe dois sinais de entrada do neurônio A), mas, se o neurônio A estiver desligado, o neurônio C também estará;
- A segunda rede executa um E lógico: o neurônio C é ativado somente quando ambos os neurônios A e B são ativados (um único sinal de entrada não é suficiente para ativar o neurônio C);
- A terceira rede executa um OU lógico: o neurônio C é ativado se o neurônio A ou o neurônio B estiverem ativados (ou ambos);
- Finalmente, se supormos que uma conexão de entrada pode inibir a atividade do neurônio (o que é o caso dos neurônios biológicos), então a quarta rede calcula uma proposição lógica ligeiramente mais complexa: o neurônio C é ativado somente se o neurônio A estiver ativo e se o neurônio B estiver desligado. Se o neurônio A estiver ativo o tempo todo, então você obtém um NÃO lógico: o neurônio C está ativo quando o neurônio B está desligado e vice-versa.

Você pode imaginar facilmente como essas redes podem ser combinadas para calcular expressões lógicas complexas (veja os exercícios no final do capítulo).

## O Perceptron

Inventado em 1957 por Frank Rosenblatt, o *Perceptron* é uma das mais simples arquiteturas RNA. Ela é baseada em um neurônio artificial ligeiramente diferente (veja a Figura 10-4) chamado de *unidade linear com threshold* (LTU, do inglês): as entradas e a saída agora são números (em vez de valores binários ligado/desligado) e cada conexão de entrada está associada a um peso. A LTU calcula uma soma ponderada de suas entradas ( $z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{w}^T \cdot \mathbf{x}$ ), então aplica uma *função degrau* a esta soma e exibe o resultado:  $h_w(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{w}^T \cdot \mathbf{x})$ .

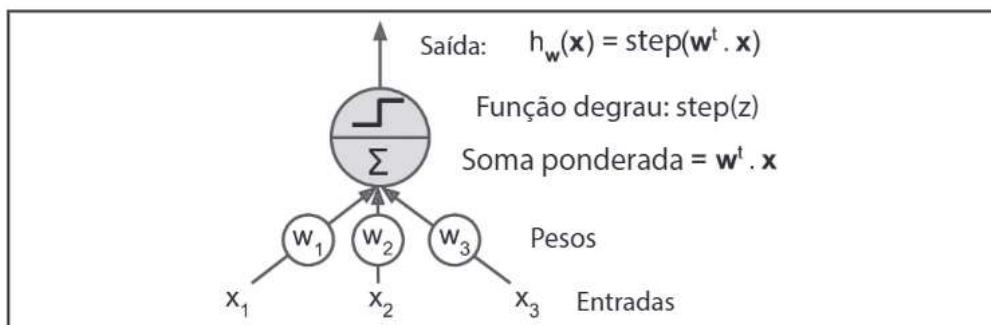


Figura 10-4. Unidade Linear com Threshold

A função degrau mais comum utilizada nos Perceptrons é a *função de Heaviside* (veja Equação 10-1). Algumas vezes é utilizada a função sinal.

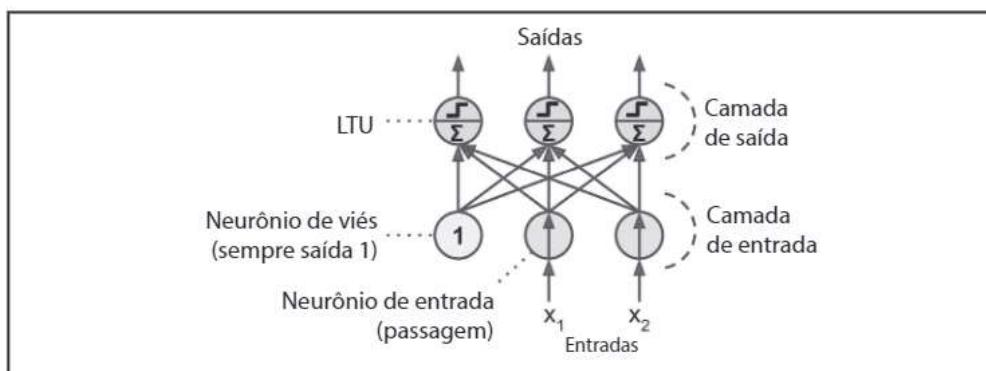
*Equação 10-1. Funções degrau comuns utilizadas em Perceptrons*

$$\text{heaviside}(z) = \begin{cases} 0 & \text{se } z < 0 \\ 1 & \text{se } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{se } z < 0 \\ 0 & \text{se } z = 0 \\ +1 & \text{se } z > 0 \end{cases}$$

Uma única LTU pode ser utilizada para classificação binária linear simples. Ela calcula uma combinação linear das entradas e, se o resultado exceder um limiar, exibe a classe positiva ou então negativa (como um classificador de Regressão Logística ou um SVM linear). Por exemplo, com base no comprimento e largura da pétala, você poderia utilizar uma única LTU para classificar as flores da íris (adicionando também uma característica extra de viés  $x_0 = 1$ , assim como fizemos em capítulos anteriores). Treinar uma LTU significa encontrar os valores corretos para  $w_0$ ,  $w_1$  e  $w_2$  (o algoritmo de treinamento será discutido em breve).

Um Perceptron é composto por uma única camada de LTUs,<sup>6</sup> com cada neurônio conectado a todas as entradas. Essas conexões são representadas geralmente com a utilização de neurônios de passagem especiais chamados de *neurônios de entrada*: eles apenas dão saída para qualquer entrada fornecida. Além disso, geralmente é adicionada uma característica extra de viés ( $x_0 = 1$ ), que é representada quando utilizamos um tipo especial de neurônio chamado *neurônio de viés*, que todas as vezes só exibe 1.

Um Perceptron com duas entradas e três saídas está representado na Figura 10-5. Este Perceptron pode classificar simultaneamente as instâncias em três classes binárias diferentes, o que o torna um classificador multioutput.



*Figura 10-5. Diagrama Perceptron*

<sup>6</sup> O nome Perceptron algumas vezes é utilizado para representar uma pequena rede com uma única LTU.

Então, como um Perceptron é treinado? O algoritmo de treinamento do Perceptron proposto por Frank Rosenblatt foi amplamente inspirado pela *regra de Hebb*. Em seu livro *The Organization of Behavior* [A Organização do Comportamento, em tradução livre] publicado em 1949, Donald Hebb sugeriu que, quando um neurônio biológico desencadeia outro neurônio com frequência, a conexão entre esses dois neurônios fica mais forte. Essa ideia foi resumida mais tarde por Siegrid Löwel nesta frase cativante: “*Cells that fire together, wire together*” [Células que disparam juntas permanecem conectadas, em tradução livre]. Esta regra tornou-se conhecida como a regra de Hebb (ou *aprendizado Hebbiano*); isto é, o peso da conexão entre dois neurônios é aumentado sempre que eles obtêm o mesmo resultado. Perceptrons são treinados com a utilização de uma variante desta regra que leva em consideração o erro produzido pela rede; não reforçando as conexões que levam à saída errada. Mais especificamente, o Perceptron é alimentado por uma instância de treinamento de cada vez e faz suas previsões para cada uma delas. Para cada neurônio de saída que produziu uma previsão incorreta, ele reforçará os pesos de conexão das entradas que contribuíram para a previsão correta. A regra é mostrada na Equação 10-2.

*Equação 10-2. Regra de Aprendizado do Perceptron (atualização do peso)*

$$w_{i,j}^{(\text{próximo degrau})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

- $w_{i,j}$  é o peso de conexão entre o  $i$ -ésimo neurônio de entrada e o  $j$ -ésimo neurônio de saída;
- $x_i$  é o valor da  $i$ -ésima entrada da instância de treinamento atual;
- $\hat{y}_j$  é a  $j$ -ésima saída do neurônio de saída para a instância de treinamento atual;
- $y_j$  é a  $j$ -ésima saída alvo do neurônio de saída para a instância de treinamento atual;
- $\eta$  é a taxa de aprendizado.

A fronteira de decisão de cada neurônio de saída é linear, então os Perceptrons são incapazes de aprender padrões complexos (como os classificadores de Regressão Logística). No entanto, se as instâncias de treinamento forem linearmente separáveis, Rosenblatt demonstrou que esse algoritmo convergiria para uma solução,<sup>7</sup> o que é chamado de *Teorema de convergência do Perceptron*.

O Scikit-Learn fornece uma classe `Perceptron` que implementa uma rede LTU simples. Ele pode ser utilizado praticamente como seria de esperar — por exemplo, no conjunto de dados da íris (introduzido no Capítulo 4):

---

<sup>7</sup> Note que esta solução geralmente não é única: em geral, quando os dados são linearmente separáveis, há uma infinidade de hiperplanos que podem separá-los.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # comprimento da pétala, largura da pétala
y = (iris.target == 0).astype(np.int) # Iris Setosa?

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

Você pode ter reconhecido que o algoritmo de aprendizado Perceptron se assemelha fortemente ao Gradiente Descendente Estocástico. Na verdade, a classe `Perceptron` do Scikit-Learn é equivalente à utilização do `SGDClassifier` com os seguintes hiperparâmetros: `loss="perceptron", learning_rate="constant", eta0=1` (a taxa de aprendizado) e `penalty=None` (sem regularização).

Observe que, contrariamente aos classificadores de Regressão Logística, os Perceptrons não produzem uma probabilidade de classe; em vez disso, apenas fazem previsões com base em um rígido limiar. Esta é uma das boas razões para preferir a Regressão Logística em relação aos Perceptrons.

Em sua monografia de 1969 intitulada *Perceptrons*, Marvin Minsky e Seymour Papert destacaram um número de sérias fraquezas dos Perceptrons, em particular o fato de serem incapazes de resolver alguns problemas triviais (por exemplo, o problema de classificação *Exclusive OR* (XOR); veja o lado esquerdo da Figura 10-6). Claro que isso também é verdade para qualquer outro modelo de classificação linear (como os classificadores de Regressão Logística), mas os pesquisadores esperavam muito mais dos Perceptrons e seu desapontamento foi grande: como resultado, muitos pesquisadores descartaram completamente o *conexionismo* (ou seja, o estudo de redes neurais) em favor de problemas de alto nível como lógica, resolução de questões e buscas.

Entretanto, algumas das suas limitações podem ser eliminadas ao empilharmos vários Perceptrons. A RNA resultante é chamada de *Perceptron Multicamada* (MLP). Em particular, como você pode verificar, a MLP pode resolver o problema XOR calculando a saída MLP representada à direita da Figura 10-6 para cada combinação de entradas: com as entradas (0, 0) ou (1, 1) a rede exibe 0, e com as entradas (0, 1) ou (1, 0) exibe 1.

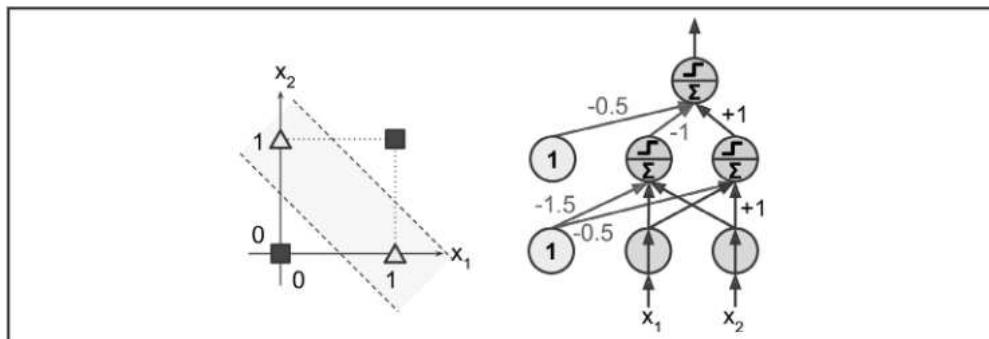


Figura 10-6. Problema de classificação XOR e um MLP que o soluciona

## Perceptron Multicamada e Retropropagação

Um MLP é composto de uma camada de entrada (*passthrough*), uma ou mais camadas LTUs chamadas de *camadas ocultas* e uma camada final LTU chamada de *camada de saída* (veja a Figura 10-7). Toda camada, exceto a camada de saída, inclui um neurônio de viés e está totalmente conectada à próxima camada. Quando uma RNA possui duas ou mais camadas ocultas, é chamada de *rede neural profunda* (DNN, do inglês).

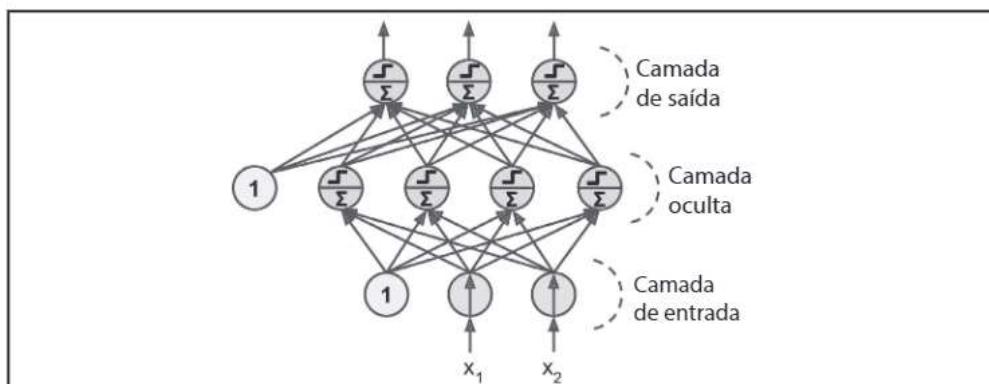


Figura 10-7. Multicamada Perceptron

Durante muitos anos, os pesquisadores esforçaram-se para encontrar uma forma de treinar MLPs, sem sucesso. Mas, em 1986, D. E. Rumelhart *et al.* publicaram um artigo inovador (<https://goo.gl/Wl7Xyc>)<sup>8</sup> apresentando o algoritmo de treinamento da retropropagação.<sup>9</sup> Hoje, o descreveríamos como o Gradiente Descendente utilizando autodiff

<sup>8</sup> "Learning Internal Representations by Error Propagation", D. Rumelhart, G. Hinton, R. Williams (1986).

<sup>9</sup> Este algoritmo foi inventado várias vezes por vários pesquisadores em diferentes campos, começando com P. Werbos em 1974.

em modo reverso (o Gradiente Descendente foi introduzido no Capítulo 4 e autodiff foi discutido no Capítulo 9).

O algoritmo alimenta cada instância de treinamento para a rede e calcula a saída de cada neurônio em cada camada consecutiva (este é o *forward pass*, assim como ao fazer previsões). Em seguida, ele mede o erro de saída da rede (isto é, a diferença entre a saída desejada e a saída real da rede) e calcula o quanto cada neurônio contribuiu para o erro de cada neurônio de saída na última camada oculta. Em seguida, passa a medir a quantidade dessas contribuições de erro provenientes de cada neurônio na camada oculta anterior, e assim por diante até o algoritmo alcançar a camada de entrada. Esta passagem reversa mede eficientemente o gradiente de erro em todos os pesos de conexão na rede, propagando para trás o gradiente de erro na rede (daí o nome do algoritmo). Se você verificar o algoritmo reverse-mode autodiff no Apêndice D descobrirá que as passagens de retropropagação forward e reverse simplesmente executam o autodiff no modo reverso. O último passo do algoritmo de retropropagação é um incremento do Gradiente Descendente com a utilização dos gradientes de erro medidos anteriormente em todos os pesos de conexão na rede.

Vamos encurtar ainda mais essa história: para cada instância de treinamento, o algoritmo de retropropagação primeiro faz uma previsão (*forward pass*), mede o erro, então passa por cada camada no reverso para medir a contribuição do erro em cada conexão (*reverse pass*) e, finalmente, ajusta os pesos da conexão para reduzir o erro (etapa Gradiente Descendente).

Para que este algoritmo funcione adequadamente, os autores fizeram uma mudança-chave na arquitetura do MLP: substituíram a função degrau pela função logística,  $\sigma(z) = 1 / (1 + \exp(-z))$ , o que foi essencial porque a função degrau contém apenas segmentos planos, portanto não há gradiente com o qual trabalhar (o Gradiente Descendente não pode se mover sobre uma superfície plana), enquanto a função logística possui uma derivada não zero bem definida em todos os lugares, permitindo que o Gradiente Descendente faça progresso a cada etapa. Em vez da função logística, o algoritmo de retropropagação pode ser utilizado com outras *funções de ativação*, das quais duas populares são:

A função tangente hiperbólica  $\tanh(z) = 2\sigma(2z) - 1$

Assim como a função logística, ela tem o formato de S, é contínua e diferenciável, mas seu valor de saída varia de -1 a 1 no início do treinamento (não de 0 a 1, como no caso da função logística), o que tende a tornar cada saída da camada mais ou menos normalizada (ou seja, centrada em torno de 0), ajudando frequentemente a acelerar a convergência.

### A função ReLU (introduzida no Capítulo 9)

$\text{ReLU}(z) = \max(0, z)$ . É contínua, mas infelizmente não é diferenciável em  $z = 0$  (a inclinação muda abruptamente, o que pode fazer com que o Gradiente Descendente salte). No entanto, na prática, ela funciona muito bem e tem a vantagem de o cálculo ser rápido. Mais importante ainda, o fato dela não possuir um valor máximo de saída também ajuda na redução de alguns problemas durante o Gradiente Descendente (voltaremos a isso no Capítulo 11).

Estas funções de ativação populares e suas derivadas estão representadas na Figura 10-8.

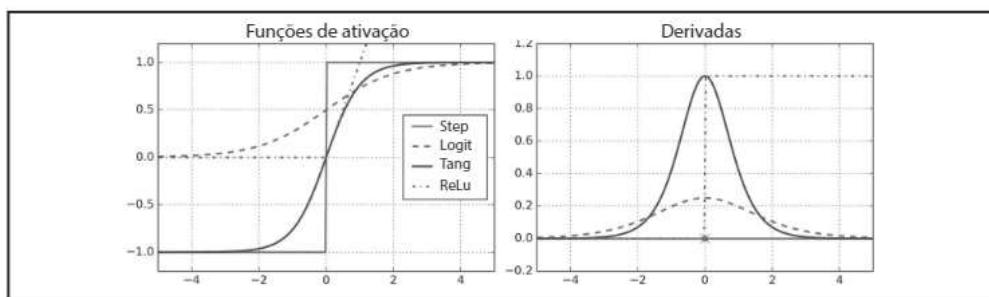


Figura 10-8. Funções de Ativação e suas derivadas

Um MLP com cada saída correspondente a uma classe binária diferente é frequentemente utilizado para classificação (por exemplo, spam/não spam, urgente/não urgente, etc.). A camada de saída normalmente é modificada ao substituir as funções de ativação individuais por uma função softmax compartilhada (veja a Figura 10-9) quando as classes são exclusivas (por exemplo, classes 0 a 9 para classificação de imagem de dígitos). A função softmax foi introduzida no Capítulo 4. A saída de cada neurônio corresponde à probabilidade estimada da classe correspondente. Observe que o sinal flui apenas em uma direção (das entradas para as saídas), então esta arquitetura é um exemplo de uma *rede neural feedforward* (FNN, do inglês).

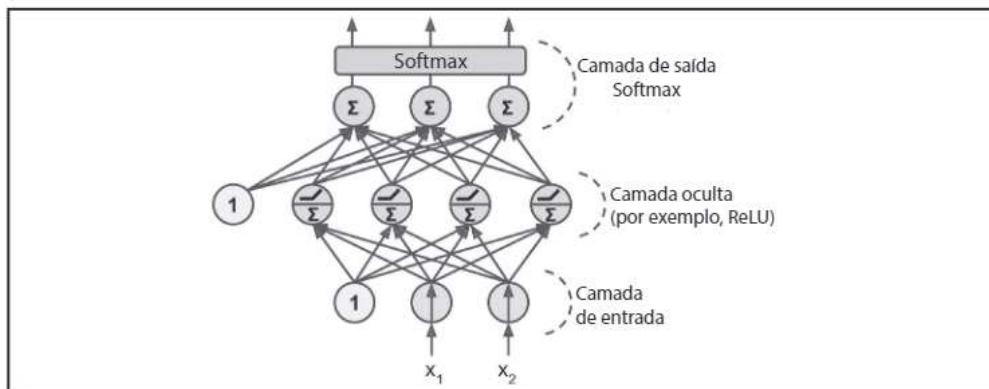


Figura 10-9. Um MLP moderno para a classificação (incluindo ReLU e softmax)



Os neurônios biológicos parecem implementar uma função de ativação aproximadamente sigmoidal (em forma de S), de modo que os pesquisadores as mantiveram por muito tempo. Mas acontece que a função de ativação ReLU geralmente funciona melhor nas RNA. Este é um dos casos em que a analogia biológica foi enganosa.

## Treinando um MLP com a API de Alto Nível do TensorFlow

A maneira mais simples de treinar um MLP com o TensorFlow é utilizar a API TF.Learn de alto nível, que oferece uma API compatível com o Scikit-Learn. A classe `DNNClassifier` facilita o treinamento de uma rede neural profunda independentemente do número de camadas ocultas, com uma camada de saída softmax para exibir probabilidades estimadas da classe. Por exemplo, o código a seguir treina uma DNN para classificação com duas camadas ocultas (uma com 300 neurônios e outra com 100 neurônios) e uma camada de saída softmax com 10 neurônios:

```
import tensorflow as tf

feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                         feature_columns=feature_cols)
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # se TensorFlow >= 1.1
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

Primeiro o código do conjunto de treinamento cria um conjunto de colunas de valor real (outros tipos de colunas, como colunas categóricas, estão disponíveis). Em seguida, criamos o `DNNClassifier` e o envolvemos em um auxiliar de compatibilidade do Scikit-Learn. Finalmente, executamos 40.000 iterações de treinamento com a utilização de lotes de 50 instâncias.

Se você executar este código no conjunto de dados MNIST (depois de escaloná-lo, por exemplo, utilizando o StandardScaler do Scikit-Learn), obterá um modelo que consegue cerca de 98,2% de acurácia no conjunto de teste! Isso é mais do que o melhor modelo que treinamos no Capítulo 3:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = dnn_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred['classes'])
0.9825000000000004
```



O pacote `tensorflow.contrib` contém muitas funções úteis, mas é um lugar de experimentação em código que ainda não evoluiu a ponto de fazer parte da API principal do TensorFlow. Então, a classe `DNNClassifier` (e qualquer outro código `contrib`) pode mudar sem aviso prévio no futuro.

Nos bastidores, a classe `DNNClassifier` cria todas as camadas de neurônios com base na função de ativação ReLU (podemos mudar isso ajustando o hiperparâmetro `activation_fn`). A camada de saída depende da função softmax e a função de custo é entropia cruzada (introduzida no Capítulo 4).

## Treinando um DNN Utilizando um TensorFlow Regular

Se você quiser mais controle sobre a arquitetura da rede, você vai preferir utilizar a API Python de nível inferior do TensorFlow (introduzida no Capítulo 9). Nesta seção, construiremos o mesmo modelo feito antes de utilizarmos esta API e implementaremos o Gradiente Descendente de Minilotes para treiná-lo no conjunto de dados MNIST. O primeiro passo é a fase de construção do grafo do TensorFlow. O segundo passo é a fase de execução, na qual executamos o grafo para treinar o modelo.

### Fase de Construção

Vamos começar. Primeiro, precisamos importar a biblioteca `tensorflow`. Então, devemos especificar o número de entradas e saídas e configurar o número de neurônios escondidos em cada camada:

```
import tensorflow as tf

n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Em seguida, utilize os nós `placeholders` para representar os dados de treinamento e metas, assim como você fez no Capítulo 9. A forma de `X` é definida parcialmente. Sabemos que será um tensor 2D (ou seja, uma matriz), com instâncias ao longo da primeira dimensão e características ao longo da segunda dimensão, e sabemos que o número de características será 28 x 28 (uma por pixel), mas ainda não sabemos quantas instâncias cada lote de treinamento conterá. Então, a forma de `X` será (`None`, `n_inputs`). Igualmente, sabemos que `y` é um tensor 1D com uma entrada por instância, mas, novamente, não sabemos o tamanho do lote de treinamento neste ponto, então o formato é (`None`).

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Agora, criaremos a rede neural real. O placeholder `X` agirá como a camada de entrada; durante a fase de execução será substituído por um lote de treinamento por vez (observe que todas as instâncias de um lote de treinamento serão processadas simultaneamente

pela rede neural). Agora você precisa criar as duas camadas ocultas e a camada de saída. As duas camadas ocultas são quase idênticas: diferem apenas das entradas às quais estão conectadas e pelo número de neurônios que contêm. A camada de saída também é muito semelhante, mas utiliza uma função de ativação softmax em vez de uma função de ativação ReLU. Criaremos uma função `neuron_layer()` que utilizaremos para criar uma camada por vez, e que precisará de parâmetros para especificar as entradas, o número de neurônios, a função de ativação e o nome da camada:

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs + n_neurons)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

Analisaremos este código linha por linha:

1. Primeiro criamos um escopo do nome utilizando o nome da camada: ele conterá todos os nós de computação para esta camada de neurônios. Isso é opcional, mas o grafo ficará muito mais agradável no TensorBoard se seus nós estiverem bem organizados;
2. Em seguida, obtemos o número de entradas (a primeira dimensão é para as instâncias) procurando a forma da matriz de entrada e obtendo o tamanho da segunda dimensão;
3. As próximas três linhas criam uma variável `W` que armazenará a matriz dos pesos (geralmente chamada de kernel da camada). Será um tensor 2D contendo todos os pesos de conexão entre cada entrada e cada neurônio; consequentemente, sua forma será (`n_inputs`, `n_neurons`). Com a utilização de uma distribuição normal (Gaussiana) truncada,<sup>10</sup> ela será inicializada aleatoriamente com um desvio padrão de  $2/\sqrt{n_{inputs} + n_{neurons}}$ . O uso deste desvio padrão específico ajuda o algoritmo a convergir muito mais rápido (discutiremos isso mais adiante no Capítulo 11; este é um desses pequenos ajustes nas redes neurais que tiveram um tremendo impacto em sua eficiência). É importante inicializar os pesos de conexão ale-

---

<sup>10</sup> Usar uma distribuição normal truncada em vez de uma distribuição normal regular garante que não haverá pesos grandes, o que poderia retardar o treinamento.

toriamente para todas as camadas ocultas para evitar quaisquer simetrias que o algoritmo Gradiente Descendente não seja capaz de quebrar;<sup>11</sup>

4. A próxima linha cria uma variável `b` para os vieses inicializados em 0 (sem problema de simetria neste caso) com um parâmetro de viés por neurônio;
5. Então, criamos um subgrafo para calcular  $Z = X \cdot W + b$ . Esta implementação vetorizada calculará eficientemente, de uma só vez, as somas ponderadas das entradas mais o termo de polarização para cada neurônio da camada para todas as instâncias do lote. Observe que adicionar um array 1D (`b`) em uma matriz 2D com o mesmo número de colunas (`X . W`) resulta na adição do array 1D em cada linha da matriz: isso é chamado de *broadcasting*;
6. Finalmente, o código retorna `activation(Z)`, ou então ele retorna apenas `Z` se for fornecido um parâmetro `activation` como o `tf.nn.relu` (isto é,  $\max(0, Z)$ ).

Certo, então, agora você tem uma função para a criação de uma camada de neurônio. Vamos utilizá-la para criar a rede neural profunda! A primeira camada oculta toma `X` como entrada. A segunda toma como entrada a saída da primeira camada oculta. E finalmente, a camada de saída toma como entrada a saída da segunda camada oculta.

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.relu)
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

Observe que, mais uma vez, usamos escopos do nome para maior clareza. Repare também que `logits` é a saída da rede neural *antes* de passar pela função de ativação softmax: por razões de otimização, lidaremos com o cálculo softmax mais tarde.

Como esperado, o TensorFlow vem com muitas funções úteis para criar camadas padrão de redes neurais, portanto nem sempre é necessário definir sua própria função `neuron_layer()` como acabamos de fazer. Por exemplo, a função `tf.layers.dense()` do TensorFlow (anteriormente chamada `tf.contrib.layers.fully_connected()`) cria uma camada totalmente conectada em que todas as entradas estão conectadas a todos os neurônios da camada. Utilizando a estratégia de inicialização apropriada, ela cuida da criação das variáveis de pesos e polarizações denominadas `kernel` e `bias`, respectivamente, e você pode definir a função de ativação utilizando o argumento `activation`. Como

---

<sup>11</sup> Por exemplo, se você definir todos os pesos como 0, todos os neurônios produzirão 0 e o gradiente de erro será o mesmo para todos os neurônios em uma determinada camada oculta. A etapa do Gradiente Descendente atualizará todos os pesos exatamente da mesma maneira em cada camada para que todos permaneçam iguais. Em outras palavras, apesar de ter centenas de neurônios por camada, seu modelo atuará como se houvesse apenas um neurônio por camada. Não vai acelerar.

veremos no Capítulo 11, ela suporta também parâmetros de regularização. Ajustaremos o código anterior para utilizar a função `dense()` em vez da nossa função `neuron_layer()`. Basta substituir a sessão de construção `dnn` pelo código a seguir:

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                             activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                             activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Agora que já temos o modelo de rede neural, precisamos definir a função de custo para treiná-lo. Assim como fizemos na Regressão Softmax no Capítulo 4, utilizaremos a entropia cruzada. Como discutimos anteriormente, a entropia cruzada penalizará os modelos que estimam uma baixa probabilidade para a classe-alvo. O TensorFlow fornece várias funções para calcular a entropia cruzada. Usaremos `sparse_softmax_cross_entropy_with_logits()`, que calcula a entropia cruzada com base em “logits” (ou seja, a saída da rede *antes* de passar pela função de ativação softmax) e espera os rótulos na forma de números inteiros que variam de 0 ao número de classes menos 1 (no nosso caso, de 0 a 9). Isso nos dará um tensor 1D que contém a entropia cruzada de cada instância. Podemos, então, utilizar a função `reduce_mean()` do TensorFlow para calcular a entropia cruzada média em todas as instâncias.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                               logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```



A função `sparse_softmax_cross_entropy_with_logits()` é equivalente a aplicar a função de ativação softmax e calcular a entropia cruzada, mas é mais eficiente e cuida adequadamente dos casos dos laterais: quando os logits são volumosos, os erros de arredondamento do ponto flutuante podem fazer com que a saída softmax seja exatamente igual a 0 ou 1 e, neste caso, a equação da entropia cruzada conteria um termo  $\log(0)$ , igual a infinito negativo. A função `sparse_softmax_cross_entropy_with_logits()` resolve este problema calculando  $\log(\epsilon)$ , sendo que  $\epsilon$  é um pequeno número positivo. É por isso que não aplicamos a função de ativação softmax anteriormente. Existe também outra função chamada `softmax_cross_entropy_with_logits()`, que leva os rótulos na forma de vetores *one-hot* (em vez de ints de 0 ao número de classes menos 1).

Nós temos o modelo de rede neural, temos a função de custo e agora precisamos definir um `GradientDescentOptimizer` que ajustará os parâmetros do modelo e minimizará a função de custo. Nada de novo; é exatamente como fizemos no Capítulo 9:

```

learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)

```

O último passo importante na fase de construção é especificar como avaliar o modelo, e nós utilizaremos a acurácia como medida de desempenho. Primeiro, determine se a previsão da rede neural está correta para cada instância verificando se o logit mais alto corresponde ou não à classe-alvo. Para isto, você pode utilizar a função `in_top_k()`. Ela retorna um tensor 1D cheio de valores booleanos, então precisamos fazer *casting* desses booleanos para `float` e, em seguida, calcular a média, gerando a acurácia geral da rede.

```

with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

```

E, como de costume, precisamos criar um nó para inicializar todas as variáveis e também um `Saver` para salvar nossos parâmetros treinados do modelo para o disco:

```

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

Ufa! Isso conclui a fase de construção. Estamos falando de menos de 40 linhas de código, mas foi bastante intenso: criamos placeholders para as entradas e os alvos, criamos uma função para construir uma camada de neurônio, a utilizamos para criar a DNN, definimos a função de custo, criamos um otimizador e, finalmente, definimos a medida de desempenho. Agora, vamos à fase de execução.

## Fase de Execução

Esta parte é bem mais curta e simples. Primeiro carregaremos o MNIST. Poderíamos utilizar o Scikit-Learn para isso, como fizemos em capítulos anteriores, mas o TensorFlow oferece seu próprio ajudante que obtém os dados, os escalona (entre 0 e 1), os embaralha e fornece uma função simples para carregar um minilote por vez. Além disso, os dados já estão divididos em um conjunto de treinamento (55.000 instâncias), um de validação (5.000 instâncias) e um de testes (10.000 instâncias). Então, utilizaremos esse ajudante:

```

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")

```

Agora, definimos o número de épocas que queremos rodar, bem como o tamanho dos minilotes:

```

n_epochs = 40
batch_size = 50

```

A seguir podemos treinar o modelo:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: mnist.validation.images,
                                           y: mnist.validation.labels})
        print(epoch, "Train accuracy:", acc_train, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Este código abre uma sessão TensorFlow e executa o nó `init` que inicializa todas as variáveis. Em seguida, ele executa o loop principal de treinamento: em cada época, o código itera através de vários minilotes que correspondem ao tamanho do conjunto de treinamento. Cada minilote é buscado pelo método `next_batch()` e, então, o código executa a operação de treinamento fornecendo os dados de entrada de minilotes atuais e alvos. Em seguida, ao final de cada época, o código avalia o modelo no último minilote no conjunto completo de validação e imprime o resultado. Finalmente, os parâmetros do modelo são salvos no disco.

## Utilizando a Rede Neural

Agora que a rede neural está treinada, você pode usá-la para fazer previsões. Para isso, reutilize a mesma fase de construção, mas mude a fase de execução desta forma:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...] # algumas novas imagens (escaladas de 0 a 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

Primeiro, o código carrega os parâmetros do modelo a partir do disco. Em seguida, carrega algumas novas imagens que classificará. Lembre-se de aplicar o mesmo escalonamento de características, assim como os dados de treinamento (neste caso, escalone-o de 0 a 1). Em seguida, o código avalia o nó `logits`. Você aplicaria a função `softmax()` aos logits se quisesse conhecer todas as probabilidades estimadas das classes, mas, se quisesse apenas prever uma classe, escolheria a que possui o maior valor de logit (a função `argmax()` cumpre essa tarefa).

## Ajustando os Hiperparâmetros da Rede Neural

A flexibilidade das redes neurais também é uma das suas principais desvantagens: há muitos hiperparâmetros para ajustar. Você pode não apenas utilizar qualquer *topologia de rede* imaginável (como os neurônios estão interconectados), mas mesmo em um MLP simples você pode alterar seu número de camadas, o número de neurônios por camada, o tipo de função de ativação utilizado em cada camada, o peso da lógica de inicialização e muito mais. Como saber qual seria a melhor combinação de hiperparâmetros para sua tarefa?

Claro que você pode utilizar a grid search com validação cruzada para encontrar os hiperparâmetros corretos, como feito em capítulos anteriores, mas existem muitos hiperparâmetros a serem ajustados, e, como o treinamento de uma rede neural em um grande conjunto de dados demora muito, você só poderá explorar uma pequena parcela do espaço de hiperparâmetro em uma quantidade razoável de tempo. É bem melhor utilizar a busca aleatória (<https://goo.gl/QFjMKu>), como discutimos no Capítulo 2. Outra opção seria utilizar uma ferramenta como o Oscar (<http://oscar.calldesk.ai/>), que implementa algoritmos mais complexos para ajudá-lo na busca de um bom conjunto de hiperparâmetros rapidamente.

Ter uma ideia de quais valores seriam razoáveis para cada hiperparâmetro é útil a fim de restringir o espaço de busca. Começaremos com o número de camadas ocultas.

### Número de Camadas Ocultas

Em vários problemas é possível começar com uma única camada oculta e obter resultados razoáveis. Ficou demonstrado que um MLP com apenas uma camada oculta pode modelar até as funções mais complexas desde que tenha neurônios suficientes. Durante muito tempo, esses fatos convenceram os pesquisadores de que não havia necessidade de investigar redes neurais mais profundas. No entanto, foi ignorado o fato de que as redes profundas têm uma *eficiência de parâmetro* muito maior do que as rasas: utilizando exponencialmente menos neurônios do que as redes superficiais, elas podem modelar funções complexas, tornando seu treinamento muito mais rápido.

Para entender o porquê, suponha que você seja convidado a desenhar uma floresta com a utilização de algum software de ilustração, mas está proibido de usar o copiar/colar. Você precisaria desenhar cada árvore individualmente, ramo por ramo, folha por folha. Se, em vez disso, você pudesse desenhar uma folha, copiar/colar para desenhar um ramo, copiar/colar esse ramo para criar uma árvore e, finalmente, copiar/colar esta árvore para criar uma floresta, você terminaria muito mais rápido. Os dados do mundo real

muitas vezes são estruturados de maneira bem hierárquica e as DNNs se aproveitam automaticamente desse fato: camadas ocultas inferiores modelam as estruturas de baixo nível (por exemplo, segmentos de linha de várias formas e orientações), camadas ocultas intermediárias combinam essas estruturas de baixo nível para modelar estruturas de nível intermediário (por exemplo, quadrados e círculos), e, para modelar estruturas de alto nível (por exemplo, rostos), as camadas ocultas mais altas e a camada de saída combinam essas estruturas intermediárias.

Essa arquitetura hierárquica não só ajuda as DNNs a convergir mais rapidamente para uma boa solução, mas também melhora sua capacidade de generalizar para novos conjuntos de dados. Por exemplo, se você já treinou um modelo para reconhecer rostos em imagens e agora deseja treinar uma nova rede neural para reconhecer penteados, então você pode iniciar o treinamento reutilizando as camadas inferiores da primeira rede. Em vez de inicializar aleatoriamente os pesos e as polarizações das primeiras camadas da nova rede neural, você pode inicializá-los no valor dos pesos e viéses das camadas inferiores da primeira rede. Desta forma, a rede não terá que aprender do zero todas as estruturas presentes na maioria das imagens; só terá que aprender aquelas do nível superior (por exemplo, penteados).

Em resumo, comece com apenas uma ou duas camadas ocultas e ela funcionará muito bem para muitos problemas (por exemplo, você pode facilmente alcançar uma acurácia acima de 97% no conjunto de dados MNIST utilizando uma única camada oculta com poucas centenas de neurônios, e acima de 98% de acurácia com a utilização de duas camadas ocultas com a mesma quantidade de neurônios, em aproximadamente a mesma quantidade de tempo de treinamento). Para problemas mais complexos, você pode aumentar gradualmente o número de camadas ocultas até começar a sobreajustar o conjunto de treinamento. Tarefas muito complexas, como a classificação de grandes imagens, ou o reconhecimento de voz, normalmente requerem redes com dezenas de camadas (ou mesmo centenas, mas não totalmente conectadas, como veremos no Capítulo 13) e elas precisam de uma enorme quantidade de dados de treinamento. No entanto, você raramente terá que treinar essas redes do zero: é muito mais comum reutilizar partes de uma rede pré-treinada de última geração que execute uma tarefa similar. O treinamento será muito mais rápido e exigirá muito menos dados (discutiremos isso no Capítulo 11).

## Número de Neurônios por Camada Oculta

Obviamente, o número de neurônios nas camadas de entrada e saída é determinado pelo tipo de entrada e saída que sua tarefa requer. Por exemplo, a tarefa MNIST requer  $28 \times 28 = 784$  neurônios de entrada e 10 neurônios de saída. Quanto às camadas ocultas, uma prática comum é dimensioná-las para formar um funil com cada vez menos neurônios

em cada camada, sendo o raciocínio que muitas características de baixo nível podem se unir em muito menos características de alto nível. Por exemplo, uma rede neural típica para o MNIST pode ter duas camadas ocultas, a primeira com 300 neurônios e a segunda com 100. No entanto, esta prática não é tão comum agora e você pode simplesmente utilizar o mesmo tamanho para todas as camadas ocultas — por exemplo, todas as camadas ocultas com 150 neurônios significa ajustar apenas um hiperparâmetro em vez de um por camada. Assim como para o número de camadas, você pode tentar aumentar o número de neurônios gradualmente até que a rede comece a se sobreajustar. Em geral, você obterá mais vantagem aumentando o número de camadas do que o número de neurônios por camada. Infelizmente, como você pode ver, encontrar a quantidade perfeita de neurônios ainda requer um pouco de magia.

Uma abordagem mais simples seria escolher um modelo com mais camadas e neurônios do que você realmente precisa e, em seguida, utilizar uma parada antecipada para impedir que ele seja sobreajustado (e outras técnicas de regularização, especialmente o *dropout*, como veremos no Capítulo 11). Isso foi apelidado de abordagem “calça elástica”:<sup>12</sup> em vez de perder tempo procurando por calças que sejam perfeitamente do seu tamanho, basta utilizar calças grandes que encolherão para o tamanho certo.

## Funções de Ativação

Na maioria dos casos, você pode utilizar a função de ativação ReLU nas camadas ocultas (ou uma de suas variantes, como veremos no Capítulo 11). É um pouco mais rápida para calcular do que outras funções de ativação e o Gradiente Descendente não fica tão preso em platôs graças ao fato de que ele não satura para grandes valores de entrada (em oposição à função logística ou à função tangente hiperbólica, que satura em 1).

Quando as classes são mutuamente exclusivas, a função de ativação softmax geralmente é uma boa opção para as tarefas de classificação na camada de saída. Quando não são (ou quando existem apenas duas classes), utilize a função logística. Nas tarefas de regressão, você pode simplesmente não utilizar nenhuma função de ativação para a camada de saída.

Concluímos, então, esta introdução às redes neurais artificiais. Nos capítulos seguintes, discutiremos técnicas para treinar redes muito profundas e distribuiremos treinamentos em vários servidores e GPUs. Então exploraremos outras arquiteturas de redes neurais populares: convolutivas, recorrentes e autoencoders.<sup>13</sup>

---

12 *Stretch pants*, por Vincent Vanhoucke em sua aula de Aprendizado Profundo (<https://goo.gl/Y5TFqz>) no Udacity.com.

13 Outras Arquiteturas Populares ANN são apresentadas no Apêndice E

## Exercícios

1. Desenhe uma RNA com a utilização dos neurônios artificiais originais (como os da Figura 10-3) que calcule  $A \oplus B$  (sendo que  $\oplus$  representa a operação XOR). Dica:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
2. Por que geralmente é preferível utilizar um classificador de Regressão Logística em vez de um Perceptron clássico (ou seja, uma única camada de unidades lineares com threshold treinadas com a utilização do algoritmo de treinamento Perceptron)? Como você pode ajustar um Perceptron para torná-lo equivalente a um classificador de Regressão Logística?
3. Por que a função de ativação logística foi um ingrediente-chave na formação dos primeiros MLPs?
4. Nomeie três funções de ativação populares. Consegue desenhá-las?
5. Suponha que você tenha um MLP composto de uma camada de entrada com 10 neurônios de passagem, seguido de uma camada oculta com 50 neurônios artificiais e, finalmente, uma camada de saída com 3 neurônios artificiais. Todos os neurônios artificiais utilizam a função de ativação ReLU.
  - Qual é a forma da matriz de entrada X?
  - O que dizer da forma do vetor de peso da camada oculta  $W_h$ , e da forma de seus vetores de polarização  $b_h$ ?
  - Qual é a forma do vetor de peso  $W_o$ , da camada de saída e seu vetor de polarização  $b_o$ ?
  - Qual é a forma da saída de rede da matriz Y?
  - Escreva a equação que calcule a saída de rede da matriz Y como uma função de X,  $W_h$ ,  $b_h$ ,  $W_o$  e  $b_o$ .
6. De quantos neurônios você precisa na camada de saída se quiser classificar o e-mail como spam ou não spam? Que função de ativação você deve utilizar na camada de saída? Se, em vez disso, você quiser abordar o MNIST, de quantos neurônios você precisa utilizando qual função de ativação na camada de saída? Responda as mesmas perguntas para fazer sua rede prever os preços imobiliários, como no Capítulo 2.
7. O que é retropropagação e como isso funciona? Qual é a diferença entre retropropagação e autodiff de modo reverso?

8. Você consegue listar todos os hiperparâmetros que podem ser ajustados em um MLP? Se o MLP se sobreajusta aos dados de treinamento, como ajustar esses hiperparâmetros para tentar resolver o problema?
9. Treine um MLP profundo no conjunto de dados MNIST e veja se você consegue obter mais de 98% de precisão. Assim como no último exercício do Capítulo 9, tente adicionar todos os extras (ou seja, salve os pontos de verificação, restaure o último ponto de verificação em caso de interrupção, adicione resumos, plote as curvas de aprendizado com a utilização do TensorBoard e assim por diante).

As soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 11

# Treinando Redes Neurais Profundas

No Capítulo 10, introduzimos as redes neurais artificiais e treinamos nossa primeira rede neural profunda. Mas era uma DNN muito rasa, com apenas duas camadas ocultas. E, se você precisar resolver um problema muito mais complexo, como detectar centenas de tipos de objetos em imagens de alta resolução? Talvez você precise treinar uma DNN muito mais profunda, com (digamos) 10 camadas cada uma contendo centenas de neurônios, conectadas por centenas de milhares de conexões, o que não seria nada fácil:

- Primeiro, você enfrentaria o complicado problema dos *vanishing gradients* (ou o problema relacionado com os *exploding gradients*) que afeta as redes neurais profundas e faz com que seja muito difícil treinar as camadas inferiores;
- Segundo, o treinamento seria extremamente lento em uma rede tão grande;
- Terceiro, um modelo com milhões de parâmetros poderia colocar em risco o sobreajuste do conjunto de treinamento.

Neste capítulo, passaremos por cada um desses problemas e apresentaremos técnicas para resolvê-los. Vamos começar por explicar o problema dos vanishing gradients e explorar algumas das soluções mais populares para ele. Em seguida, analisaremos vários otimizadores que podem acelerar tremendamente o treinamento de grandes modelos em comparação com o Gradiente Descendente regular. Finalmente, passaremos por algumas técnicas populares de regularização para grandes redes neurais.

Com estas ferramentas, você poderá treinar redes muito profundas: bem-vindo ao Aprendizado Profundo!

## Problemas dos Gradientes: Vanishing/Exploding

Conforme discutimos no Capítulo 10, o algoritmo de retropropagação funciona quando passa da camada de saída para a camada de entrada, propagando o gradiente de erro no caminho. Uma vez que o algoritmo tenha calculado o gradiente da função de custo

em relação a cada parâmetro na rede, ele utiliza esses gradientes para atualizar cada parâmetro com uma etapa do Gradiente Descendente.

Infelizmente, os gradientes geralmente ficam cada vez menores à medida que o algoritmo avança para as camadas inferiores. Como resultado, a atualização do Gradiente Descendente deixa os pesos de conexão da camada inferior praticamente inalterados e o treinamento nunca converge para uma boa solução. Este é o chamado problema dos vanishing gradients. Em alguns casos, o contrário pode acontecer: os gradientes podem crescer cada vez mais, as camadas recebem atualizações de peso insanamente grandes e o algoritmo diverge, o que é o problema dos exploding gradients, encontrados principalmente em redes neurais recorrentes (veja o Capítulo 14). No geral, as redes neurais profundas sofrem de gradientes instáveis; diferentes camadas podem aprender em velocidades muito diferentes.

Embora durante muito tempo este comportamento infeliz tenha sido observado empiricamente (foi uma das razões pelas quais as redes neurais profundas foram abandonadas por muito tempo), somente por volta de 2010 é que foi feito um progresso significativo para sua compreensão. Um artigo intitulado “Understanding the Difficulty of Training Deep Feedforward Neural Networks” (<http://goo.gl/IrhAef>) por Xavier Glorot e Yoshua Bengio<sup>1</sup> encontrou alguns suspeitos, incluindo a combinação da popular função de ativação sigmoidal logística e a técnica de inicialização do peso que foi mais popular na época, ou seja, a inicialização aleatória, quando utilizamos uma distribuição normal com uma média de 0 e um desvio padrão de 1. Em suma, eles mostraram que, com esta função de ativação e esse esquema de inicialização, a variância das saídas de cada camada era muito maior do que a variância de suas entradas. Ao avançar na rede, a variância continua aumentando após cada camada até que a função de ativação sature nas camadas superiores. Isso fica pior pelo fato de a função logística ter uma média de 0,5, não 0 (a função tangente hiperbólica tem uma média de 0 e se comporta um pouco melhor em redes profundas do que a função logística).

Olhando para a função de ativação logística (veja a Figura 11-1), você verá que, quando as entradas se tornam grandes (negativas ou positivas), a função satura em 0 ou 1 com uma derivada extremamente próxima de 0. Assim, quando a retropropagação entra em ação, ela praticamente não tem um gradiente para se propagar de volta através da rede, e o pouco gradiente existente continua diluindo à medida que a retropropagação avança através das camadas superiores, de forma a não sobrar praticamente nada para as camadas inferiores.

---

<sup>1</sup> “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, X. Glorot, Y. Bengio (2010).

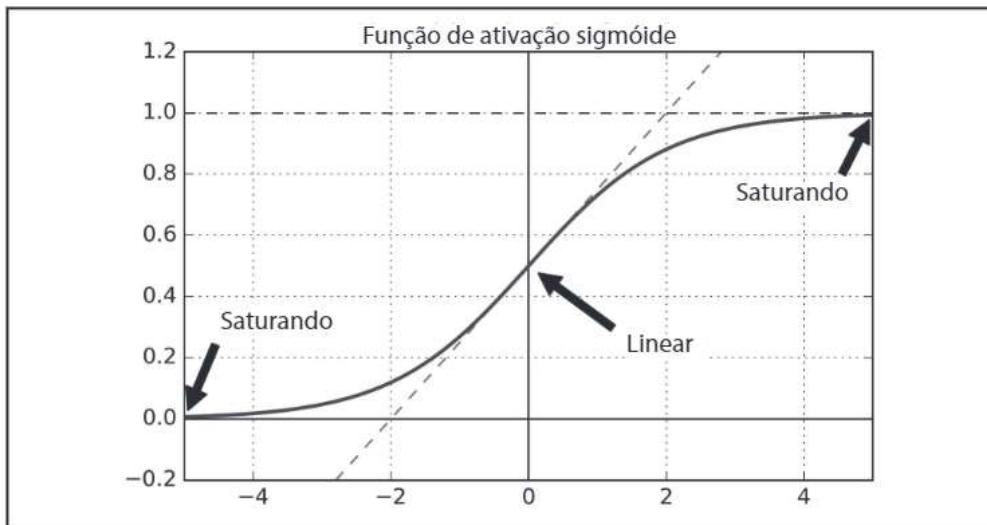


Figura 11-1. Saturação da função de ativação logística

## Inicialização Xavier e Inicialização He

Glorot e Bengio propõem em seu artigo uma forma de aliviar significativamente esse problema. Precisamos que o sinal flua corretamente em ambas as direções: na direção *forward* quando fizer previsões e na direção *reverse* ao retropropagar gradientes. Nós não queremos que o sinal desapareça, nem queremos que ele exploda e sature. Para que o sinal flua corretamente, os autores argumentam que a variância das saídas de cada camada deve ser igual à variância de suas entradas,<sup>2</sup> e também que os gradientes tenham igual variância antes e depois de fluir na direção reversa através de uma camada (verifique o artigo se você está interessado nos detalhes matemáticos). Na verdade, não é possível garantir ambos, a menos que a camada tenha um número igual de conexões de entrada e saída, mas o que eles propuseram funciona muito bem na prática: os pesos de conexão devem ser inicializados aleatoriamente, conforme descrito na Equação 11-1, em que  $n_{\text{entradas}}$  e  $n_{\text{saídas}}$  são o número de conexões de entrada e saída para a camada cujos pesos estão sendo inicializados (também chamado de *fan-in* e *fan-out*). Esta estratégia de inicialização é chamada *Inicialização Xavier* (homenageando o primeiro nome do autor), ou algumas vezes *Inicialização Glorot*.

2 Leia a analogia: se você ajustar o botão do amplificador de microfone muito perto de zero, as pessoas não ouvirão sua voz, mas, se ajustar bem perto do máximo, sua voz ficará saturada e as pessoas não entenderão o que você está dizendo. Agora, imagine uma cadeia de tais amplificadores: todos precisam ser configurados adequadamente para que sua voz saia alta e clara no final da corrente. Sua voz deve sair de cada amplificador com a mesma amplitude que entrou.

*Equação 11-1. Inicialização Xavier (quando se utiliza a função de ativação logística)*

$$\text{Distribuição Normal com média 0 e desvio padrão } \sigma = \sqrt{\frac{2}{n_{\text{entradas}} + n_{\text{saídas}}}}$$

$$\text{Ou uma distribuição uniforme entre } -r \text{ e } +r, \text{ com } r = \sqrt{\frac{6}{n_{\text{entradas}} + n_{\text{saídas}}}}$$

Quando o número de conexões de entrada é aproximadamente igual ao número de conexões de saída, você obtém equações mais simples<sup>3</sup> (por exemplo,  $\sigma = 1/\sqrt{n_{\text{entradas}}}$  ou  $r = \sqrt{3}/\sqrt{n_{\text{saídas}}}$ ).

O uso da estratégia da inicialização Xavier pode acelerar consideravelmente o treinamento e é um dos truques que levaram ao sucesso atual do Aprendizado Profundo. Alguns artigos recentes (<http://goo.gl/VHP3pB>)<sup>4</sup> forneceram estratégias semelhantes para diferentes funções de ativação, conforme mostrado na Tabela 11-1. A estratégia de inicialização para a função de ativação ReLU (e suas variantes, incluindo a ativação ELU descrita brevemente) às vezes é chamada de *Inicialização He* (sobrenome do autor). Esta é a estratégia que utilizamos no Capítulo 10.

*Tabela 11-1. Parâmetros de inicialização para cada tipo de função de ativação*

Função de ativação	Distribuição uniforme $[-r, r]$	Distribuição normal
Logística	$r = \sqrt{\frac{6}{n_{\text{entradas}} + n_{\text{saídas}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{entradas}} + n_{\text{saídas}}}}$
Tangente Hiperbólica	$r = 4\sqrt{\frac{6}{n_{\text{entradas}} + n_{\text{saídas}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{entradas}} + n_{\text{saídas}}}}$
ReLU (e suas variantes)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{entradas}} + n_{\text{saídas}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{entradas}} + n_{\text{saídas}}}}$

Por padrão, a função `tf.layers.dense()` (introduzida no Capítulo 10) utiliza a inicialização Xavier (com uma distribuição uniforme), mas é possível mudar para a inicialização He usando a função `variance_scaling_initializer()` desta forma:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                         kernel_initializer=he_init, name="hidden1")
```

<sup>3</sup> Esta estratégia simplificada já foi proposta anteriormente — por exemplo, no livro de 1998 *Neural Networks: Tricks of the Trade* de Genevieve Orr e Klaus-Robert Müller (Springer).

<sup>4</sup> Tal como “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, K. He *et al.* (2015).



A inicialização He considera somente o fan-in, não a média entre fan-in e fan-out, como na inicialização Xavier. Este também é o padrão para a função `variance_scaling_initializer()`, mas você pode mudar isso configurando o argumento `mode="FAN_AVG"`.

## Funções de Ativação Não Saturadas

Uma das ideias no artigo de 2010 de Glorot e Bengio dizia que os problemas dos gradientes *vanishing/exploding* aconteceram em parte devido a uma má escolha da função de ativação. Até então, a maioria das pessoas tinha assumido que, se a Mãe Natureza escolheu utilizar funções de ativação mais ou menos sigmoides em neurônios biológicos, esta deveria ser a escolha certa. Mas, acontece que outras funções de ativação se comportam muito melhor em redes neurais profundas, em particular a função de ativação ReLU, principalmente porque esta não fica saturada com valores positivos (e também porque sua computação é muito rápida).

Infelizmente, a função de ativação ReLU não é perfeita. Ela sofre de um problema conhecido como *dying ReLUs*: durante o treinamento, alguns neurônios efetivamente morrem, o que significa que deixam de exibir algo que não seja 0. Em alguns casos, parece que a metade dos neurônios da sua rede está morta, especialmente quando foi utilizada uma grande taxa de aprendizado. Se os pesos de um neurônio forem atualizados durante o treinamento de modo que a soma ponderada das entradas seja negativa, ele começará a exibir 0. Quando isso acontecer, será improvável que ele volte à vida útil uma vez que o gradiente da função ReLU será 0 quando sua entrada for negativa.

Para resolver este problema, utilizamos uma variante da função ReLU, como a *leaky ReLU*, definida como  $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$  (veja a Figura 11-2). O hiperparâmetro  $\alpha$  define o quanto a função “vaza”:  $z < 0$  é a inclinação da função, e é tipicamente configurada para 0,01. Esta pequena inclinação garante que as *leaky ReLUs* nunca morram; elas podem entrar em um coma profundo, mas têm a chance de eventualmente acordar. Um artigo recente (<https://goo.gl/B1xhKn>)<sup>5</sup> comparou diversas variantes da função de ativação ReLU e uma de suas conclusões foi que as variantes vazantes sempre superaram a rígida função de ativação ReLU. De fato, configurar  $\alpha=0,2$  (grande vazamento) parece resultar em um melhor desempenho do que  $\alpha=0,01$  (pequeno vazamento). Eles também avaliaram a *randomized leaky ReLU* (RReLU), em que  $\alpha$  é escolhido aleatoriamente em um determinado intervalo durante o treinamento, e corrigido para um valor médio durante o teste. Esta função também teve um bom desempenho e pareceu agir como um regularizador (reduzindo

<sup>5</sup> “Empirical Evaluation of Rectified Activations in Convolution Network”, B. Xu *et al.* (2015).

o risco de se sobreajustar ao conjunto de treinamento). Finalmente, eles também avaliaram a *parametric leaky ReLU* (PReLU), em que  $\alpha$  está autorizado a ser aprendido durante o treinamento (em vez de ser um hiperparâmetro, ele se torna um parâmetro que pode ser modificado por retropropagação como qualquer outro parâmetro). Isso supera o ReLU em conjuntos de dados com grandes imagens, mas em conjuntos de dados menores ele corre o risco de se sobreajustar ao conjunto de treinamento.



Figura 11-2. Leaky ReLU

Por último, mas não menos importante, um artigo de 2015 (<http://goo.gl/Sdl2P7>) por Djork-Arné Clevert *et al.*<sup>6</sup> propôs uma nova função de ativação chamada *exponential linear unit* (ELU), que superou todas as variantes ReLU em suas experiências: o tempo de treinamento foi reduzido e a rede neural ficou melhor no conjunto de testes. Ela está representada na Figura 11-3 e a Equação 11-2 mostra sua definição.

Equação 11-2. Função de ativação ELU

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

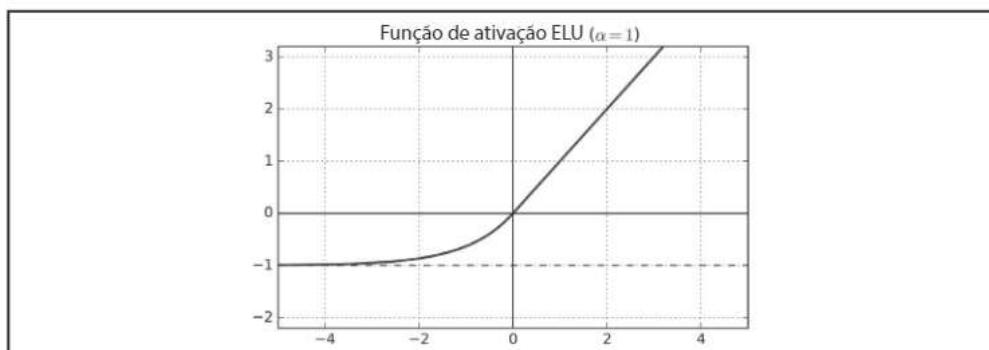


Figura 11-3. Função de ativação ELU

<sup>6</sup> “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELU)”, D. Clevert, T. Unterthiner, S. Hochreiter (2015).

Ela se parece muito com a função ReLU, mas tem algumas diferenças importantes:

- Primeiro, ela assume valores negativos quando  $z < 0$ , o que permite que a unidade tenha uma saída média mais próxima de 0, ajudando a aliviar o problema dos vanishing gradients, como discutido anteriormente. O hiperparâmetro  $\alpha$  define o valor de aproximação da função ELU quando  $z$  for um grande número negativo. Normalmente, ele é configurado em 1, mas você pode ajustá-lo como qualquer outro hiperparâmetro, se quiser;
- Segundo, ela tem um gradiente diferente de zero para  $z < 0$ , o que evita o problema das unidades mortas;
- Terceiro, a função é suave em todas as partes, incluindo quando está próxima de  $z = 0$ , o que ajuda a acelerar o Gradiente Descendente uma vez que não rebate tanto a esquerda e a direita de  $z = 0$ .

A principal desvantagem da função de ativação ELU é que seu cálculo é mais lento do que o da ReLU e suas variantes (devido ao uso da função exponencial), mas, durante o treinamento, isso é compensado pela taxa de convergência mais rápida. No entanto, no período de teste, uma rede ELU será mais lenta do que uma rede ReLU.



Então, qual função de ativação devemos utilizar para as camadas ocultas de redes neurais profundas? Embora a sua velocidade varie, no geral  $\text{ELU} > \text{leaky ReLU}$  (e suas variantes)  $> \text{ReLU} > \tanh > \text{logistic}$ . Se você se importa muito com o desempenho em tempo de execução, prefira as leaky ReLUs em relação às ELUs. Se você não quiser ajustar mais um outro hiperparâmetro, pode utilizar os valores padrão de  $\alpha$  sugeridos anteriormente (0,01 para a leaky ReLU e 1 para ELU). Se você tiver tempo livre e poder de computação, utilize a validação cruzada na avaliação de outras funções de ativação, em particular RReLU, se a sua rede estiver sobreajustada, ou PReLU, se tiver um conjunto de treinamento enorme.

O TensorFlow oferece a função `elu()` que pode ser utilizada na construção de sua rede neural. Configure o argumento `activation` desta forma ao chamar a função `dense()`:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

O TensorFlow não possui uma função predefinida para leaky ReLUs, mas é fácil definir:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = tf.layers.dense(X, n_hidden1, activation=leaky_relu, name="hidden1")
```

## Normalização em Lote

Embora a utilização da inicialização He juntamente com ELU (ou qualquer variante de ReLU) possa reduzir significativamente os problemas dos gradientes *vanishing/exploding* no início do treinamento, isto não garante que eles não retornem durante o treinamento.

Em um artigo de 2015 (<https://goo.gl/gA4GSP>),<sup>7</sup> Sergey Ioffe e Christian Szegedy propuseram uma técnica para endereçar os problemas dos gradientes *vanishing/exploding* chamada *Normalização em Lote* (BN, em inglês) e, de uma forma geral, o problema da mudança da distribuição das entradas de cada camada durante o treinamento à medida que mudam os parâmetros das camadas anteriores (que eles chamam de problema *Internal Covariate Shift*).

A técnica consiste em adicionar uma operação no modelo imediatamente antes da função de ativação de cada camada, ao simplesmente centralizar em zero e normalizar as entradas, e então escalar e deslocar o resultado utilizando dois novos parâmetros por camada (um para dimensionamento, outro para deslocamento). Em outras palavras, esta operação permite que o modelo aprenda a escala ideal e a média das entradas para cada camada.

O algoritmo precisa estimar a média e o desvio padrão das entradas para centralizar em zero e normalizá-las, que ele faz ao avaliar o desvio padrão e a média das entradas em relação ao minilote atual (daí o nome “Normalização em Lote”). Toda esta operação está resumida na Equação 11-3.

*Equação 11-3. Algoritmo de normalização em lote*

$$1. \quad \mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$2. \quad \sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2$$

$$3. \quad \hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$4. \quad \mathbf{z}^{(i)} = \gamma \hat{\mathbf{x}}^{(i)} + \beta$$

- $\mu_B$  é a média empírica avaliada em todo o minilote  $B$ ;
- $\sigma_B$  é o desvio padrão empírico também avaliado em relação a todo o minilote;

---

<sup>7</sup> “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, S. Ioffe and C. Szegedy (2015).

- $m_B$  é o número de instâncias no minilote;
- $\hat{\mathbf{x}}^{(i)}$  é a entrada normalizada e centralizada em zero;
- $\gamma$  é o parâmetro de escala para a camada;
- $\beta$  é o parâmetro de deslocamento (*offset*) para a camada;
- $\epsilon$  é um número minúsculo para evitar a divisão por zero (tipicamente  $10^{-5}$ ), chamado *smoothing term*;
- $\mathbf{z}^{(i)}$  é a saída da operação BN: é uma versão escalonada e deslocada das entradas.

No período de teste, não há minilote para calcular a média empírica e o desvio padrão, então utilizamos a média e o desvio padrão do conjunto de treinamento. Estes geralmente são calculados eficientemente durante o treinamento com a utilização de uma média móvel. No total, quatro parâmetros são aprendidos para cada camada normalizada em lote:  $\gamma$  (escala),  $\beta$  (deslocamento),  $\mu$  (média), e  $\sigma$  (desvio padrão).

Os autores demonstraram que esta técnica melhorou consideravelmente todas as redes neurais profundas experimentadas. O problema dos vanishing gradients foi reduzido drasticamente, ao ponto em que eles poderiam utilizar funções de ativação saturadas como a tanh e até mesmo a de ativação logística. As redes também eram muito menos sensíveis à inicialização do peso. Eles puderam utilizar taxas de aprendizado bem maiores, acelerando significativamente o processo de aprendizado. Especificamente, eles observam que “Aplicada a um modelo de classificação de imagem de última geração, a Normalização em Lote atinge a mesma precisão com 14 vezes menos etapas de treinamento e supera o modelo original por uma margem significativa. [...] Ao utilizarmos um ensemble de redes normalizadas por lotes, desenvolvemos ainda mais o melhor resultado publicado na classificação ImageNet: atingindo um erro de validação de 4,9% no top5 (e de 4,8% no teste de erro), superando a precisão dos avaliadores humanos.” Finalmente, como um brinde, a Normalização em Lote também funciona como um regulador, reduzindo a necessidade de outras técnicas de regularização (como o *dropout*, descrito mais adiante no capítulo).

No entanto, a normalização em lote adiciona um tanto de complexidade ao modelo (embora remova a necessidade de normalizar os dados de entrada, pois a primeira camada oculta cuidará disso, desde que seja normalizada em lote). Além disso, há uma penalidade em tempo de execução: a rede neural faz previsões mais lentas devido aos cálculos extras necessários em cada camada. Então, se você precisa que as previsões sejam rápidas, verifique o desempenho da ELU + inicialização He antes de brincar com a Normalização em Lote.



Você pode achar que o treinamento é bem lento no início, enquanto o Gradiente Descendente estiver procurando os melhores escalonamentos e deslocamentos para cada camada, mas ele acelera ao encontrar valores razoavelmente bons.

## Implementando a Normalização em Lote com o TensorFlow

O TensorFlow fornece uma função `tf.nn.batch_normalization()` que centra e normaliza as entradas, mas você deve calcular a média e o desvio padrão (baseado nos dados em minilotes durante o treinamento ou no conjunto completo de dados durante o teste, como discutido) e passá-los como parâmetros para esta função, além de lidar também com a criação dos parâmetros de escalonamento e deslocamento (e passá-los para esta função). Não é a abordagem mais conveniente, mas é possível. Em vez disso, utilize a função `tf.layers.batch_normalization()`, que cuida de tudo isso para você, como no código a seguir:

```
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

training = tf.placeholder_with_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn, training=training,
                                       momentum=0.9)
```

Acompanharemos este código. Até definirmos o placeholder `training`, as primeiras linhas são autoexplicativas: durante o treinamento, vamos configurá-lo para `True`, senão o padrão será `False`. Isto será utilizado para dizer à função `tf.layers.batch_normalization()` se ela deve utilizar a média atual dos minilotes e desvio padrão (durante o treinamento) ou a média de todo o conjunto de treinamento e desvio padrão (durante o teste).

Em seguida, alternamos camadas totalmente conectadas e camadas de normalização em lote: quando utilizamos a função `tf.layers.dense()`, as camadas totalmente conectadas são criadas, assim como fizemos no Capítulo 10. Observe que não especificamos nenhu-

ma função de ativação para as camadas totalmente conectadas porque queremos aplicar a função de ativação após cada camada de normalização de lote.<sup>8</sup> Com a utilização da função `tf.layers.batch_normalization()`, criamos as camadas de normalização em lote configurando seus parâmetros `training` e `momentum`. O algoritmo BN utiliza o *decaimento exponencial* para calcular as médias em execução, razão pela qual requer o parâmetro `momentum`: dado um novo valor  $v$ , a média em execução  $v$  é atualizada pela equação:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Um bom valor para o `momentum` seria próximo de 1 — por exemplo, 0,9, 0,99, ou 0,999 (coloque mais “9” para conjuntos maiores e minilotes menores).

Você deve ter notado que o código é bem repetitivo, com os mesmos parâmetros de normalização em lote se repetindo. Para evitar esta repetição, utilizamos a função `partial()` do módulo `functools` (parte da biblioteca padrão do Python), que cria um invólucro fino em torno de uma função e permite definir valores padrão para alguns parâmetros. A criação das camadas da rede no código anterior pode ser modificada desta forma:

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                             training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

Pode não parecer muito melhor do que antes neste pequeno exemplo, mas esse truque tornará seu código muito mais legível se você tiver 10 camadas e quiser utilizar a mesma função de ativação, o mesmo inicializador, o mesmo regularizador, etc., em todas as camadas.

O resto da fase de construção é o mesmo do Capítulo 10: definir a função de custo, criar um otimizador, instruí-lo a minimizar a função de custo, definir as operações de avaliação, criar um inicializador de variáveis, criar um `Saver` e assim por diante.

A fase de execução também é praticamente a mesma com duas exceções. Primeiro, durante o treinamento, você precisa definir o placeholder `training` para `True` sempre que executar uma operação que depende da camada `batch_normalization()`. Segundo,

---

<sup>8</sup> Muitos pesquisadores argumentam que é tão bom quanto, ou até melhor, colocar as camadas de normalização de lote após (e não antes) as ativações.

a função `batch_normalization()` cria algumas operações que devem ser avaliadas em cada etapa durante o treinamento a fim de atualizar as médias móveis (lembre-se de que essas médias móveis são necessárias na avaliação da média e desvio padrão do conjunto de treinamento). Essas operações são adicionadas automaticamente à coleção `UPDATE_OPS`, então apenas precisamos obter a lista de operações nessa coleção e executá-las em cada iteração do treinamento:

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run([training_op, extra_update_ops],
                    feed_dict={training: True, X: X_batch, y: y_batch})
            accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                    y: mnist.test.labels})
            print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Isso é tudo! Neste pequeno exemplo com apenas duas camadas, é improvável que a Normalização em Lote tenha um impacto muito positivo, mas para redes mais profundas pode fazer uma grande diferença.

## Gradient Clipping

Uma técnica popular para diminuir o problema das explosões dos gradientes é limitá-los durante a retropropagação, de modo que nunca excedam um limiar (isto é útil para redes neurais recorrentes, veja o Capítulo 14), e é chamada de *Gradient Clipping* (<http://goo.gl/dRDAaf>).<sup>9</sup> Em geral, as pessoas preferem a Normalização em Lote, mas ainda é útil conhecer o Gradient Clipping e como implementá-lo.

A função `minimize()` do otimizador cuida de calcular e aplicar os gradientes no TensorFlow, então você deve, em vez disso, chamar o primeiro método do otimizador `compute_gradients()`, criar uma operação para limitar os gradientes com a utilização da função `clip_by_value()` e, finalmente, criar uma operação para aplicar os limiares nos gradientes com a utilização do método `apply_gradients()` do otimizador:

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

---

<sup>9</sup> “On the difficulty of training recurrent neural networks” R. Pascanu *et al.* (2013).

Como de costume, você executaria então este `training_op` em cada etapa de treinamento e ele calculará os gradientes, os limitará entre -1.0 e 1.0 e os aplicará. O limiar é um hiperparâmetro ajustável.

## Reutilizando Camadas Pré-Treinadas

Geralmente, não é uma boa ideia iniciar o treinamento do zero em uma DNN muito grande: em vez disso, tente sempre encontrar uma rede neural existente que realize uma tarefa similar à que você está tentando implementar, e então reutilize as camadas inferiores desta rede. Esta técnica é denominada *transfer learning*, e não só acelerará consideravelmente o treinamento, mas também exigirá muito menos dados de treinamento.

Por exemplo, suponha que você tenha acesso a uma DNN que tenha sido treinada para classificar imagens em 100 categorias diferentes, incluindo animais, plantas, veículos e objetos do cotidiano. Agora, você deseja treinar uma DNN para classificar tipos específicos de veículos. Essas tarefas são muito semelhantes, então você deve tentar reutilizar partes da primeira rede (veja a Figura 11-4).

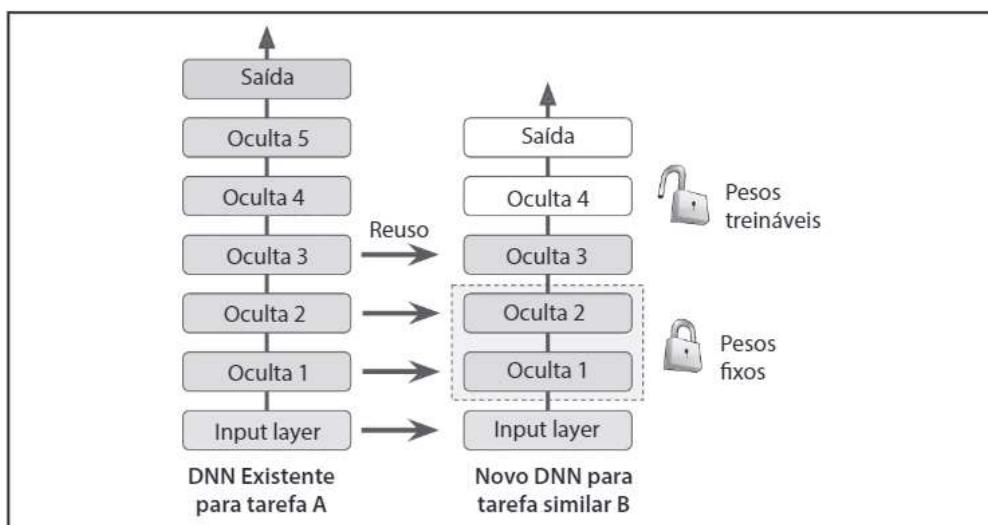


Figura 11-4. Reutilizando camadas pré-treinadas



Se as imagens de entrada da sua nova tarefa não tiverem o mesmo tamanho das usadas na tarefa original, você terá que adicionar uma etapa de pré-processamento para redimensioná-las para o tamanho esperado pelo modelo original. De forma mais geral, a transferência de aprendizado só funcionará bem se as entradas tiverem características de baixo nível semelhantes.

## Reutilizando um Modelo do TensorFlow

Se o modelo original foi treinado com o TensorFlow, você pode restaurá-lo e treiná-lo na nova tarefa. Utilizamos a função `import_meta_graph()` para importar as operações para o grafo padrão, como discutido no Capítulo 9, retornando um `Saver` que pode ser utilizado mais tarde para carregar o estado do modelo:

```
saver = tf.train.import_meta_graph("./my_model_final.ckpt.meta")
```

Lidaremos com as operações e os tensores necessários para o treinamento por meio dos métodos do grafo `get_operation_by_name()` e `get_tensor_by_name()`. O nome de um tensor é o nome da operação que o produz seguido por `:0` (ou `:1` se for a segunda saída, `:2` se for a terceira, e assim por diante):

```
X = tf.get_default_graph().get_tensor_by_name("X:0")
y = tf.get_default_graph().get_tensor_by_name("y:0")
accuracy = tf.get_default_graph().get_tensor_by_name("eval/accuracy:0")
training_op = tf.get_default_graph().get_operation_by_name("GradientDescent")
```

Teremos que explorar o grafo para encontrar os nomes das operações necessárias se o modelo pré-treinado não estiver bem documentado. Neste caso, podemos explorar o grafo utilizando o TensorBoard (para isso você deve primeiro exportar o grafo utilizando um `FileWriter`, conforme discutido no Capítulo 9) ou você pode utilizar o método `get_operations()` do grafo para listar todas as operações:

```
for op in tf.get_default_graph().get_operations():
    print(op.name)
```

Se você é o autor do modelo original, documentar e nomear as operações de forma bem clara facilita o trabalho das pessoas que reutilizarão seu modelo. Outra abordagem seria a criação de uma coleção com todas as operações importantes para que as pessoas possam ter conhecimento:

```
for op in (X, y, accuracy, training_op):
    tf.add_to_collection("my_important_ops", op)
```

Desta forma, as pessoas que reutilizam seu modelo poderão escrever simplesmente:

```
X, y, accuracy, training_op = tf.get_collection("my_important_ops")
```

Você pode, então, restaurar o estado do modelo e continuar o treinamento com seus próprios dados utilizando o `Saver`:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    [...] # treine o modelo com seus próprios dados
```

Como alternativa, se você tiver acesso ao código Python que criou o grafo original, você pode usá-lo em vez de `import_meta_graph()`.

No geral, você reutilizará apenas parte do modelo original, tipicamente as camadas inferiores. Se você utilizar o `import_meta_graph()` para restaurar o grafo, ele carregará todo o grafo original, mas nada o impede de ignorar as camadas nas quais você não tem interesse. Por exemplo, como mostrado na Figura 11-4, você poderia construir novas camadas (por exemplo, uma camada oculta e uma camada de saída) no topo de uma camada pré-treinada (por exemplo, camada oculta pré-treinada 3). Você também precisaria calcular a perda para esta nova saída e criar um otimizador para minimizá-la.

Se você tem acesso ao código Python do grafo pré-treinado, poderá reutilizar apenas as partes necessárias e cortar o resto. Neste caso, você precisa de um `Saver` para restaurar o modelo pré-treinado (especificando as variáveis que deseja restaurar, caso contrário o TensorFlow reclamará que os grafos não coincidem) e outro `Saver` para salvar o novo modelo. Por exemplo, o código a seguir restaura apenas as camadas ocultas 1, 2 e 3:

```
[...] # construa o novo modelo com as mesmas camadas ocultas 1-3 como antes

reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                               scope="hidden[123]") # expressão regular
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # para restaurar camadas 1-3

init = tf.global_variables_initializer() # para iniciar todas as variáveis, velhas e novas
saver = tf.train.Saver() # para salvar o novo modelo

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")
    [...] # treine o modelo
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

Primeiro, criamos o nosso modelo certificando-nos de copiar as camadas ocultas de 1 a 3 do modelo original. Depois, utilizando a expressão regular "hidden[123]", obtemos a lista de todas as variáveis nas camadas ocultas de 1 a 3 e criamos um dicionário que mapeará o nome de cada variável com seu nome no novo modelo (geralmente você manterá os mesmos nomes), no modelo original. Em seguida, criamos um `Saver` que restaurará apenas essas variáveis. Também criamos uma operação para inicializar todas as variáveis, não apenas as camadas 1 a 3 (antigas e novas) e um segundo `Saver` para salvar o novo modelo. Em seguida, iniciamos uma sessão, inicializamos todas as variáveis no modelo e restauramos os valores da variável das camadas 1 a 3 do modelo original. Finalmente, treinamos e salvamos o modelo na nova tarefa.



Quanto mais semelhantes forem as tarefas, mais camadas você reutilizará (começando com as camadas inferiores). Para tarefas muito semelhantes, você pode tentar manter todas as camadas ocultas e substituir apenas a camada de saída.

## Reutilizando Modelos de Outras Estruturas

Você precisará carregar manualmente os parâmetros do modelo se ele foi treinado com a utilização de outra estrutura (por exemplo, utilizar o código Theano se for treinado com Theano) e então os atribuir às variáveis apropriadas, trabalho que pode ser bem tedioso. Por exemplo, o código a seguir mostra como você copiará o peso e os vieses da primeira camada oculta de um modelo treinado em outra estrutura:

```
original_w = [...] # carregue os pesos do outro modelo
original_b = [...] # carregue os vieses do outro modelo
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu, name="hidden1")
[...] # Construa o resto do modelo

# Entenda os nós de atribuição para as variáveis ocultas 1
graph = tf.get_default_graph()
assign_kernel = graph.get_operation_by_name("hidden1/kernel/Assign")
assign_bias = graph.get_operation_by_name("hidden1/bias/Assign")
init_kernel = assign_kernel.inputs[1]
init_bias = assign_bias.inputs[1]

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init, feed_dict={init_kernel: original_w, init_bias: original_b})
    # [...] Treine o modelo em sua nova tarefa
```

Nesta implementação, primeiro carregamos o modelo pré-treinado na outra estrutura (não mostrada aqui) e extraímos os parâmetros do modelo que queremos reutilizar. Então construímos nosso modelo do TensorFlow como de costume. Em seguida, vem a parte complicada: cada variável do TensorFlow tem uma operação de atribuição associada que é utilizada em sua inicialização. Começamos por nos aprofundar nessas operações de atribuição, que têm o mesmo nome que a variável, além de `/Assign`. Também nos aprofundaremos na segunda entrada da operação de atribuição que corresponde ao valor que será atribuído à variável, que neste caso é o valor de inicialização da variável. Uma vez iniciada a sessão, executamos a operação de inicialização de sempre, mas, desta vez, alimentamos os valores que queremos para as variáveis que queremos reutilizar. Como alternativa, poderíamos criar novas operações de atribuição, placeholders e utilizá-los para definir os valores das variáveis após a inicialização. Mas por que criar novos nós no grafo quando tudo o que precisamos já está lá?

## Congelando as Camadas Inferiores

É provável que as camadas inferiores da primeira DNN tenham aprendido a detectar características de baixo nível em imagens para as reutilizarmos do jeito que estão, o que será útil em ambas as tarefas de classificação. Geralmente, é uma boa ideia “congelar” seus pesos ao

treinar a nova DNN: se os pesos da camada inferior forem fixos, será mais fácil treinar os pesos da camada superior (porque não terão que aprender com um alvo em movimento). Uma solução para congelar as camadas inferiores durante o treinamento é dar ao otimizador a lista de variáveis a serem treinadas, excluindo as das camadas inferiores:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                               scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

A primeira linha obtém a lista de todas as variáveis treináveis nas camadas ocultas 3 e 4 e na camada de saída, deixando de fora as variáveis nas camadas ocultas 1 e 2. Em seguida, fornecemos esta lista restrita de variáveis treináveis para a função `minimize()` do otimizador. É isso! As camadas 1 e 2 estão congeladas: elas não se moverão durante o treinamento e, por isso, geralmente são chamadas de *frozen layers*.

Outra opção seria adicionar uma camada `stop_gradient()` ao grafo. Qualquer camada abaixo dela será congelada:

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                            name="hidden1") # reutilizado, congelado
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                            name="hidden2") # reutilizado, congelado
    hidden2_stop = tf.stop_gradient(hidden2)
    hidden3 = tf.layers.dense(hidden2_stop, n_hidden3, activation=tf.nn.relu,
                            name="hidden3") # reutilizado, descongelado
    hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu,
                            name="hidden4") # novo!
    logits = tf.layers.dense(hidden4, n_outputs, name="outputs") # novo!
```

## Armazenamento em Cache das Camadas Congeladas

É possível armazenar em cache a saída da camada congelada mais alta para cada instância de treinamento uma vez que as camadas congeladas não mudam. Como o treinamento passa muitas vezes por todo o conjunto de dados, isso lhe dará um enorme ganho de velocidade, pois você só precisa passar por camadas congeladas uma vez por instância de treinamento (em vez de uma vez por época). Por exemplo, você pode primeiro executar todo o conjunto de treinamento pelas camadas inferiores (supondo que você tenha memória RAM suficiente) e, então, durante o treinamento, em vez de construir lotes de instâncias de treinamento, criar lotes de saídas da camada oculta 2 e alimentá-los para a operação de treinamento:

```

import numpy as np

n_batches = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")

    h2_cache = sess.run(hidden2, feed_dict={X: mnist.train.images})

    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(mnist.train.num_examples)
        hidden2_batches = np.array_split(h2_cache[shuffled_idx], n_batches)
        y_batches = np.array_split(mnist.train.labels[shuffled_idx], n_batches)
        for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
            sess.run(training_op, feed_dict={hidden2:hidden2_batch, y:y_batch})

    save_path = saver.save(sess, "./my_new_model_final.ckpt")

```

A última linha do loop de treinamento executa a operação de treinamento definida anteriormente (que não toca as camadas 1 e 2) e alimenta um lote de saídas da segunda camada oculta (assim como os alvos para esse lote). Como fornecemos a saída da camada oculta 2 ao TensorFlow, ele não tenta avaliá-la (ou qualquer nó que dependa dela).

## Ajustando, Descartando ou Substituindo as Camadas Superiores

A camada de saída do modelo original geralmente deve ser substituída, pois é provável que não seja útil para a nova tarefa e talvez nem tenha o número certo de saídas para tanto.

Da mesma forma, as camadas ocultas superiores do modelo original são menos propensas a ser tão úteis quanto as camadas inferiores, uma vez que as características de alto nível que são mais úteis para a nova tarefa podem diferir significativamente daquelas que foram mais úteis para a tarefa original. Desejamos encontrar o número certo de camadas para serem reutilizadas.

Primeiro, tente congelar todas as camadas copiadas, depois treine seu modelo e veja como ele funciona. Em seguida, tente descongelar uma ou duas das camadas ocultas mais altas para permitir que a retropropagação as altere e veja se o desempenho melhora. Quanto mais dados de treinamento você tiver, mais camadas poderá descongelar.

Caso não consiga um bom desempenho e ainda possua poucos dados de treinamento, tente descartar a(s) camada(s) oculta(s) superior(es) e congelar novamente todas as camadas ocultas restantes. Você pode iterar até encontrar o número certo de camadas para a reutilização. Caso possua muitos dados de treinamento, pode tentar substituir as camadas ocultas superiores em vez de descartá-las e até mesmo adicionar mais camadas ocultas.

## Zoológicos de Modelos

Onde você pode encontrar uma rede neural treinada para uma tarefa semelhante à que você quer executar? O primeiro lugar a pesquisar é obviamente o seu próprio catálogo de modelos. Esta é uma boa razão para salvar todos os seus modelos e organizá-los para recuperá-los mais facilmente. Outra opção é buscar em um *Zoológico de Modelos*. Muitas pessoas treinam modelos de Aprendizado de Máquina para várias tarefas e liberam seus modelos pré-treinados para o público.

O TensorFlow tem seu próprio zoológico de modelos disponível em <https://github.com/tensorflow/models>, que contém a maioria das redes de classificação de imagens de última geração como a VGG, Inception e ResNet (veja o Capítulo 13, e verifique o diretório *models/slim*), incluindo os códigos, os modelos pré-treinados e as ferramentas para baixar conjuntos de dados de imagens populares.

Outro zoológico de modelos popular é o Caffe's Model Zoo (<https://goo.gl/XI02X3>). Ele também possui muitos modelos de visão computacional (por exemplo, LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, Inception) treinados em vários conjuntos de dados (por exemplo, ImageNet, Places Database, CIFAR10, etc.). Saumitro Dasgupta projetou um conversor que está disponível em <https://github.com/ethereon/caffe-tensorflow>.

## Pré-treinamento Não Supervisionado

Suponha que você queira enfrentar uma tarefa complexa para a qual não tenha muitos dados rotulados de treinamento e infelizmente não consiga encontrar um modelo treinado em uma tarefa similar. Não perca a esperança! Em primeiro lugar, você deve naturalmente tentar reunir mais dados rotulados de treinamento, mas se isso for muito difícil, ou muito caro, você ainda pode realizar um *pré-treinamento não supervisionado* (veja a Figura 11-5). Ou seja, se você tem uma abundância de dados de treinamento não rotulados, pode tentar treinar as camadas uma a uma, começando com a camada inferior e então subir, com a utilização de um algoritmo detector de características sem supervisão como as *Máquinas Restritas de Boltzmann* (RBMs, ver Apêndice E) ou autoencoders (veja o Capítulo 15). Cada camada é treinada na saída das camadas previamente treinadas (todas as camadas estão congeladas, exceto a que está sendo treinada). Uma vez que todas as camadas foram treinadas dessa forma, você pode ajustar a rede utilizando o aprendizado supervisionado (isto é, com retropropagação).

Este é um processo bastante longo e tedioso, mas muitas vezes funciona bem; na verdade, é essa técnica que Geoffrey Hinton e sua equipe utilizaram em 2006 e que levou ao renascimento das redes neurais e ao sucesso do Aprendizado Profundo. Até 2010, o

pré-treinamento não supervisionado (utilizando-se RBMs) era a norma para redes profundas, e foi somente após o problema do vanishing gradient ter sido amenizado que se tornou mais comum treinar DNNs com a utilização da retropropagação. No entanto, o pré-treinamento não supervisionado (com a utilização de autoencoders em vez de RBMs) ainda é uma boa opção quando temos uma tarefa complexa para resolver na ausência de um modelo similar que possa ser reutilizado, ou poucos dados de treinamento rotulados com muitos dados de treinamento não rotulados.

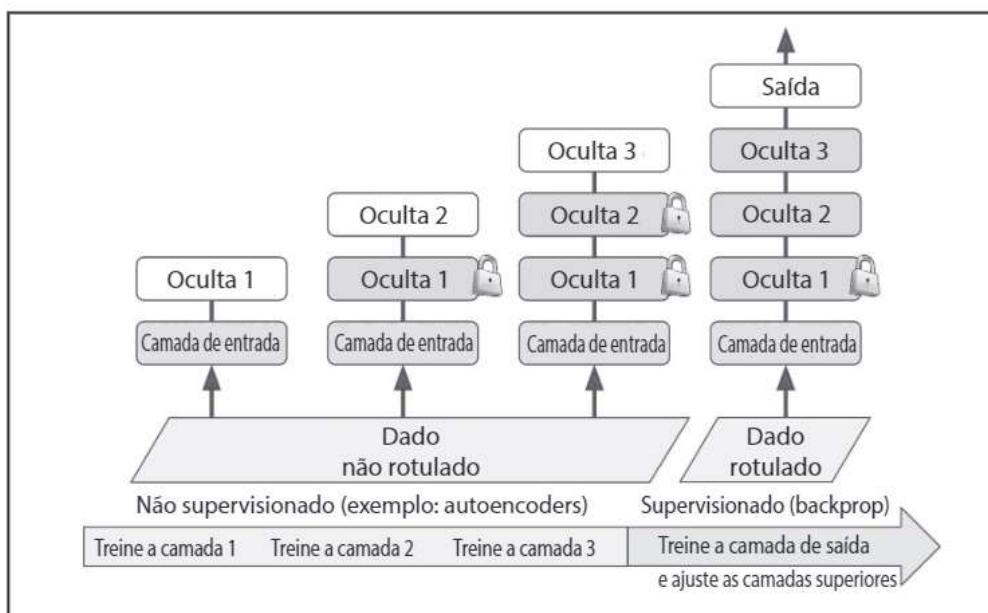


Figura 11-5. Pré-treinamento sem supervisão

## Pré-treinamento em uma Tarefa Auxiliar

Uma última opção seria treinar uma primeira rede neural em uma tarefa auxiliar para a qual você pode facilmente obter ou gerar dados rotulados e depois reutilizar as camadas inferiores dessa rede para sua tarefa real. As primeiras camadas inferiores da rede neural aprenderão a detectar características que provavelmente serão reutilizáveis pela segunda rede neural.

Por exemplo, se você quer criar um sistema para reconhecer rostos e tem somente algumas fotos de cada indivíduo, claramente isto não é suficiente para treinar um bom classificador. Não seria prático reunir centenas de imagens de cada pessoa. No entanto, você pode reunir muitas fotos aleatórias de pessoas na internet e treinar a primeira rede neural para detectar se duas fotos diferentes apresentam ou não a mesma pessoa. Essa

rede teria bons detectores de características de rostos, portanto, reutilizar suas camadas inferiores permitiria que você formasse um bom classificador de rostos utilizando poucos dados de treinamento.

Normalmente, reunir exemplos de treinamento não rotulados é barato, mas rotulá-los é bem caro. Uma técnica comum nessa situação é rotular todos os seus exemplos de treinamento como “bom”, em seguida gerar várias novas instâncias de treinamento corrompendo as boas e rotulando essas instâncias corrompidas como “ruim”. Então você pode treinar uma primeira rede neural a fim de classificar instâncias como boas ou ruins. Por exemplo, você pode baixar milhões de frases, rotulá-las como “boas” e, então, alterar aleatoriamente uma palavra em cada frase e rotular as frases resultantes como “ruim”. Se uma rede neural consegue dizer que “*O cachorro dorme*” é uma boa sentença, mas “*O cachorro eles*” é ruim, provavelmente ela sabe muito sobre linguagem. Muitas tarefas do processamento de idiomas podem se beneficiar com a reutilização de suas camadas inferiores.

Outra abordagem seria treinar uma primeira rede para produzir uma pontuação para cada instância de treinamento e utilizar uma função de custo que garanta que a pontuação de uma boa instância seja maior que a pontuação de uma ruim, com pelo menos alguma margem, o que é chamado de *max margin learning*.

## Otimizadores Velozes

Treinar uma enorme rede neural profunda pode ser um processo lentíssimo. Até agora, vimos quatro maneiras de acelerar o treinamento (e alcançar uma solução melhor): aplicar uma boa estratégia de inicialização para os pesos da conexão, utilizar uma boa função de ativação, utilizar a Normalização em Lote e reutilizar partes de uma rede pré-treinada. Outro grande impulso na velocidade vem do uso de um otimizador mais veloz do que o otimizador regular do Gradiente Descendente. Nesta seção, apresentaremos os mais populares: otimização Momentum, Gradiente Acelerado de Nesterov, AdaGrad, RMSProp, e finalmente otimização Adam.

### Otimização Momentum

Imagine uma bola de boliche滚着在光滑的表面上：其滚动将缓慢地开始，但很快会获得足够的冲力，直到达到最终速度（如果有摩擦或空气阻力）。这就是Boris Polyak在1964年提出的“*Momentum*”优化方法背后的简单想法（<https://goo.gl/F1SE8c>）。<sup>10</sup> 在

<sup>10</sup> “Some methods of speeding up the convergence of iteration methods”, B. Polyak (1964).

contraste, o Gradiente Descendente regular dará pequenos passos regulares em direção à inclinação, então demorará muito mais tempo para chegar ao fundo.

Lembre-se que o Gradiente Descendente simplesmente atualiza os pesos  $\theta$  subtraindo diretamente o gradiente da função de custo  $J(\theta)$  com relação aos pesos ( $\nabla_{\theta}J(\theta)$ ) multiplicados pela taxa de aprendizado  $\eta$ . A equação é:  $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$ . Ele não se importa com os gradientes anteriores e irá bem devagar se o gradiente local for pequeno.

A otimização Momentum se preocupa muito com os gradientes anteriores: a cada iteração, ela subtrai o gradiente local do *vetor momentum*  $m$  (multiplicado pela taxa de aprendizado  $\eta$ ) e atualiza os pesos adicionando esse vetor momentum (veja a Equação 11-4). Em outras palavras, o gradiente é utilizado como aceleração, não como velocidade. Para simular algum tipo de mecanismo de fricção e evitar que o momentum cresça muito, o algoritmo introduz um novo hiperparâmetro  $\beta$  chamado simplesmente de momentum que deve ser ajustado entre 0 (alta fricção) e 1 (sem fricção). Um valor típico do momentum é 0,9.

#### *Equação 11-4. Algoritmo Momentum*

1.  $m \leftarrow \beta m - \eta \nabla_{\theta}J(\theta)$
2.  $\theta \leftarrow \theta + m$

Verificamos facilmente que, se o gradiente permanece constante, a velocidade terminal (ou seja, o tamanho máximo das atualizações do peso) será igual ao gradiente multiplicado pela taxa de aprendizado  $\eta$  multiplicada por  $\frac{1}{1-\beta}$  (ignorando o sinal). Por exemplo, se  $\beta = 0,9$ , então a velocidade terminal é igual a 10 vezes o gradiente vezes a taxa de aprendizado e a otimização Momentum acaba 10 vezes mais rápida do que Gradiente Descendente, permitindo que ela escape de platôs com muito mais rapidez do que o Gradiente Descendente. Em particular, vimos no Capítulo 4 que, quando as entradas têm escalas muito diferentes da função de custo, elas têm a aparência de uma tigela alongada (veja a Figura 4-7). O Gradiente Descendente desce a encosta íngreme muito rapidamente, mas demora muito para descer o vale. Em contrapartida, a otimização Momentum rolará para o fundo do vale cada vez mais rápido até chegar ao seu destino (o ótimo). As camadas superiores geralmente têm entradas em redes neurais profundas com escalas muito diferentes que não utilizam a Normalização em Lote, então utilizar a otimização Momentum ajuda muito, além de poder auxiliar a superar o local optimum.



Devido ao momentum, o otimizador pode ultrapassar um pouco, então voltar, ultrapassar novamente, e oscilar assim por muitas vezes antes de se estabilizar no mínimo. Esta é uma das razões pelas quais é bom termos um pouco de fricção no sistema: ele se livra dessas oscilações e assim acelera a convergência.

É muito fácil implementar a otimização Momentum no TensorFlow: substitua o `GradientDescentOptimizer` pelo `MomentumOptimizer` e, então, relaxe e aproveite!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

Sua única desvantagem é que ela ainda adiciona outro hiperparâmetro que deve ser configurado. No entanto, o valor de 0,9 do momentum geralmente funciona bem na prática e quase sempre é mais rápido do que o Gradiente Descendente.

## Gradiente Acelerado de Nesterov

Uma pequena variante da otimização Momentum proposta por Yurii Nesterov em 1983 (<https://goo.gl/V01IvD>),<sup>11</sup> quase sempre é mais rápida do que a otimização Momentum normal. A ideia da *otimização Momentum de Nesterov*, ou *Gradiente Acelerado de Nesterov* (NAG, em inglês) é medir o gradiente da função de custo, não na posição local, mas ligeiramente à frente, na direção do momentum (ver Equação 11-5). A única diferença da otimização Momentum normal é que o gradiente é medido em  $\theta + \beta\mathbf{m}$  em vez de  $\theta$ .

*Equação 11-5. Algoritmo Gradiente Acelerado de Nesterov*

1.  $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2.  $\theta \leftarrow \theta + \mathbf{m}$

Este pequeno ajuste funciona porque em geral o vetor momentum estará apontando na direção certa (ou seja, em direção ao optimum), então ele será um pouco mais preciso quando utilizarmos o gradiente medido ligeiramente mais adiante nessa direção em vez de usar o gradiente na posição original, como você pode ver na Figura 11-6 (em que  $\nabla_1$  representa o gradiente da função de custo medida no ponto inicial  $\theta$ , e  $\nabla_2$  representa o gradiente no ponto localizado em  $\theta + \beta\mathbf{m}$ ). Como podemos ver, a atualização Nesterov termina um pouco mais próxima do optimum e, depois de um tempo, essas pequenas melhorias se somam e o NAG acaba sendo significativamente mais rápido do que a otimização Momentum. Além disso, note que, quando o momentum empurra os pesos por um vale,  $\nabla_1$  continua a avançar ainda mais, enquanto  $\nabla_2$  puxa para trás em direção ao fundo do vale, ajudando a reduzir as oscilações e assim convergir mais rapidamente.

Em comparação com a otimização Momentum, o NAG quase sempre acelerará o treinamento. Para utilizá-la, ao criar o `MomentumOptimizer` configure `use_nesterov=True`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

---

<sup>11</sup> “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k<sup>2</sup>)”, Yurii Nesterov (1983).

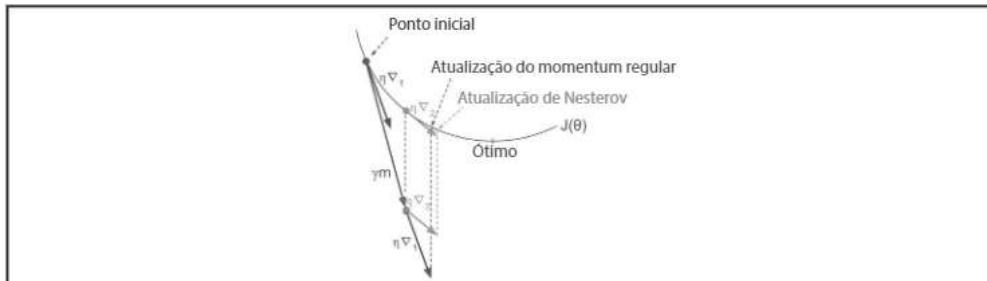


Figura 11-6. Otimização Regular versus otimização Momentum de Nesterov

## AdaGrad

Considere novamente o problema da tigela alongada: o Gradiente Descendente começa a descer a encosta mais íngreme rapidamente para depois descer mais lentamente em direção ao fundo do vale. Seria bom se o algoritmo pudesse detectar isso antecipadamente e corrigir sua direção para apontar um pouco mais para o global optimum.

O algoritmo *AdaGrad* (<http://goo.gl/4Tyd4j>)<sup>12</sup> consegue isso escalonando o vetor gradiente na dimensão mais íngreme (veja a Equação 11-6):

Equação 11-6. Algoritmo AdaGrad

1.  $s \leftarrow s + \nabla_{\theta}J(\theta) \otimes \nabla_{\theta}J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta) \oslash \sqrt{s + \epsilon}$

O primeiro passo acumula o quadrado dos gradientes no vetor  $s$  (o símbolo  $\otimes$  representa a multiplicação elemento por elemento). Esta forma vetorializada é equivalente a calcular  $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$  para cada elemento  $s_i$  do vetor  $s$ ; em outras palavras, cada  $s_i$  acumula os quadrados da derivada parcial da função de custo com relação ao parâmetro  $\theta_i$ . Se a função de custo for íngreme ao longo da  $i$ -ésima dimensão, então  $s_i$  ficará cada vez maior a cada iteração.

O segundo passo é quase idêntico ao Gradiente Descendente, mas com uma grande diferença: o vetor gradiente é reduzido por um fator de  $\sqrt{s + \epsilon}$  (o símbolo  $\oslash$  representa a divisão elemento por elemento, e  $\epsilon$  é um termo de suavização para evitar a divisão por zero, tipicamente ajustada para  $10^{-10}$ ). Esta forma vetorializada é equivalente a calcular  $\theta_i \leftarrow \theta_i - \eta \partial J(\theta) / \partial \theta_i / \sqrt{s_i + \epsilon}$  para todos os parâmetros  $\theta_i$  (simultaneamente).

Em suma, este algoritmo degrada a taxa de aprendizado, mas com mais rapidez para dimensões íngremes do que para dimensões com declividade mais suave, o que é chamado

<sup>12</sup> “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, J. Duchi *et al.* (2011).

de taxa de aprendizado adaptativa. Ele ajuda a apontar mais diretamente as atualizações resultantes em direção ao global optimum (veja a Figura 11-7), além de ter o benefício adicional de requerer muito menos ajuste da taxa de aprendizado do hiperparâmetro  $\eta$ .

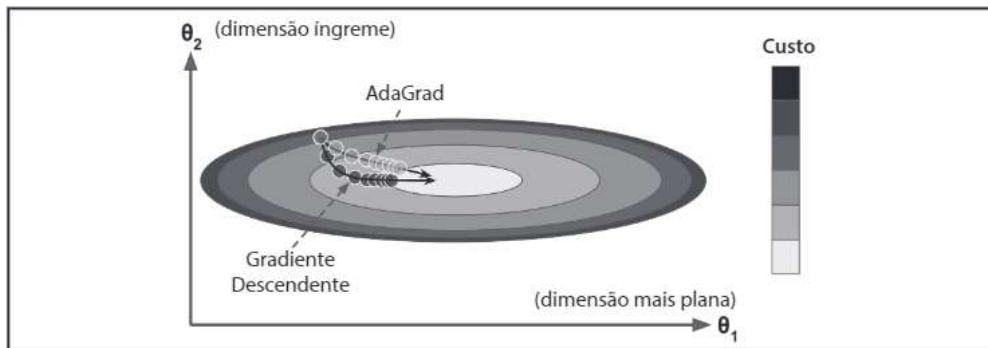


Figura 11-7. AdaGrad versus Gradiente Descendente

AdaGrad geralmente funciona bem para problemas quadráticos simples mas, infelizmente, muitas vezes ele para muito cedo quando treinamos redes neurais. A taxa de aprendizado diminui tanto que o algoritmo acaba parando completamente antes de atingir o global optimum. Embora o TensorFlow tenha o `AdagradOptimizer`, você não deve utilizá-lo no treinamento de redes neurais profundas (embora seja eficiente para tarefas mais simples, como a Regressão Linear).

## RMSProp

Embora o AdaGrad fique devagar muito rapidamente e nunca convirja para o global optimum, o algoritmo *RMSProp*<sup>13</sup> corrige isto acumulando somente os gradientes das iterações mais recentes (em vez de todos os gradientes desde o início do treinamento) por meio do decaimento exponencial no primeiro passo (veja a Equação 11-7).

Equação 11-7. Algoritmo RMSProp

1.  $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

A taxa de degradação  $\beta$  é ajustada para 0,9. Sim, mais uma vez é um novo hiperparâmetro, mas esse valor padrão geralmente funciona bem, então você não precisa sincronizá-lo.

<sup>13</sup> Este algoritmo foi criado por Tijmen Tieleman e Geoffrey Hinton em 2012 e apresentado por Geoffrey Hinton em sua aula no Coursera sobre redes neurais (slides: <http://goo.gl/RsQei>; vídeo: <https://goo.gl/XUbIyJ>). Surpreendentemente, uma vez que os autores não escreveram um artigo para descrevê-lo, os pesquisadores frequentemente citam “slide 29 da aula 6” em seus artigos.

Como era de se esperar, o TensorFlow tem uma classe `RMSPropOptimizer`:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Exceto em problemas muito simples, esse otimizador quase sempre funciona muito melhor do que o AdaGrad. Na verdade, foi o algoritmo de otimização preferido de muitos pesquisadores até surgir a Otimização de Adam.

## Otimização Adam

*Adam* (<https://goo.gl/Un8Axa>),<sup>14</sup> que significa *estimativa de momento adaptativo [adaptive moment estimation]*, combina as ideias de otimização Momentum e RMSProp: assim como a otimização Momentum, ela acompanha uma média exponencialmente decadente de gradientes passados, e, assim como o RMSProp, controla uma média exponencialmente decadente de gradientes quadrados passados (veja a Equação 11-8).<sup>15</sup>

*Equação 11-8. Algoritmo Adam*

1.  $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2.  $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3.  $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4.  $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5.  $\theta \leftarrow \theta + \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

- $t$  representa o número da iteração (começando em 1).

Se você visualizar as etapas 1, 2 e 5, notará a semelhança de Adam com a otimização Momentum e RMSProp. A única diferença é que o passo 1 calcula a média exponencialmente decadente em vez da soma exponencialmente decadente, mas essas são equivalentes, exceto por um fator constante (a média decadente é apenas  $1 - \beta_1$  vezes a soma decadente). Os passos 3 e 4 são detalhes técnicos: uma vez que  $\mathbf{m}$  e  $\mathbf{s}$  são inicializados em 0, serão polarizados em direção a 0 no início do treinamento, então essas duas etapas ajudarão a impulsionar  $\mathbf{m}$  e  $\mathbf{s}$  neste momento inicial.

---

<sup>14</sup> "Adam: A Method for Stochastic Optimization", D. Kingma, J. Ba (2015).

<sup>15</sup> Estas são estimativas da variância média (não centralizada) dos gradientes. A média é frequentemente chamada de *primeiro momento*, enquanto a variância é frequentemente chamada de *segundo momento*, daí o nome do algoritmo.

O hiperparâmetro  $\beta_1$  de decaimento de momentum é tipicamente inicializado em 0,9, enquanto o hiperparâmetro  $\beta_2$  de decaimento escalonado é inicializado em 0,999. Como anteriormente, o termo de suavização  $\epsilon$  é inicializado com um número minúsculo, como  $10^{-8}$ . Estes são os valores padrão para a classe `AdamOptimizer` do TensorFlow, então você pode utilizar:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

De fato, como o Adam é um algoritmo de taxa de aprendizado adaptativa (como o AdaGrad e RMSProp) ele requer menos ajuste do hiperparâmetro de taxa de aprendizado  $\eta$ , portanto, você pode utilizar o valor padrão  $\eta = 0,001$ , fazendo com que a utilização de Adam seja ainda mais fácil do que a do Gradiente Descendente.



Este livro inicialmente recomendava o uso da otimização de Adam porque ela era considerada mais rápida e melhor do que outros métodos. No entanto, um artigo de 2017 (<https://goo.gl/NAkWla>)<sup>16</sup> de Ashia C. Wilson *et al.* mostrou que os métodos de otimização adaptativa (ou seja, AdaGrad, RMSProp e otimização de Adam) podem levar a soluções que generalizam mal em alguns conjuntos de dados. Então, é melhor manter a otimização do Momentum ou Gradiente Acelerado de Nesterov por enquanto até que os pesquisadores tenham uma melhor compreensão desta questão.

Todas as técnicas de otimização discutidas até agora somente se basearam nas *derivadas parciais de primeira ordem* (*Jacobians*). A literatura de otimização contém algoritmos surpreendentes com base nas *derivadas parciais de segunda ordem* (as *Hessianas*), mas, infelizmente, é muito difícil aplicá-los às redes neurais profundas porque há  $n^2$  Hessianas por saída (sendo que  $n$  é o número de parâmetros), ao contrário de apenas  $n$  Jacobianas por saída. Como as DNNs normalmente possuem dezenas de milhares de parâmetros, os algoritmos de otimização de segunda ordem muitas vezes nem cabem na memória, e, mesmo quando o fazem, o cálculo para as Hessianas é muito lento.

## Treinando Modelos Esparsos

Todos os algoritmos de otimização que acabamos de apresentar produzem modelos densos, o que significa que a maioria dos parâmetros será diferente de zero. Se você precisa de um modelo muito rápido em tempo de execução, ou se precisar de menos memória, talvez seja melhor usar um modelo esparso.

<sup>16</sup> "The Marginal Value of Adaptive Gradient Methods in Machine Learning," A. C. Wilson *et al.* (2017).

Uma maneira trivial de conseguir isso é treinar o modelo como de costume e depois livrar-se dos pesos muito pequenos (configure-os para 0).

Outra opção é aplicar uma forte regularização  $\ell_1$  durante o treinamento, pois ele empurra o otimizador para zerar o máximo de pesos possível (como discutido no Capítulo 4 sobre a Regressão Lasso).

No entanto, em alguns casos, essas técnicas podem ser insuficientes. Uma última opção seria aplicar *Dual Averaging*, chamado *Follow The Regularized Leader* (FTRL), uma técnica proposta por Yurii Nesterov (<https://goo.gl/xSQD4C>).<sup>17</sup> Quando utilizada com a regularização  $\ell_1$ , esta técnica leva a modelos muito esparsos. O TensorFlow implementa uma variante FTRL chamada *FTRL-Proximal* (<https://goo.gl/bxme2B>)<sup>18</sup> na classe `FTRL Optimizer`.

## Cronograma da Taxa de Aprendizado

Pode ser complicado encontrar uma boa taxa de aprendizado. Configurando-a alta demais, o treinamento pode, na verdade, divergir (como discutimos no Capítulo 4). Se configurá-la muito baixa, o treinamento acabará por convergir para o optimum, mas demorará muito. Ao configurá-la um pouco alta, ela avançará muito rapidamente no início, mas acabará dançando em torno do optimum, nunca se estabelecendo (a menos que você utilize um algoritmo de otimização da taxa de aprendizado adaptativo como o AdaGrad, RMSProp ou Adam, mas, mesmo assim, pode demorar para se estabelecer). Se você tiver um orçamento computacional limitado, talvez seja necessário interromper o treinamento antes que ele possa convergir corretamente, resultando em uma solução subótima (veja a Figura 11-8).

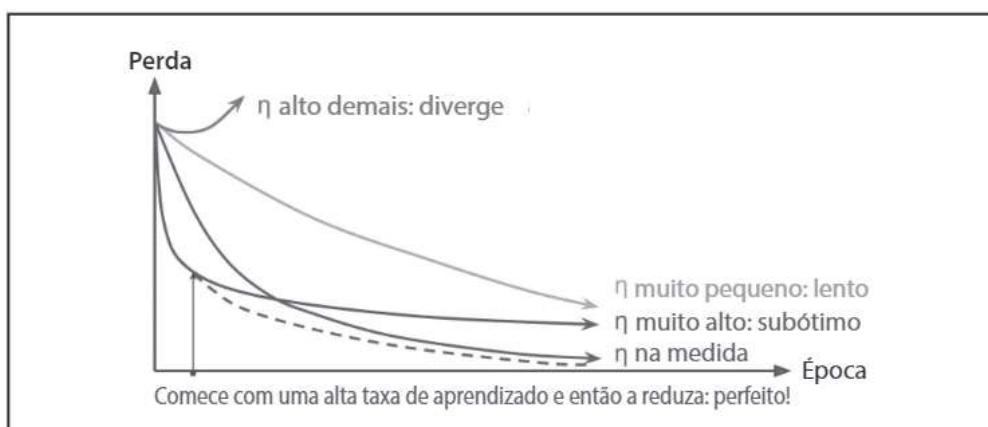


Figura 11-8. Curvas de aprendizado para várias taxas de aprendizado  $\eta$

<sup>17</sup> “Primal-Dual Subgradient Methods for Convex Problems,” Yurii Nesterov (2005).

<sup>18</sup> “Ad Click Prediction: a View from the Trenches,” H. McMahan et al. (2013).

Talvez seja possível encontrar uma taxa de aprendizado muito boa ao treinar sua rede várias vezes durante apenas algumas épocas utilizando diversas taxas de aprendizado e comparando suas curvas. A taxa de aprendizado ideal aprenderá rapidamente e convergirá para uma boa solução.

No entanto, tem uma opção melhor do que a taxa de aprendizado constante: se você começar com uma alta taxa de aprendizado e em seguida reduzi-la quando deixar de fazer progresso rápido, alcançará mais rapidamente uma boa solução do que com a taxa constante ideal de aprendizado. Existem muitas estratégias diferentes para reduzir a taxa de aprendizado durante o treinamento, chamadas *cronogramas de aprendizado* (introduzimos brevemente esse conceito no Capítulo 4), as mais comuns são:

#### *Taxa de aprendizado constante por partes predeterminadas*

Por exemplo, configure a taxa de aprendizado para  $\eta_0 = 0,1$ , então para  $\eta_1 = 0,001$  após 50 épocas. Embora esta solução possa funcionar muito bem, muitas vezes requererá que a alteremos para descobrirmos as taxas de aprendizado e quando utilizá-las corretamente.

#### *Agendamento de Desempenho*

Meça o erro de validação a cada  $N$  passos (assim como para a parada antecipada) e reduza a taxa de aprendizado por um fator  $\lambda$  quando o erro parar de cair.

#### *Agendamento Exponencial*

Defina a taxa de aprendizado para uma função do número da iteração  $t$ :  $\eta(t) = \eta_0 10^{-t/r}$ . Isso funciona muito bem, mas requer ajuste em  $\eta_0$  e  $r$ . A taxa de aprendizado cairá com um fator de 10 a cada  $r$  passos.

#### *Agendamento de Energia*

Defina a taxa de aprendizado em  $\eta(t) = \eta_0 (1 + t/r)^{-c}$ . O hiperparâmetro  $c$  normalmente é ajustado em 1. É parecido com o agendamento exponencial, mas a taxa de aprendizado cai bem mais lentamente.

Um artigo de 2013 (<http://goo.gl/Hu6Zyq>)<sup>19</sup> escrito por Andrew Senior *et al.* comparou o desempenho de alguns dos cronogramas de aprendizado mais populares para o reconhecimento da fala ao treinar redes neurais profundas com a utilização da otimização Momentum. Os autores concluíram que, nesta configuração, tanto o agendamento de desempenho quanto o agendamento exponencial tiveram um bom desempenho, mas preferiram o agendamento exponencial porque é mais fácil de implementar e de ajustar, e convergiu um pouco mais rápido para a solução ótima.

---

19 “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition”, A. Senior *et al.* (2013).

Implementar um cronograma de aprendizado com o TensorFlow é fácil:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False, name="global_step")
learning_rate = tf.train.exponential_decay(initial_learning_rate, global_step,
                                            decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

Depois de configurar os valores dos hiperparâmetros, criamos uma variável não treinável `global_step` (inicializada em 0) para acompanhar o número atual da iteração de treinamento. Então definimos uma taxa de aprendizado de decaimento exponencial (com  $\eta_0 = 0,1$  e  $r = 10.000$ ) usando a função `exponential_decay()` do TensorFlow. A seguir, criamos um otimizador (neste exemplo, um `MomentumOptimizer`) usando esta taxa de aprendizado decadente. Finalmente, criamos a operação de treinamento chamando o método `minimize()` do otimizador, desde que passemos a variável `global_step`, ele cuidará de incrementá-la. É isso!

Já que AdaGrad, RMSProp e a otimização Adam reduzem automaticamente a taxa de aprendizado durante o treinamento, não é necessário adicionar um cronograma de aprendizado extra. O uso do decaimento exponencial ou programação de desempenho pode acelerar consideravelmente a convergência para outros algoritmos de otimização.

## Evitando o Sobreajuste Por Meio da Regularização

Com quatro parâmetros, eu coloco um elefante e com cinco eu consigo fazê-lo balançar sua tromba.

—John von Neumann, *citado por Enrico Fermi em Nature* 427

As redes neurais profundas geralmente possuem dezenas de milhares de parâmetros, às vezes até milhões. Com tantos parâmetros, a rede tem uma incrível quantidade de liberdade e pode conter uma grande variedade de conjuntos de dados complexos. Mas essa grande flexibilidade também significa que está propensa a sobreajustar o conjunto de treinamento.

Com milhões de parâmetros, você consegue colocar todo o zoológico. Nesta seção, apresentaremos algumas das técnicas de regularização mais populares para as redes neurais e como implementá-las com o TensorFlow: parada antecipada, regularização  $\ell_1$  e  $\ell_2$ , dropout, regularização max-norm e aumento de dados.

## Parada Antecipada

Para evitar o sobreajuste do conjunto de treinamento, uma excelente solução é a parada antecipada (introduzida no Capítulo 4): basta interromper o treinamento quando seu desempenho no conjunto de validação começa a cair.

Uma forma de implementar isso com o TensorFlow seria avaliar o modelo em um conjunto de validação posto em intervalos regulares (por exemplo, a cada 50 etapas) e salvar um snapshot “vencedor” se ele superar os snapshots vencedores anteriores. Conte o número de etapas desde o último “vencedor” salvo e interrompa o treinamento quando esse número atingir um limite (por exemplo, 2.000 passos). Em seguida, restaure o último snapshot “vencedor”.

Embora a parada antecipada funcione muito bem na prática, você pode obter um desempenho bem melhor de sua rede combinando-a com outras técnicas de regularização.

## Regularização $\ell_1$ e $\ell_2$

Você pode utilizar a regularização  $\ell_1$  e  $\ell_2$  para restringir os pesos da conexão de uma rede neural (mas não suas polarizações), assim como você fez no Capítulo 4 para os modelos lineares mais simples.

Uma maneira de fazer isso é adicionar os termos de regularização adequados à sua função de custo ao utilizar o TensorFlow. Por exemplo, digamos que você tem apenas uma camada oculta com peso  $W_1$  e uma camada oculta de saída com peso  $W_2$ , então você pode aplicar a regularização  $\ell_1$  desta forma:

```
[...] # construa a rede neural
W1 = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("outputs/kernel:0")

scale = 0.001 # hiperparâmetro de regularização l1

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
    reg_losses = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
    loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

No entanto, essa abordagem não será muito conveniente se houver muitas camadas. Felizmente, o TensorFlow oferece uma opção melhor. Muitas funções que criam variáveis (como a `get_variable()` ou `tf.layers.dense()`) aceitam um argumento `*_regularizer`

para cada variável criada (por exemplo, `kernel_regularizer`). Você pode passar qualquer função que toma pesos como argumento e retorna a perda correspondente de regularização. As funções `l1_regularizer()`, `l2_regularizer()`, e `l1_l2_regularizer()` retornam essas funções. O código a seguir junta tudo isso:

```
my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                           name="outputs")
```

Este código cria uma rede neural com duas camadas ocultas e uma camada de saída e também cria nós no gráfico para calcular a perda de regularização  $\ell_1$  correspondente aos pesos de cada camada. O TensorFlow adiciona automaticamente esses nós a uma coleção especial que contém todas as perdas de regularização e você só precisa adicionar essas perdas à sua perda geral, desta forma:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```



Não se esqueça de adicionar as perdas de regularização à sua perda geral, ou então elas simplesmente serão ignoradas.

## Dropout

A técnica de regularização mais popular para redes neurais profundas é indiscutivelmente o *dropout* [descarte]. Ela foi proposta (<https://goo.gl/PMjVnG>)<sup>20</sup> por G. E. Hinton em 2012, detalhada em um artigo (<http://goo.gl/DNKZoI>)<sup>21</sup> por Nitish Srivastava *et al.*, e provou ser altamente bem-sucedida: com a adição do dropout, até mesmo as redes neurais de última geração tiveram um aumento de 1-2% na acurácia. Isso pode não parecer muito, mas, quando um modelo já possui 95% de acurácia, obter um aumento de acurácia de 2% significa reduzir sua taxa de erro em quase 40% (passando de 5% para aproximadamente 3%).

<sup>20</sup> “Improving neural networks by preventing co-adaptation of feature detectors”, G. Hinton *et al.* (2012).

<sup>21</sup> “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, N. Srivastava *et al.* (2014).

É um algoritmo bastante simples: em cada etapa do treinamento, cada neurônio (incluindo os neurônios de entrada, mas excluindo os de saída) tem uma probabilidade  $p$  de ser temporariamente “descartado”, o que significa que ele será totalmente ignorado durante esta etapa do treinamento, mas poderá estar ativo durante a próxima (veja a Figura 11-9). O hiperparâmetro  $p$  é chamado de *taxa de dropout* e normalmente é ajustado em 50% e os neurônios não são mais descartados após o treinamento. Isso é tudo (exceto por um detalhe técnico que discutiremos momentaneamente).

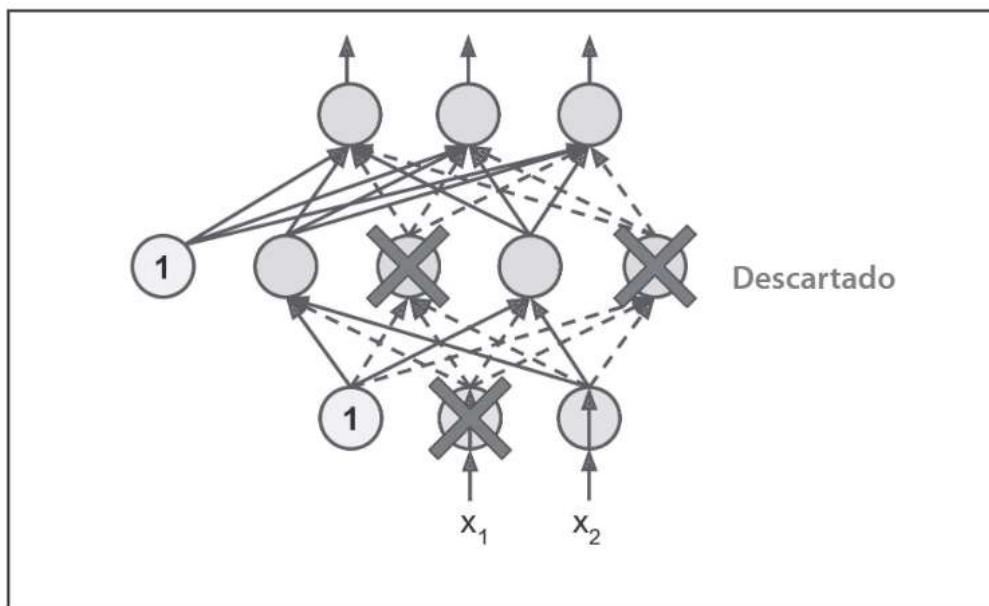


Figura 11-9. Regularização Dropout

Inicialmente, é surpreendente que esta técnica brutal funcione. Será que uma empresa funcionaria melhor se seus funcionários fossem solicitados a jogar uma moeda para o alto todas as manhãs para decidir se deveriam ou não trabalhar? Bem, quem sabe; talvez funcionasse! A empresa obviamente seria forçada a adaptar sua organização; não seria mais possível passar para uma única pessoa a responsabilidade de colocar pó na máquina de café ou executar outras tarefas críticas, então essa habilidade teria que ser distribuída entre várias pessoas. Os funcionários teriam que aprender a cooperar com muitos de seus colegas de trabalho, não apenas alguns deles. A empresa se tornaria muito mais resiliente. Se uma pessoa saísse, não faria muita diferença. Não está claro se essa ideia realmente funcionaria para as empresas, mas certamente funciona para as redes neurais. Os neurônios treinados com o dropout não podem coadaptar com seus neurônios vizinhos; eles devem ser o mais úteis possível por conta própria. Eles também não podem confiar excessivamente

em apenas alguns neurônios de entrada, devendo prestar atenção a cada um deles, o que torna-os menos sensíveis a pequenas mudanças nas entradas. No final, você obtém uma rede mais robusta que generaliza melhor.

Outra maneira de entender o poder do dropout é perceber que uma rede neural única é gerada em cada etapa do treinamento. Há um total de  $2^N$  redes possíveis (sendo que  $N$  é o número total de neurônios descartados) uma vez que cada neurônio pode estar presente ou ausente. Este é um número tão grande que é praticamente impossível que a mesma rede neural seja amostrada duas vezes. Após ter executado 10 mil passos do treinamento, você treinou basicamente 10 mil redes neurais diferentes (cada uma com apenas uma instância de treinamento). Essas redes neurais obviamente não são independentes, pois compartilham muitos de seus pesos, no entanto, são diferentes. A rede neural resultante pode ser vista como um conjunto das médias de todas essas redes neurais menores.

Há um pequeno, mas importante, detalhe técnico. Supondo que  $p = 50\%$  no caso de um teste no qual um neurônio é conectado ao dobro de neurônios de entrada (na média) em relação ao treinamento, precisamos multiplicar os pesos da conexão de entrada de cada neurônio por 0,5 após o treinamento para compensar esse fato. Se não o fizermos, cada neurônio receberá um sinal total de entrada quase duas vezes maior do que a rede em que ele foi treinado, e é improvável que ele tenha um bom desempenho. Geralmente, após o treinamento, precisamos multiplicar cada peso da conexão de entrada pela *keep probability* ( $1 - p$ ). Como alternativa, podemos dividir a saída de cada neurônio pela *keep probability* durante o treinamento (essas alternativas não são perfeitamente equivalentes, mas funcionam igualmente bem).

Quando utilizamos o TensorFlow para implementar o dropout, aplicamos a função `tf.layers.dropout()` à camada de entrada e/ou à saída de qualquer camada oculta que você quiser. Esta função descarta alguns itens aleatoriamente durante o treinamento (configurando-os para 0) e divide os restantes pela *keep probability*. Após o treinamento, esta função não faz nada. O código a seguir aplica a regularização dropout em nossa rede neural de três camadas:

```
[...]
training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5 # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                            name="hidden1")
    hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
```

```
hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                         name="hidden2")
hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")
```



Utilize a função `tf.layers.dropout()`, não a função `tf.nn.dropout()`. A primeira desliga (no-op) quando não está treinando, o que é o que você quer, enquanto a segunda não desliga.

Claro, assim como você fez antes para a Normalização em Lotes, você precisa configurar `training` para `True` para treinar e deixar o valor padrão `False` para testar.

Se você notar que o modelo está se sobreajustando, tente aumentar a taxa de dropout e, por outro lado, se o modelo se subajustar ao conjunto de treinamento, você deve tentar diminuí-la. Também pode ajudar se aumentarmos a taxa de dropout para grandes camadas e reduzi-la para as menores.

Quando ajustado corretamente, o dropout tende a reduzir significativamente a convergência, mas geralmente resulta em um modelo muito melhor. Então, vale a pena o tempo extra e o esforço.



*Dropconnect* é uma variante do dropout em que conexões individuais são descartadas aleatoriamente em vez de neurônios inteiros. No geral, o dropout tem um desempenho melhor.

## Regularização Max-Norm

Outra técnica de regularização bastante popular para as redes neurais é chamada de *regularização max-norm*: restringe o peso  $w$  das conexões de entrada de tal modo para cada neurônio que  $\|w\|_2 \leq r$  sendo que  $r$  é o hiperparâmetro *max-norm* e  $\|\cdot\|_2$  é a norma  $\ell_2$ .

Tipicamente implementamos essa restrição calculando  $\|w\|_2$  após cada passo do treinamento e limitando  $w$  se necessário ( $w \leftarrow w \frac{r}{\|w\|_2}$ ).

A redução de  $r$  aumenta a quantidade de regularização e ajuda a reduzir o sobreajuste. A regularização max-norm também pode ajudar a aliviar os problemas dos gradientes vanishing/exploding (se você não estiver utilizando a Normalização em Lote).

O TensorFlow não tem um regulador max-norm disponível, mas não é muito difícil de implementar. O código a seguir obtém um controle sobre os pesos da primeira camada oculta, então ele utiliza a função `clip_by_norm()` para criar uma operação que cortará

os pesos ao longo do segundo eixo para que cada vetor de linha fique com uma norma máxima de 1,0. A última linha cria uma operação de atribuição dos pesos limitados às variáveis de peso:

```
threshold = 1.0
weights = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

Então, basta aplicar esta operação após cada etapa de treinamento, assim:

```
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
clip_weights.eval()
```

Em geral, você faria isso para cada camada oculta. Embora esta solução funcione bem, ela é um pouco confusa e uma ideia mais limpa seria criar uma função `max_norm_regularizer()` e utilizá-la como a função anterior `l1_regularizer()`:

```
def max_norm_regularizer(threshold, axes=1, name="max_norm",
                         collection="max_norm"):
    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None # não há um termo de perda de regularização
    return max_norm
```

Esta função retorna uma função parametrizada `max_norm()` que você pode utilizar como qualquer outro regularizador:

```
max_norm_reg = max_norm_regularizer(threshold=1.0)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                             kernel_regularizer=max_norm_reg, name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
                             kernel_regularizer=max_norm_reg, name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Observe que a regularização max-norm não requer a adição de um termo de perda de regularização à sua função de perda geral, e é por isso que a função `max_norm()` retorna `None`. Mas você ainda precisa ser capaz de executar as operações `clip_weights` após cada etapa do treinamento, então você tem que poder alterá-las. É por isso que a função `max_norm()` adiciona a operação `clip_weights` a uma coleção de operações max-norm de recortes. Você precisa buscar essas operações de limitação e executá-las após cada etapa de treinamento:

```
clip_all_weights = tf.get_collection("max_norm")

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
```

```
X_batch, y_batch = mnist.train.next_batch(batch_size)
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
sess.run(clip_all_weights)
```

Um código *bem* mais limpo, não?

## Data Augmentation

Uma última técnica de regularização, o *data augmentation*, consiste na geração de novas instâncias de treinamento a partir das já existentes, aumentando artificialmente o tamanho do conjunto de treinamento, o que reduzirá o sobreajuste, tornando-a uma técnica de regularização. O truque é gerar instâncias de treinamento realistas; normalmente um humano não deve ser capaz de dizer quais instâncias foram geradas e quais não foram. Além disso, simplesmente adicionar o ruído branco não ajudará; as modificações que você aplicar devem ser aprendidas (o ruído branco não).

Por exemplo, se o seu modelo for destinado a classificar imagens de cogumelos, você pode mover ligeiramente, girar e redimensionar cada imagem no conjunto de treinamento por vários valores e adicionar as imagens resultantes ao conjunto (veja a Figura 11-10), forçando o modelo a ser mais tolerante quanto ao posicionamento, orientação e tamanho dos cogumelos na imagem. Se você deseja que o modelo seja mais tolerante às condições de iluminação, também pode gerar muitas imagens com diferentes contrastes. Supondo que os cogumelos são simétricos, você também pode virar as fotos horizontalmente. Ao combinar essas transformações, você aumenta consideravelmente o tamanho do seu conjunto de treinamento.

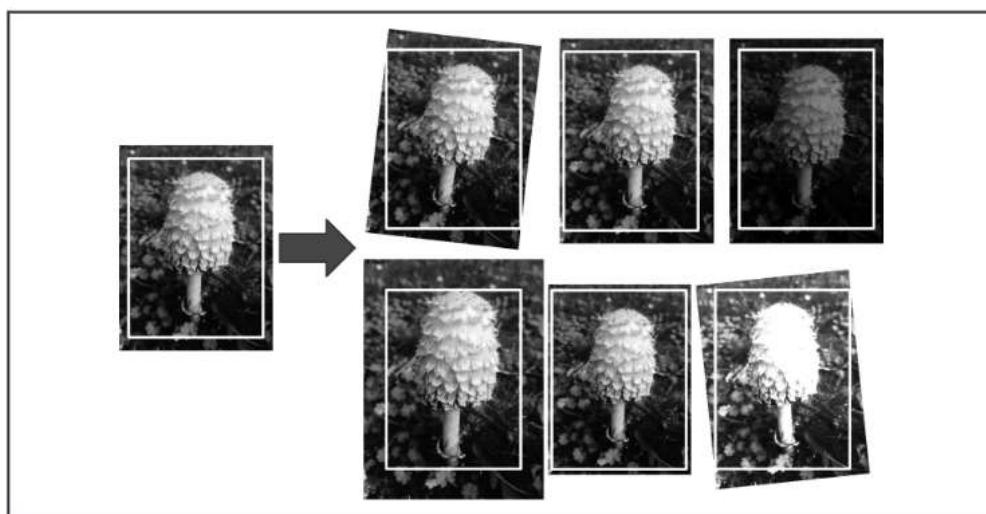


Figura 11-10. Gerando novas instâncias de treinamento a partir das já existentes

Normalmente é preferível gerar instâncias de treinamento durante o treinamento em vez de desperdiçar espaço de armazenamento e banda da rede de internet. O TensorFlow oferece várias operações de manipulação de imagem, como transposição (deslocamento), rotação, redimensionamento, recorte e corte, bem como o ajuste do brilho, contraste, saturação e matiz (consulte a documentação da API para obter mais detalhes). Isso facilita a implementação do aumento de dados para conjuntos de dados com imagens.



Outra técnica poderosa para treinar redes neurais profundas é adicionar *skip connections* (uma skip connection é a adição da entrada de uma camada à saída de uma camada superior). Exploraremos essa ideia no Capítulo 13, quando falarmos sobre redes residuais profundas.

## Diretrizes Práticas

Neste capítulo, cobrimos uma ampla gama de técnicas e você deve estar se perguntando quais deveria utilizar. A configuração na Tabela 11-2 funciona bem na maioria dos casos.

*Tabela 11-2. Configuração Padrão da DNN*

Inicialização	Inicialização He
Função de Ativação	ELU
Normalização	Normalização em Lote
Regularização	Dropout
Otimizador	Gradiente Acelerado de Nesterov
Cronograma da taxa de aprendizado	Nenhum

Claro, se conseguir encontrar uma que resolva um problema semelhante, você deve tentar reutilizar partes de uma rede neural pré-treinada.

Essa configuração padrão pode precisar ser ajustada:

- Se você não conseguir encontrar uma boa taxa de aprendizado (a convergência foi muito lenta, então você aumentou a taxa de treinamento e agora a convergência é rápida, mas a precisão da rede é subótima), tente adicionar um cronograma de aprendizado como o decaimento exponencial;
- Se o conjunto de treinamento for pequeno, implemente o data augmentation;
- Adicione alguma regularização  $\ell_1$  à mistura se precisar de um modelo esparsão (e zere opcionalmente os pequenos pesos após o treinamento). Tente utilizar a FTRL em vez da otimização Adam em conjunto com a regularização  $\ell_1$  se precisar de um modelo ainda mais esparsão;

- Se você precisar de um modelo com tempo de execução extremamente rápido, descarte a Normalização de Lote e possivelmente substitua a função de ativação ELU pela leaky ReLU. Ter um modelo esparso também pode ajudar.

Com estas orientações, você está pronto para treinar redes muito profundas — bem, isto é, se você for muito paciente! Se você utiliza apenas uma única máquina, talvez seja necessário esperar por dias ou até meses para a conclusão do treinamento. No próximo capítulo, discutiremos como utilizar o TensorFlow distribuído para treinar e executar modelos em vários servidores e GPUs.

## Exercícios

1. É correto inicializarmos todos os pesos no mesmo valor desde que esse valor seja selecionado aleatoriamente usando a inicialização He?
2. É correto inicializarmos os termos de polarização em 0?
3. Nomeie três vantagens da função de ativação ELU em relação à ReLU.
4. Em quais casos você utilizaria cada uma das seguintes funções de ativação: ELU, leaky ReLU (e suas variantes), ReLU, tanh, logistic, e softmax?
5. O que pode acontecer se você ajustar o hiperparâmetro `momentum` muito próximo de 1 (por exemplo, 0,99999) quando utilizar um `MomentumOptimizer`?
6. Nomeie três maneiras de produzir um modelo esparso.
7. O dropout retarda o treinamento? Isso retarda a inferência (ou seja, fazer previsões em novas instâncias)?
8. Aprendizado Profundo.
  - a. Crie uma DNN com cinco camadas ocultas de 100 neurônios cada, inicialização He e a função de ativação ELU.
  - b. Utilizando a otimização de Adam e a parada antecipada, tente treiná-lo no MNIST, mas apenas nos dígitos de 0 a 4, pois utilizaremos o aprendizado de transferência para os dígitos 5 a 9 no próximo exercício. Você precisará de uma camada de saída softmax com cinco neurônios e, como sempre, assegure-se de salvar os pontos de verificação em intervalos regulares e salvar o modelo final para que você possa reutilizá-lo mais tarde.
  - c. Ajuste os hiperparâmetros utilizando a validação cruzada e veja que nível de precisão você consegue atingir.
  - d. Agora tente adicionar a Normalização de Lote e compare as curvas de aprendizado: ela está convergindo mais rápido que antes? Ela produz um modelo melhor?

- e. O modelo está se sobreajustando ao conjunto de treinamento? Tente adicionar dropout a cada camada e tente novamente. Isso ajuda?
9. Aprendizado de transferência.
  - a. Crie uma nova DNN que reutilize todas as camadas ocultas pré-treinadas do modelo anterior, congele e substitua a camada de saída softmax por uma nova.
  - b. Treine esta nova DNN nos dígitos 5 a 9 utilizando apenas 100 imagens por dígito e cronometre quanto tempo leva. Apesar do pequeno número de exemplos, você consegue atingir uma alta precisão?
  - c. Tente pegar as camadas congeladas e treine o modelo novamente: quanto mais rápido ele está agora?
  - d. Tente novamente reutilizando apenas quatro camadas ocultas em vez de cinco. Você consegue atingir uma precisão mais alta?
  - e. Agora, descongele as duas camadas ocultas mais altas e continue treinando. Você consegue fazer com que o modelo tenha um desempenho ainda melhor?
10. Pré-treinamento em uma tarefa auxiliar.
  - a. Neste exercício, você construirá uma DNN que compara duas imagens de dígitos do MNIST e prevê se elas representam o mesmo dígito ou não. Então você reutilizará as camadas inferiores desta rede para treinar um classificador MNIST que utiliza poucos dados de treinamento. Comece construindo duas DNNs (vamos chamá-las DNN A e B), ambas similares à que você construiu anteriormente, mas sem a camada de saída: cada DNN deve ter cinco camadas ocultas de 100 neurônios cada, inicialização He e ativação ELU. A seguir, adicione mais uma camada oculta com 10 unidades no topo de ambas as DNNs. Para fazer isso, você deve utilizar a função `concat()` do TensorFlow com `axis=1` para concatenar as saídas de ambas as DNNs para cada instância e alimentar o resultado para a camada oculta. Finalmente, adicione uma camada de saída com um único neurônio usando a função de ativação logística.
  - b. Divida o conjunto de treinamento MNIST em dois: o conjunto #1 deve conter 55 mil imagens, e o conjunto #2 deve conter 5 mil imagens. Crie uma função que gere um lote de treinamento em que cada instância é um par de imagens MNIST escolhidas a partir do conjunto #1. A metade das instâncias de treinamento deve ter pares de imagens que pertencem à mesma classe, enquanto a outra metade deve conter imagens de diferentes classes. Para cada par, se as imagens forem da mesma classe, o label de treinamento deve ser 0, ou 1, se forem de classes diferentes.

- c. Treine a DNN neste conjunto de treinamento. Para cada par de imagens, você alimenta simultaneamente a primeira imagem na DNN A e a segunda imagem na DNN B. Toda a rede aprenderá gradualmente se duas imagens pertencem à mesma classe ou não.
- d. Agora, crie uma nova DNN reutilizando e congelando as camadas ocultas da DNN A e adicione uma camada de saída softmax com 10 neurônios no topo. Treine esta rede no conjunto #2 e veja se pode alcançar um alto desempenho apesar de ter apenas 500 imagens por classe.

As soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 12

# Distribuindo o TensorFlow Por Dispositivos e Servidores

No Capítulo 11, discutimos várias técnicas que podem acelerar consideravelmente o treinamento: melhorar a inicialização do peso, a Normalização em Lote, otimizadores sofisticados e assim por diante. No entanto, mesmo com todas essas técnicas, o treinamento de uma grande rede neural em uma única máquina com uma única CPU pode levar dias ou até mesmo semanas.

Neste capítulo, veremos como utilizar o TensorFlow para distribuir cálculos em vários dispositivos (CPUs e GPUs) e executá-los em paralelo (veja a Figura 12-1). Em primeiro lugar, distribuiremos cálculos através de vários dispositivos em apenas uma máquina, depois em vários dispositivos em várias máquinas.

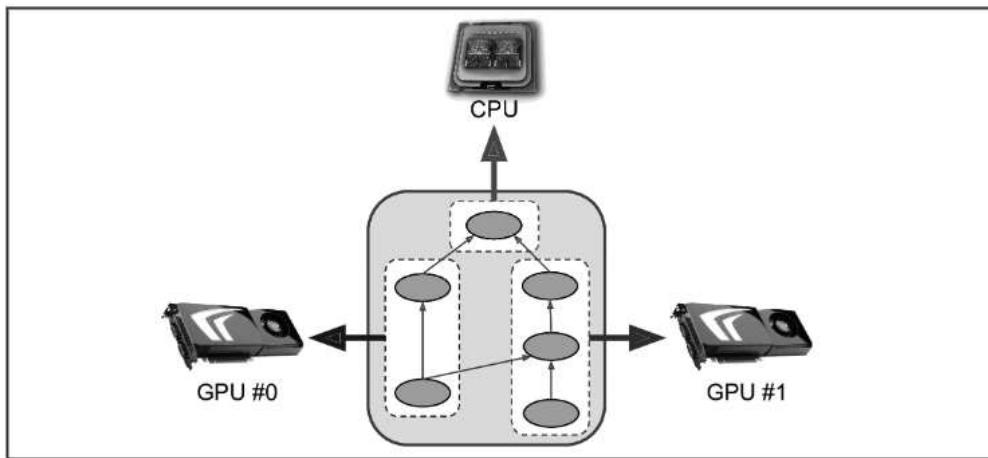


Figura 12-1. Executando um grafo do TensorFlow em múltiplos dispositivos paralelos

O suporte do TensorFlow ao cálculo distribuído é um dos principais destaques em comparação com outras estruturas de redes neurais. Ele oferece controle total sobre como dividir (ou replicar) seu gráfico de cálculo entre dispositivos e servidores, e permite paralelizar e sincronizar operações de formas flexíveis para que você possa escolher entre todos os tipos de abordagens de paralelização.

Examinaremos algumas das abordagens mais populares para paralelizar a execução e o treinamento de uma rede neural. Em vez de esperar semanas para que um algoritmo de treinamento seja concluído, pode ser que você só precise aguardar algumas horas. Isso não apenas economiza uma quantidade enorme de tempo, mas também permite que você experimente vários modelos com muito mais facilidade e, com frequência, treine novamente seus modelos em novos dados.

Outros ótimos casos de uso de paralelização incluem a exploração de um espaço de hiperparâmetro muito maior para ajustar seu modelo e a execução eficiente de grandes ensembles de redes neurais.

Mas precisamos aprender a caminhar antes de correr. Começaremos paralelizando grafos simples através de várias GPUs em uma única máquina.

## Múltiplos Dispositivos em uma Única Máquina

Você pode obter um grande aumento de desempenho adicionando placas GPU a uma única máquina. De fato, em muitos casos isso será suficiente e não será necessário utilizar várias máquinas. Por exemplo, você pode treinar uma rede neural com a mesma rapidez utilizando 8 GPUs em uma única máquina em vez de 16 GPUs em várias máquinas (devido ao atraso extra imposto pelas comunicações de rede em uma configuração de várias máquinas).

Nesta seção, veremos como configurar seu ambiente para que o TensorFlow possa utilizar várias placas GPU em uma máquina. Em seguida, veremos como você pode distribuir as operações pelos dispositivos disponíveis e executá-las em paralelo.

### Instalação

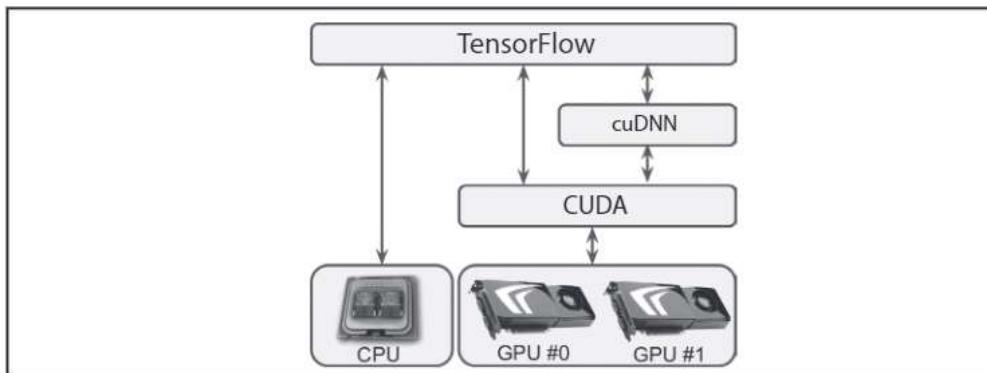
Para executar o TensorFlow em várias placas GPU, você precisa primeiro certificar-se de que suas placas têm Nvidia Compute Capability (maior ou igual a 3.0). Isso inclui as placas Titan, Titan X, K20 e K40 da Nvidia (caso tenha outra placa, verifique sua compatibilidade em <https://developer.nvidia.com/cuda-gpus>).



Se você não possui placas GPU, pode utilizar um serviço de hospedagem com capacidade de GPU, como o Amazon AWS. As instruções detalhadas para configurar o TensorFlow 0.9 com o Python 3.5 em uma instância de GPU da Amazon AWS estão disponíveis na publicação do blog de Žiga Avsec (<http://goo.gl/kbge5b>). Não deve ser difícil atualizá-la para a versão mais recente do TensorFlow. O Google também lançou um serviço em nuvem chamado *Cloud Machine Learning* (<https://cloud.google.com/ml>) para executar grafos do TensorFlow. Em maio de 2016, a empresa anunciou que sua plataforma agora inclui servidores equipados com *unidades de processamento do tensor* (TPUs), processadores especializados em Aprendizado de Máquina que são muito mais rápidos do que as GPUs para muitas das tarefas do AM. Claro, outra opção seria comprar sua própria GPU. Tim Dettmers escreveu uma excelente postagem no blog (<https://goo.gl/pCtSAn>) para ajudar você a escolhê-la, e a atualiza com bastante regularidade.

Baixe e instale a versão apropriada das bibliotecas CUDA e cuDNN (CUDA 8.0 e cuDNN v6 se estiver utilizando a instalação binária do TensorFlow 1.3) e defina algumas variáveis de ambiente para que o TensorFlow saiba onde encontrar CUDA e cuDNN. As instruções detalhadas de instalação estão suscetíveis a mudanças constantes, por isso é melhor que você siga as instruções no site do TensorFlow.

A biblioteca da Nvidia *Compute Unified Device Architecture* (CUDA) permite que desenvolvedores utilizem GPUs *CUDA-enabled* para todos os tipos de cálculos (não apenas aceleração gráfica). A biblioteca da Nvidia *CUDA Deep Neural Network* (cuDNN) é uma biblioteca *GPU-accelerated* para DNNs primitivas. Ela fornece implementações otimizadas para cálculos comuns da DNN, tais como camadas de ativação, normalização, contornos avançados e atrasados e agrupamento (consulte o Capítulo 13). Ela faz parte do Aprendizado Profundo SDK da Nvidia (requer a criação de uma conta de desenvolvedor Nvidia para efetuar o download). O TensorFlow utiliza CUDA e cuDNN para controlar as placas GPU e acelerar os cálculos (veja a Figura 12-2).



*Figura 12-2. O TensorFlow utiliza CUDA e cuDNN para controlar GPUs e aumentar as DNNs*

Você pode utilizar o comando `nvidia-smi` para verificar se CUDA está instalado de acordo, pois ele lista os cartões GPU disponíveis e processa a execução em cada um:

```

$ nvidia-smi
Wed Sep 16 09:50:03 2016
+-----+
| NVIDIA-SMI 352.63      Driver Version: 352.63      |
+-----+
| GPU  Name     Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====
|  0  GRID K520          Off | 0000:00:03.0  Off |                  N/A |
| N/A   27C   P8    17W / 125W |      11MiB /  4095MiB |     0%       Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU     PID  Type  Process name        Usage        |
| =====+=====+=====+=====
| No running processes found               |
+-----+
  
```

Finalmente, você deve instalar o TensorFlow com suporte a GPU. Se criou um ambiente isolado com o virtualenv, primeiro você precisa ativá-lo:

```

$ cd $ML_PATH           # Seu diretório de AM (por ex., $HOME/ml)
$ source env/bin/activate
  
```

Instale a versão GPU-enabled apropriada do TensorFlow:

```

$ pip3 install --upgrade tensorflow-gpu
  
```

Agora, você pode abrir um Python shell e verificar se o TensorFlow detecta e utiliza CUDA e cuDNN corretamente ao importar e criar uma sessão no TensorFlow:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0: Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```

Parece ok! O TensorFlow detectou as bibliotecas CUDA e cuDNN e utilizou a biblioteca CUDA para detectar o cartão GPU (neste caso, um cartão Nvidia Grid K520).

## Gerenciando a RAM da GPU

Por padrão, na primeira vez que você executa um grafo, o TensorFlow captura automaticamente toda a RAM em todas as GPUs disponíveis, então você não conseguirá iniciar um segundo programa do TensorFlow enquanto o primeiro ainda estiver em execução. Se tentar, receberá o seguinte erro:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes) from
device: CUDA_ERROR_OUT_OF_MEMORY
```

Uma solução seria executar cada processo em diferentes cartões GPU. Para fazer isso, a opção mais simples é configurar a variável de ambiente `CUDA_VISIBLE_DEVICES` para que cada processo veja somente os cartões GPU apropriados. Por exemplo, você poderia iniciar dois programas assim:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# e em outro terminal:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

O programa #1 somente verá os cartões GPU 0 e 1 (numerados 0 e 1, respectivamente), e o programa #2 somente verá os cartões GPU 2 e 3 (numerados 1 e 0, respectivamente). Tudo funcionará de acordo (veja a Figura 12-3).

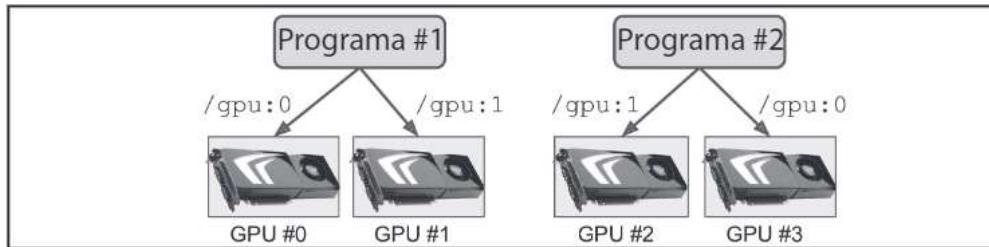


Figura 12-3. Cada programa pega duas GPUs para si

Outra opção seria dizer ao TensorFlow que capture apenas uma fração da memória. Por exemplo, crie um objeto `ConfigProto` para fazer o TensorFlow capturar apenas 40% da memória de cada GPU, ajuste sua opção `gpu_options.per_process_gpu_memory_fraction` para 0.4 e crie a sessão utilizando esta configuração:

```

config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
  
```

Agora dois programas como este podem rodar em paralelo usando as mesmas GPU (mas não três, pois  $3 \times 0.4 > 1$ ). Veja a Figura 12-4.

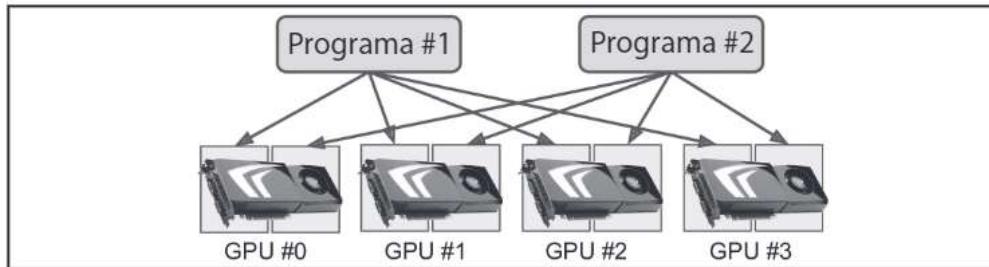


Figura 12-4. Cada programa capture todas as quatro GPUs, mas somente com 40% da RAM cada uma

Se executar o comando `nvidia-smi` enquanto ambos os programas estão rodando, verificará que cada processo utiliza mais ou menos 40% da RAM total de cada cartão:

```

$ nvidia-smi
[...]
+-----+
| Processes:
| GPU     PID  Type  Process name          GPU Memory |
|           |       |      |                         Usage   |
| ======+=====+=====+=====+=====+=====+=====+=====
|     0     5231  C    python                  1677MiB |
|     0     5262  C    python                  1677MiB |
|     1     5231  C    python                  1677MiB |
|     1     5262  C    python                  1677MiB |
[...]
  
```

Outra opção é dizer ao TensorFlow que capture memória somente quando precisar dela. Para fazer isso, você deve configurar `config.gpu_options.allow_growth` como `True`. No entanto, uma vez que a tenha capturado, o TensorFlow nunca liberará memória (para evitar a fragmentação), então você ainda poderá ficar sem memória depois de um tempo. Garantir um comportamento determinista ao utilizar esta opção pode ser mais difícil, portanto, utilize uma das opções anteriores.

Ok, agora você tem uma instalação GPU-enabled funcional do TensorFlow. Vejamos como utilizá-la!

## Colocando Operações em Dispositivos

O *whitepaper* do TensorFlow (<http://goo.gl/vSjA14>)<sup>1</sup> apresenta um algoritmo de *posicionamento dinâmico* amigável que distribui automaticamente operações em todos os dispositivos disponíveis levando em consideração itens como o tempo de cálculo medido em execuções anteriores do gráfico, estimativas do tamanho dos tensores de entrada e saída para cada operação, a quantidade de RAM disponível em cada dispositivo, o atraso de comunicação ao transferir dados dentro e fora dos dispositivos, sugestões e restrições do usuário, e muito mais. Infelizmente, este algoritmo sofisticado é interno do Google; não foi lançado na versão open source do TensorFlow. A razão pela qual foi deixado de fora parece ser que, na prática, um pequeno conjunto de regras, especificadas pelo usuário, realmente resulta em um posicionamento mais eficiente do que do *dynamic placer*. No entanto, a equipe do TensorFlow está trabalhando para melhorar este *dynamic placer*, e talvez um dia ele esteja bom o bastante para ser lançado.

Até então, o TensorFlow depende de um *posicionador simples*, o qual (como seu nome sugere) é muito básico.

### Posicionamento simples

Sempre que você executar um grafo, se o TensorFlow precisar avaliar um nó que ainda não esteja posicionado em um dispositivo, ele utilizará o posicionamento simples para posicioná-lo juntamente com todos os outros nós que ainda não foram posicionados. O posicionamento simples respeita as seguintes regras:

- Se um nó já foi posicionado em um dispositivo em uma execução anterior do grafo, ele é deixado neste dispositivo;
- Caso contrário, se o usuário *fixou* um nó em um dispositivo (descrito a seguir), o posicionador o coloca neste dispositivo;

---

<sup>1</sup> “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” Google Research (2015).

- Caso contrário, ele padroniza para GPU #0 ou para CPU se não houver GPU.

Como você pode ver, posicionar as operações no dispositivo apropriado depende principalmente de você. Caso não faça nada, todo o grafo será posicionado no dispositivo padrão, portanto, você deve criar um bloco de dispositivo com a função `device()` para posicionar os nós em um dispositivo. Por exemplo, o código a seguir posiciona a variável `a` e a constante `b` na CPU, mas o nó de multiplicação `c` não está fixado em nenhum dispositivo, então será posicionado no dispositivo padrão:

```
with tf.device("/cpu:0"):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)

    c = a * b
```



O dispositivo “/cpu:0” agrupa todas as CPUs em um sistema multi-CPU. Atualmente, não há como fixar nós em CPUs específicas ou utilizar apenas um subconjunto de todas as CPUs.

## Registro dos Posicionamentos

Verificaremos se o posicionamento simples respeita as restrições de posicionamento que acabamos de definir. Para isso, você pode definir a opção `log_device_placement` para `True`; que leva o posicionador a registrar uma mensagem sempre que posicionar um nó. Por exemplo:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GRID K520,
pci bus id: 0000:00:03.0)
[...]
>>> a.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

As linhas que começam com “I” de Info são as mensagens de registro. Quando criamos uma sessão, o TensorFlow grava uma mensagem para nos avisar que encontrou uma GPU (neste caso, o cartão Grid K520). Então, na primeira vez que rodamos o grafo (neste caso quando inicializamos a variável `a`), o posicionador simples é executado e coloca cada nó no dispositivo ao qual foi atribuído. Como esperado, as mensagens do log mostram que

todos os nós estão posicionados em “/cpu:0”, exceto o nó de multiplicação que acaba no dispositivo padrão “/gpu:0” (por ora você pode ignorar com segurança o prefixo /job:localhost/replica:0/task:0; em breve falaremos a respeito). Observe que, ao executarmos o grafo pela segunda vez (para computar c), o posicionador não é utilizado, pois todos os nós que o TensorFlow precisa para calcular c já estão posicionados.

### Função de posicionamento dinâmico

Ao criar um bloco de dispositivo, você pode especificar uma função em vez de um nome para o dispositivo. O TensorFlow chamará essa função para cada operação necessária no bloco e a função deve retornar o nome do dispositivo que a operação será fixada. Por exemplo, o código a seguir posiciona todos os nós das variáveis em “/cpu:0” (neste caso, apenas a variável a) e todos os outros nós em “/gpu:0”:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b
```

Você pode implementar algoritmos mais complexos em um estilo *round-robin*, como o posicionamento de variáveis através das GPUs.

### Operações e kernels

Para que uma operação do TensorFlow seja executada em um dispositivo, ela precisa ter uma implementação para ele; isso é chamado de *kernel*. Muitas operações possuem kernels para CPUs e GPUs, mas não todas. Por exemplo, o TensorFlow não possui um kernel GPU para variáveis inteiras, então o código a seguir falhará quando o TensorFlow tentar anexar a variável i na GPU #0:

```
>>> with tf.device("/gpu:0"):
...     i = tf.Variable(3)
...
>>> sess.run(i.initializer)
Traceback (most recent call last):
...
tensorflow.python.framework.errors.InvalidArgumentError: Cannot assign a device
to node 'Variable': Could not satisfy explicit device specification
```

Observe que o TensorFlow infere que a variável deve ser do tipo `int32`, uma vez que o valor de inicialização é um número inteiro. Se você alterar o valor de inicialização para

`3,0` em vez de `3` ou se configurar explicitamente `dtype=tf.float32` ao criar a variável, tudo trabalhará de acordo.

### Posicionamento suave

Por padrão, se tentar inserir uma operação em um dispositivo para o qual não possua kernel, você obterá a exceção mostrada anteriormente quando o TensorFlow tentar posicionar a operação no dispositivo. Se preferir que o TensorFlow retroceda para a CPU, configure a opção `allow_soft_placement` como `True`:

```
with tf.device("/gpu:0"):
    i = tf.Variable(3)

config = tf.ConfigProto()
config.allow_soft_placement = True
sess = tf.Session(config=config)
sess.run(i.initializer) # Executa o posicionamento e volta para a posição anterior
```

Até agora discutimos como posicionar nós em diferentes dispositivos. Agora, veremos como o TensorFlow executará esses nós em paralelo.

### Execução em Paralelo

Quando o TensorFlow executa um grafo, ele começa descobrindo a lista de nós que precisam ser avaliados e conta quantas dependências cada um tem. O TensorFlow, então, começa a avaliar os nós com dependências zero (ou seja, nós de origem). Se esses nós forem posicionados em dispositivos separados, obviamente serão avaliados em paralelo. Caso forem posicionados no mesmo dispositivo, serão avaliados em diferentes threads para que também possam ser executadas em paralelo (em threads de GPU separadas ou núcleos de CPU).

O TensorFlow gerencia um pool de threads em cada dispositivo para paralelizar operações (veja Figura 12-5), chamadas de *inter-op thread pools*. Algumas operações têm kernels multithreaded, que podem utilizar outros pools de threads (um por dispositivo) chamados *intra-op thread pools*.

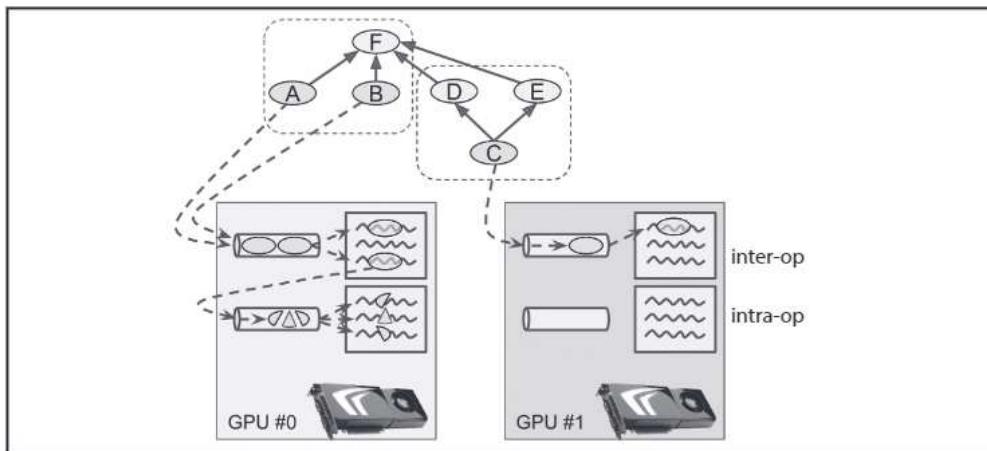


Figura 12-5. Execução paralela de um grafo do TensorFlow

Por exemplo, na Figura 12-5 as operações A, B e C são source ops, então podem ser avaliadas imediatamente. As operações A e B são posicionadas em GPU #0, enviadas a este pool de threads inter-op do dispositivo e imediatamente avaliadas em paralelo. A operação A passa a ter um kernel multithread; seus cálculos, que são executados em paralelo pelo pool de threads inter-op, são divididos em três partes. A operação C vai para o pool de threads inter-op de GPU #1.

Os contadores de dependência das operações D e E serão diminuídos assim que terminar a operação C, e ambos alcançarão 0 de modo que as duas operações serão enviadas para o conjunto de threads inter-op a ser executado.



Você pode controlar o número de threads pelo pool inter-op definindo a opção `inter_op_parallelism_threads`. Observe que a primeira sessão iniciada cria os pools de threads inter-op. Todas as outras sessões apenas os reutilizarão, a menos que você configure a opção `use_per_session_threads` como `True`. Você pode controlar o número de threads por grupo intra-op definindo a opção `intra_op_parallelism_threads`.

## Dependências de Controle

Mesmo que todas as operações dependentes sejam executadas, em alguns casos pode ser aconselhável adiar a avaliação de uma operação. Por exemplo, se ela utiliza muita memória, mas seu valor é necessário apenas mais tarde no grafo, seria melhor avaliá-la no último momento, evitando assim ocupar a RAM que outras operações podem necessitar. Outro exemplo é um conjunto de operações que depende de dados localizados fora do dispositivo. Se todos rodam ao mesmo tempo, podem saturar a largura de banda

da comunicação do dispositivo e acabarão todos aguardando em I/O. Outras operações que precisam comunicar dados também serão bloqueadas. Seria preferível executar sequencialmente essas operações pesadas de comunicação permitindo que o dispositivo execute outras operações em paralelo.

Uma solução simples seria adicionar *dependências de controle* para adiar a avaliação de alguns nós. Por exemplo, o código a seguir pede ao TensorFlow para avaliar `x` e `y` somente após avaliar `a` e `b`:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Obviamente, como `z` depende de `x` e `y`, avaliar `z` também implica esperar que `a` e `b` sejam avaliados, embora não esteja explícito no bloco `control_dependencies()`. Também, como `b` depende de `a`, poderíamos simplificar o código anterior criando apenas uma dependência de controle em `[b]` em vez de `[a, b]`, mas, em alguns casos, “explícito é melhor que implícito”.

Ótimo! Agora você sabe:

- Como inserir operações em vários dispositivos da maneira que desejar;
- Como essas operações são executadas em paralelo;
- Como criar dependências de controle para otimizar a execução em paralelo.

É hora de distribuir os cálculos pelos múltiplos servidores!

## Vários Dispositivos em Vários Servidores

Para executar um grafo em vários servidores, você precisa primeiro definir um *cluster*. Um cluster é composto por um ou mais servidores do TensorFlow espalhados em várias máquinas chamadas *tasks* (veja a Figura 12-6). Cada task pertence a um *job*. Um job é apenas um grupo de tarefas que tem um papel comum, como manter o controle dos parâmetros do modelo (este job é denominado “`ps`” *parameter server*), ou executar cálculos (este job é nomeado “`worker`”).

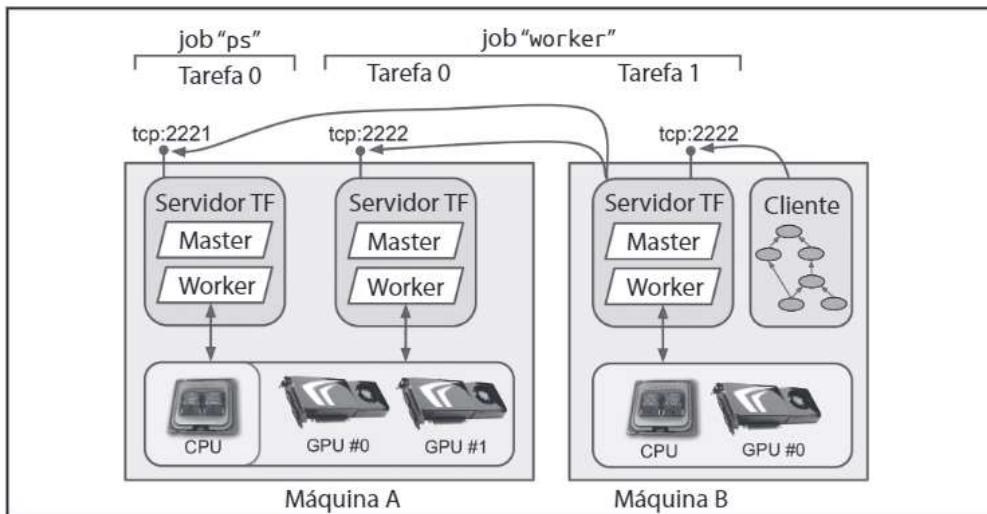


Figura 12-6. Cluster do TensorFlow

A seguinte *especificação de cluster* define dois jobs, “*ps*” e “*worker*” contendo uma e duas tasks respectivamente. Neste exemplo, a máquina A hospeda dois servidores do TensorFlow (isto é, tasks) escutando em diferentes portas: uma faz parte do job “*ps*” e a outra faz parte do job “*worker*”. A máquina B apenas hospeda um servidor do TensorFlow, parte do job “*worker*”.

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221", # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222", # /job:worker/task:0
        "machine-b.example.com:2222", # /job:worker/task:1
    ]
})
```

Passando a especificação do cluster, você deve criar um objeto `Server` para iniciar um servidor do TensorFlow (para que ele possa se comunicar com outros servidores) e seu próprio nome de job e o número da task. Por exemplo, para iniciar a primeira task worker, você executaria o seguinte código na máquina A:

```
server = tf.train.Server(cluster_spec, job_name="worker", task_index=0)
```

Costuma ser mais simples executar uma tarefa por máquina, mas o exemplo anterior demonstra que o TensorFlow permite que você execute várias tarefas, se desejar, na mesma máquina.<sup>2</sup> Se tiver vários servidores em uma máquina, é necessário garantir que eles não consumam toda a RAM de cada GPU, conforme explicado anteriormente. Por

<sup>2</sup> Você pode até iniciar várias tarefas no mesmo processo. Pode ser útil para testes, mas não é recomendado na produção.

exemplo, na Figura 12-6, a tarefa “ps” não enxerga os dispositivos GPU, pois presumivelmente seu processo foi disparado com `CUDA_VISIBLE_DEVICES=""`. Observe que a CPU é compartilhada por todas as tarefas localizadas na mesma máquina.

Se você quiser que o processo não faça nada além de executar o servidor do TensorFlow, bloqueeie o thread principal pedindo para que ele espere que o servidor termine de utilizar o método `join()` (caso contrário, o servidor estará desabilitado assim que finalizar sua thread principal). Como atualmente ainda não é possível parar o servidor, isso realmente bloqueará para sempre:

```
server.join() # bloqueia até que o servidor pare (ou seja, nunca)
```

## Abrindo uma Sessão

Uma vez que todas as tarefas estejam sendo executadas (ainda não fazendo nada), você pode abrir uma sessão em qualquer um dos servidores, de um cliente localizado em qualquer processo em qualquer máquina (mesmo de um processo rodando uma das tarefas) e utilizar essa sessão como uma sessão local normal. Por exemplo:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3

with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval()) # 9.0
```

Este código do cliente primeiro cria um grafo simples, então abre uma sessão no servidor do TensorFlow localizado na máquina B (que chamaremos *master*), e o instrui para avaliar `c`. O master começa a inserir as operações nos dispositivos apropriados. Neste exemplo, uma vez que não posicionamos nenhuma operação em nenhum dispositivo, o master simplesmente os posiciona em seu próprio dispositivo padrão — neste caso, o dispositivo GPU da máquina B. Então, ele apenas avalia `c` conforme instruído pelo cliente e retorna o resultado.

## Os Serviços Master e Worker

O cliente utiliza o protocolo *gRPC* (*Google Remote Procedure Call*) para se comunicar com o servidor. Esta é uma estrutura eficiente de código aberto para chamar funções remotas e obter suas saídas em uma variedade de plataformas e linguagens.<sup>3</sup> É baseado

---

<sup>3</sup> É a próxima versão do serviço interno Stubby do Google que foi utilizado com sucesso por mais de uma década. Veja <http://grpc.io/> para mais detalhes.

no HTTP2, que abre uma conexão e a deixa aberta durante toda a sessão, permitindo uma comunicação bidirecional eficiente assim que a conexão for estabelecida.

Os dados são transmitidos na forma de *protocol buffers*, outra tecnologia de código aberto do Google, que é um formato intercambiável de dados binários leves.



Todos os servidores em um cluster do TensorFlow podem se comunicar com qualquer outro servidor, portanto, assegure-se de abrir as portas apropriadas em seu firewall.

Todo o servidor do TensorFlow oferece dois serviços: o *master service* e o *worker service*. O master service permite que clientes abram sessões e as utilizem para rodar grafos, além de coordenar os cálculos nas tarefas confiando no service worker para realmente executar cálculos em outras tarefas e obter seus resultados.

Esta arquitetura oferece muita flexibilidade. Um cliente pode se conectar a vários servidores abrindo várias sessões em diferentes threads. Um servidor pode lidar com várias sessões simultaneamente de um ou mais clientes. Você pode rodar um cliente por tarefa (normalmente dentro do mesmo processo), ou apenas um cliente para controlar todas as tarefas. Todas as opções são possíveis.

## Fixando Operações em Tarefas

Você pode utilizar blocos do dispositivo para inserir operações em qualquer dispositivo gerenciado por qualquer tarefa especificando o nome do job, o índice da task, o tipo de dispositivo e o índice do dispositivo. Por exemplo, o código a seguir fixa `a` na CPU da primeira tarefa no job “`ps`” (esta é a CPU da máquina A), e fixa `b` na segunda GPU gerenciada pela primeira tarefa do job “`worker`” (esta é a GPU #1 da máquina A). Finalmente, `c` não é fixada em dispositivo algum, então o master a posiciona em seu próprio dispositivo padrão (o dispositivo da máquina B GPU #0).

```
with tf.device("/job:ps/task:0/cpu:0"):
    a = tf.constant(1.0)

with tf.device("/job:worker/task:0/gpu:1"):
    b = a + 2

c = a + b
```

Como anteriormente, se você omitir o tipo e o índice do dispositivo, o TensorFlow utilizará o dispositivo padrão para a task; por exemplo, fixar uma operação em “`/job:ps/task:0`” irá inseri-la no dispositivo padrão da primeira tarefa do job “`ps`” (CPU da máquina A). Se

você também omitir o índice da task (por exemplo, “/job:ps”), o TensorFlow padroniza em “/task:0” e, se você omitir o nome do job e o índice da task, o TensorFlow padroniza para o master task da sessão.

## Particionando Variáveis em Múltiplos Servidores de Parâmetros

Como veremos em breve, um padrão comum no treinamento de uma rede neural em uma instalação distribuída é armazenar os parâmetros do modelo em um conjunto de servidores de parâmetros (ou seja, as tarefas no job “ps”), enquanto outras tarefas se concentram em cálculos (ou seja, as tarefas no job “worker”). Para grandes modelos com milhões de parâmetros, vale a pena partitionar estes parâmetros em vários servidores, a fim de reduzir o risco de saturação da placa de rede do servidor de parâmetro unitário. Seria bastante tedioso se você tivesse que inserir manualmente cada variável em um servidor diferente. Felizmente, o TensorFlow fornece a função `replica_device_setter()` no formato round-robin, que distribui variáveis em todas as tarefas “ps”. Por exemplo, o código a seguir fixa cinco variáveis para dois servidores de parâmetros:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # pinned to /job:ps/task:0
    v2 = tf.Variable(2.0) # pinned to /job:ps/task:1
    v3 = tf.Variable(3.0) # pinned to /job:ps/task:0
    v4 = tf.Variable(4.0) # pinned to /job:ps/task:1
    v5 = tf.Variable(5.0) # pinned to /job:ps/task:0
```

Em vez de passar o número de `ps_tasks`, você pode passar a especificação do cluster `clus ter=cluster_spec` e o TensorFlow contará o número de tarefas no job “ps”.

O TensorFlow fixa automaticamente em “/job:worker” se você criar outras operações no bloco além das variáveis, que será padrão para o primeiro dispositivo gerenciado pela primeira tarefa no job “worker”. Definindo o parâmetro `worker_device`, você pode fixá-lo em outro dispositivo, mas uma melhor abordagem seria utilizar blocos de dispositivos incorporados. Um bloco de dispositivo interno pode substituir o job, a tarefa ou o dispositivo definido em um bloco externo. Por exemplo:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):
    v1 = tf.Variable(1.0) # marcado em /job:ps/task:0 (+ padrão em /cpu:0)
    v2 = tf.Variable(2.0) # marcado em /job:ps/task:1 (+ padrão em /cpu:0)
    v3 = tf.Variable(3.0) # marcado em /job:ps/task:0 (+ padrão em /cpu:0)
    [...]
    s = v1 + v2          # marcado em /job:worker (+ padrão em task:0/gpu:0)
    with tf.device("/gpu:1"):
        p1 = 2 * s      # marcado em /job:worker/gpu:1 (+ padrão em /task:0)
        with tf.device("/task:1"):
            p2 = 3 * s  # marcado em /job:worker/task:1/gpu:1
```



Este exemplo pressupõe que os servidores de parâmetros são somente para a CPU, o que normalmente é o caso, pois eles só precisam armazenar e comunicar parâmetros, não realizar cálculos intensivos.

## Compartilhando Estado entre Sessões com a Utilização de Contêiner de Recursos

Quando você utiliza uma *sessão local* comum (não do tipo distribuída), o estado de cada variável é gerenciado pela própria sessão; todos os valores variáveis são perdidos assim que acaba. Além disso, várias sessões locais não podem compartilhar nenhum estado, mesmo que ambos rodem o mesmo grafo; cada sessão tem sua própria cópia de cada variável (como discutimos no Capítulo 9). Em contraste, se você estiver usando *sessões distribuídas*, o estado da variável é gerenciado pelos *contêineres de recursos* localizados no próprio cluster, não pelas sessões. Então, se você criar uma variável chamada `x` utilizando uma sessão do cliente (mesmo que ambas as sessões estejam conectadas a um servidor diferente), ela estará automaticamente disponível para qualquer outra sessão no mesmo cluster. Por exemplo, considere o seguinte código do cliente:

```
# simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2]==["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Suponhamos que você tenha um cluster do TensorFlow instalado e rodando nas máquinas A e B, porta 2222. Você poderia iniciar o cliente, abrir uma sessão com o servidor na máquina A e informá-lo para inicializar a variável, incrementá-la e imprimir seu valor executando o seguinte comando:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222 init
1.0
```

Agora, se você disparar o cliente com o seguinte comando, ele se conectará ao servidor na máquina B e reutilizará magicamente a mesma variável `x` (desta vez não pedimos ao servidor para inicializar a variável):

```
$ python3 simple_client.py grpc://machine-b.example.com:2222
2.0
```

Este recurso tem vantagens e desvantagens: é ótimo se você deseja compartilhar variáveis em diversas sessões, mas, para executar cálculos completamente independentes no mesmo cluster, você terá que ter cuidado para não utilizar, por acidente, os mesmos nomes de variáveis. Uma maneira de garantir que você não terá conflitos de nomes seria envolver toda a sua fase de construção dentro de um escopo variável com um nome exclusivo para cada cálculo, por exemplo:

```
with tf.variable_scope("my_problem_1"):
    [...] # Fase de construção do problema
```

Uma solução melhor seria utilizar um bloco de contêiner:

```
with tf.container("my_problem_1"):
    [...] # Fase de construção do problema
```

Isso utilizará um contêiner dedicado ao problema #1 em vez do padrão (cujo nome é uma string vazia “ ”). Suas vantagens incluem o fato de os nomes das variáveis ficarem fáceis e curtos, e ser possível facilmente resetar um contêiner nomeado. Por exemplo, o comando a seguir se conectará ao servidor na máquina A e pedirá para resetar o contêiner denominado “`my_problem_1`”, que liberará todos os recursos que ele utilizou (e também fechará todas as sessões abertas no servidor). Qualquer variável gerenciada por este contêiner deve ser inicializada antes de ser utilizada novamente:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Os contêineres de recursos facilitam de forma flexível o compartilhamento de variáveis entre as sessões. Por exemplo, a Figura 12-7 mostra quatro clientes que rodam gráficos diferentes no mesmo cluster, mas compartilham algumas variáveis. Os clientes A e B compartilham a mesma variável `x` gerenciada pelo contêiner padrão, enquanto os clientes C e D compartilham outra variável chamada `x` gerenciada pelo contêiner chamado “`my_problem_1`”. Observe que o cliente C utiliza as variáveis de ambos os contêineres.

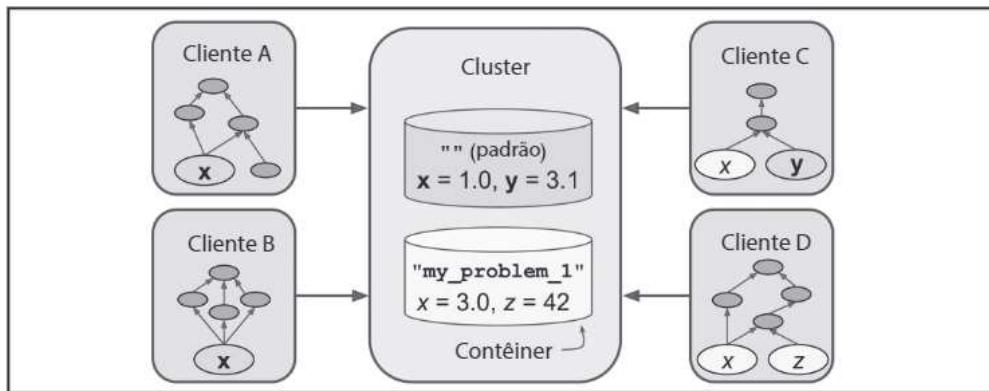


Figura 12-7. Contêineres de recursos

Os contêineres de recursos denominados filas e leitores também preservam o estado de outras operações com status. Daremos uma olhada em filas primeiro.

## Comunicação Assíncrona com a Utilização de Filas do TensorFlow

Filas são outra ótima maneira de trocar dados entre várias sessões; por exemplo, um caso de uso comum é ter um cliente que cria um grafo que carrega os dados de treinamento e os empurra para uma fila, enquanto outro cliente cria um grafo que puxa os dados da fila e treina um modelo (veja a Figura 12-8). Isso pode acelerar consideravelmente o treinamento porque as operações não precisam esperar o próximo minilote em cada etapa.

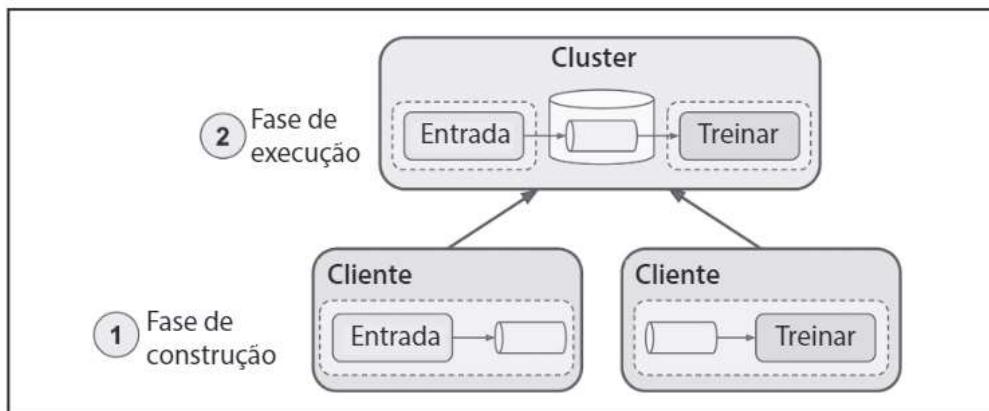


Figura 12-8. Utilizando filas para carregar os dados de treinamento de forma assíncrona

O TensorFlow fornece vários tipos de filas e o mais simples deles é a fila *first-in first-out* (FIFO). Por exemplo, o código a seguir cria uma fila FIFO que pode armazenar até 10 tensores contendo dois valores de flutuação cada:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
                  name="q", shared_name="shared_q")
```



Para compartilhar variáveis entre as sessões, você só precisava especificar o mesmo nome e contêiner nas duas extremidades. O TensorFlow não utiliza o atributo `name` em filas, mas sim o `shared_name`, portanto é importante especificá-lo (mesmo que seja o mesmo que o `name`). E, claro, que utilize o mesmo contêiner.

## Enfileirando dados

Para empurrar dados para uma fila, você deve criar uma operação `enqueue`. Por exemplo, o código a seguir empurra três instâncias de treinamento para a fila:

```
# training_data_loader.py
import tensorflow as tf

q = tf.FIFOQueue(capacity=10, [...], shared_name="shared_q")
training_instance = tf.placeholder(tf.float32, shape=[2])
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
    sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
    sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Você pode enfileirar várias instâncias ao mesmo tempo em vez de enfileirá-las uma por uma utilizando a operação `enqueue_many`:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue_many([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
             feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})
```

Ambos os exemplos adicionam os mesmos três tensores à fila.

## Desenfileirando os dados

Para retirar as instâncias da fila, do outro lado, você precisa utilizar a operação `dequeue`:

```
# trainer.py
import tensorflow as tf

q = tf.FIFOQueue(capacity=10, [...], shared_name="shared_q")
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue)) # [1., 2.]
    print(sess.run(dequeue)) # [3., 4.]
    print(sess.run(dequeue)) # [5., 6.]
```

Puxe um minilote inteiro ao mesmo tempo em vez de apenas uma instância por vez utilizando uma operação `dequeue_many` especificando o tamanho do minilote:

```
[...]
batch_size = 2
dequeue_mini_batch = q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch)) # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch)) # bloqueado esperando por outra instância
```

A operação `enqueue` [enfileiramento] será bloqueada até que os itens sejam removidos por uma operação `dequeue` [desenfileiramento] quando uma fila estiver cheia. Da mesma forma, quando uma fila está vazia (ou você está utilizando `dequeue_many()` e há menos itens do que o tamanho do minilote), a operação `dequeue` será bloqueada quando utilizarmos uma operação `enqueue` até que itens suficientes sejam empurrados para a fila.

## Filas de tuplas

Em vez de apenas um único tensor, cada item em uma fila pode ser uma tupla (lista ordenada) de tensores (de vários tipos e formas). Por exemplo, a seguinte fila armazena pares de tensores, um do tipo `int32` e formato `()`, e o outro do tipo `float32` no formato `[3,2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32], shapes=[[[],[3,2]], name="q", shared_name="shared_q")
```

À operação de enfileirar devem ser dados pares de tensores (observe que cada par representa apenas um item na fila):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session(...) as sess:
    sess.run(enqueue, feed_dict={a: 10, b:[[1., 2.], [3., 4.], [5., 6.]]})
    sess.run(enqueue, feed_dict={a: 11, b:[[2., 4.], [6., 8.], [0., 2.]]})
    sess.run(enqueue, feed_dict={a: 12, b:[[3., 6.], [9., 2.], [5., 8.]]})
```

Por outro lado, a função `dequeue()` cria um par de operações de desenfileirar:

```
dequeue_a, dequeue_b = q.dequeue()
```

No geral, você deve executar essas operações em conjunto:

```
with tf.Session(...) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b])
    print(a_val) # 10
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```



Se você executar `dequeue_a` por conta própria, vai desenfileirar um par e retornar apenas o primeiro elemento; o segundo será perdido (da mesma forma, se você rodar `dequeue_b` por conta própria, o primeiro elemento será perdido).

A função `dequeue_many()` também retorna um par de operações:

```
batch_size =
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

Você pode utilizá-la como esperado:

```
with tf.Session(...) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
    print(a) # [10, 11]
    print(b) # [[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]
    a, b = sess.run([dequeue_a, dequeue_b]) # bloqueado esperando por outro par
```

### Fechando uma fila

É possível fechar uma fila para sinalizar para as outras sessões que os dados não serão mais enfileirados:

```
close_q = q.close()

with tf.Session(...) as sess:
    [...]
    sess.run(close_q)
```

As execuções subsequentes das operações `enqueue` ou `enqueue_many` levantarão uma exceção. Por padrão, a menos que você recorra a `q.close(cancel_pending_enqueues=True)`, qualquer requisição pendente enfileirada será atendida.

As execuções subsequentes das operações `dequeue` ou `dequeue_many` continuarão a ser bem-sucedidas desde que haja itens na fila, mas falharão quando não houver itens restantes o suficiente. Se você utilizar a operação `dequeue_many` e houver algumas instâncias deixadas na fila, mas menos que o tamanho do minilote, elas serão perdidas. É melhor usar a operação `dequeue_up_to`, que se comporta exatamente como `dequeue_many` exceto quando uma fila é fechada e há menos destas do que instâncias `batch_size` deixadas na fila, caso em que apenas as retorna.

### RandomShuffleQueue

O TensorFlow também suporta mais alguns tipos de filas, incluindo a `RandomShuffleQueue`, que pode ser utilizada assim como o `FIFOQueue`, mas os itens são desenfileirados de forma aleatória, podendo ser úteis para embaralhar instâncias de treinamento em cada época durante o treinamento. Primeiro, criaremos a fila:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                           dtypes=[tf.float32], shapes=[()],
                           name="q", shared_name="shared_q")
```

O `min_after_dequeue` especifica o número mínimo de itens que devem permanecer na fila após a operação `dequeue` para garantir que haverá instâncias suficientes na fila a ter aleatoriedade suficiente (assim que a fila é fechada, o limite `min_after_dequeue` é ignorado). Agora, suponha que você posicionou 22 itens nesta fila (flutua de 1. a 22.). Veja como você pode desenfileirá-los:

```
dequeue = q.dequeue_many(5)

with tf.Session([...]) as sess:
    print(sess.run(dequeue)) # [ 20. 15. 11. 12. 4.] (faltam 17 itens)
    print(sess.run(dequeue)) # [ 5. 13. 6. 0. 17.] (faltam 12 itens)
    print(sess.run(dequeue)) # 12 - 5 < 10: bloqueado esperando por mais 3 instâncias
```

## PaddingFIFOQueue

Um `PaddingFIFOQueue` também pode ser utilizado como um `FIFOQueue`, só que ele aceita tensores de tamanhos variáveis ao longo de qualquer dimensão (mas com uma classificação fixa). Ao desenfileirá-las em uma operação `dequeue_many`, ou `dequeue_up_to`, cada tensor é preenchido com zeros para torná-lo do mesmo tamanho que o maior tensor no minilote ao longo de cada dimensão variável. Por exemplo, você poderia enfileirar tensores 2D (matrizes) de tamanhos arbitrários:

```
q = tf.PaddingFIFOQueue(capacity=50,
                        dtypes=[tf.float32], shapes=[(None, None)],
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]})      # 3x2
    sess.run(enqueue, feed_dict={v: [[1.]]})                                # 1x1
    sess.run(enqueue, feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

Se retirarmos um item por vez, teremos exatamente os mesmos tensores que foram enfileirados. Mas, se retirarmos vários itens de cada vez, a fila automaticamente preenche os tensores adequadamente (utilizando `dequeue_many()` ou `dequeue_up_to()`). Por exemplo, se desenfileirarmos os três itens ao mesmo tempo, todos os tensores serão preenchidos com zeros para se tornarem  $3 \times 4$ , uma vez que o tamanho máximo para a primeira dimensão é 3 (primeiro item) e o tamanho máximo para a segunda dimensão é 4 (terceiro item).

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
...
[[[ 1.  2.  0.  0.]
 [ 3.  4.  0.  0.]
 [ 5.  6.  0.  0.]]]

[[ 1.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]]

[[ 7.  8.  9.  5.]
 [ 6.  7.  8.  9.]
 [ 0.  0.  0.  0.]]]
```

Este tipo de fila pode ser útil quando você está lidando com entradas de comprimento variável como sequências de palavras (veja o Capítulo 14).

Ok, agora pausaremos por um segundo: até aqui você aprendeu a distribuir cálculos em vários dispositivos e servidores, compartilhar variáveis entre as sessões e se comunicar de maneira assíncrona com a utilização de filas. No entanto, antes de começar a treinar redes neurais, há um último tópico que precisamos discutir: como carregar com eficiência os dados de treinamento.

## Carregando Dados Diretamente do Grafo

Até agora, assumimos que, com a utilização de placeholders, os clientes carregariam os dados de treinamento e os forneceriam para o cluster. Isso é simples e funciona muito bem para configurações simples, mas, uma vez que transfere várias vezes os dados de treinamento, é bastante ineficiente:

1. Do sistema de arquivos para o cliente;
2. Do cliente para a master task;
3. Possivelmente da master task para outras tasks nas quais os dados são necessários.

Isso piora se você tiver vários clientes treinando várias redes neurais usando os mesmos dados de treinamento (por exemplo, para ajuste do hiperparâmetro): se cada cliente carregar simultaneamente os dados, você pode até saturar seu servidor de arquivos ou a largura de banda da rede.

### Pré-carregue os dados em uma variável

Para conjuntos de dados que podem caber na memória, uma opção melhor é carregar os dados de treinamento uma vez, atribuí-los a uma variável e, então, utilizar essa variável

em seu grafo, o que é chamado de *pré-carregamento* do conjunto de treinamento. Desta forma, os dados serão transferidos apenas uma vez do cliente para o cluster (mas ainda podem ser movidos de uma task para outra, dependendo de quais operações necessitarem dele). O código a seguir mostra como carregar o conjunto completo de treinamento em uma variável:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False, collections=[],
                         name="training_set")

with tf.Session([...]) as sess:
    data = [...] # carregue os dados de treinamento do datastore
    sess.run(training_set.initializer, feed_dict={training_set_init: data})
```

Você deve configurar `trainable=False` para que os otimizadores não tentem ajustar esta variável. Você também deve configurar `collections=[]` para garantir que esta variável não seja adicionada à coleção `GraphKeys.GLOBAL_VARIABLES`, que é utilizada para salvar e restaurar pontos de verificação.



Este exemplo assume que todo o seu conjunto de treinamento (incluindo os rótulos) consiste apenas de valores float32. Se esse não for o caso, você precisará de uma variável por tipo.

### Lendo dados de treinamento diretamente do grafo

Se o conjunto de treinamento não cabe na memória, uma boa solução seria utilizar as *operações de leitura*, que conseguem ler dados diretamente do sistema de arquivos. Desse forma, os dados de treinamento nunca precisam fluir pelos clientes. O TensorFlow fornece leitores para vários formatos de arquivo:

- CSV;
- Registros binários de comprimento fixo;
- O formato `TFRecords` do TensorFlow, baseado em buffers de protocolo.

Vejamos um exemplo simples de leitura de um arquivo CSV (para outros formatos, verifique a documentação da API). Suponha que você tenha um arquivo chamado `my_test.csv`, que contém instâncias de treinamento, e você quer criar operações para ler o arquivo. Suponha que ele tenha o seguinte conteúdo, com dois recursos flutuantes `x1` e `x2` e um inteiro `target` representando uma classe binária:

```
x1, x2, target
1. , 2. , 0
4. , 5 , 1
7. ,     , 0
```

Primeiro, criaremos um `TextLineReader` para ler este arquivo. Um `TextLineReader` abre um arquivo (uma vez que dizemos a ele qual abrir) e lê as linhas uma por uma. É uma operação dinâmica, como variáveis e filas: preserva seu estado em múltiplas execuções do grafo acompanhando qual arquivo está lendo atualmente e qual é sua posição atual neste arquivo.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Em seguida, criamos uma fila que o leitor puxará para saber qual arquivo ler em seguida. Para enviar qualquer nome de arquivo que desejarmos para a fila, também criamos uma operação de enfileiramento, um placeholder e uma operação para fechar a fila quando não tivermos mais arquivos a serem lidos:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Agora estamos prontos para criar uma operação de leitura que lerá um registro (ou seja, uma linha) por vez e retornará um par chave/valor. A chave é o identificador exclusivo do registro — uma string composta do nome do arquivo, dois pontos (:) e o número da linha — e o valor é uma string com o conteúdo da linha:

```
key, value = reader.read(filename_queue)
```

Temos tudo o que precisamos para ler o arquivo, linha por linha! Mas ainda não terminamos — precisamos analisar essa string para obter as características e o destino:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
features = tf.stack([x1, x2])
```

A primeira linha utiliza o parser CSV do TensorFlow para extrair os valores da linha atual. Os valores padrão são utilizados quando um campo está ausente (neste exemplo, a característica `x2` da terceira instância de treinamento) e também para determinar o tipo de cada campo (nesse caso, dois flutuantes e um inteiro).

Finalmente, podemos empurrar essa instância de treinamento e seu alvo para um `RandomShuffleQueue`, que compartilharemos com o grafo de treinamento (para extraímos minilotes), e criar uma operação para fechar essa fila quando terminarmos de empurrar as instâncias nela:

```
instance_queue = tf.RandomShuffleQueue(
    capacity=10, min_after_dequeue=2,
    dtypes=[tf.float32, tf.int32], shapes=[[2], []],
    name="instance_q", shared_name="shared_instance_q")
enqueue_instance = instance_queue.enqueue([features, target])
close_instance_queue = instance_queue.close()
```

Ufa! Isso foi muito trabalho para ler apenas um arquivo. Além disso, apenas criamos o grafo, então precisamos executá-lo:

```
with tf.Session([...]) as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    try:
        while True:
            sess.run(enqueue_instance)
    except tf.errors.OutOfRangeError as ex:
        pass # não há mais registros no arquivo atual e não há mais arquivos para ler
    sess.run(close_instance_queue)
```

Primeiro, abrimos a sessão, enfileiramos o arquivo “`my_test.csv`” e imediatamente fechamos essa fila já que não enfileiraremos mais nomes de arquivos. Em seguida, executamos um loop infinito para enfileirar instâncias, uma a uma. A `enqueue_instance` depende que o leitor leia a próxima linha, então um novo registro é lido a cada iteração até chegar ao final do arquivo. Nesse ponto, ele tenta ler a fila `filename` para saber qual arquivo ler em seguida e, como a fila está fechada, lança uma exceção `OutOfRangeError` (se não fecharmos a fila, ela permanecerá bloqueada até que digitemos outro nome de arquivo ou a fechemos). Por fim, fechamos a fila de instâncias para que as operações de treinamento não sejam bloqueadas para sempre. A Figura 12-9 resume o que aprendemos; ela representa um grafo típico para ler instâncias de treinamento em um conjunto de arquivos CSV.

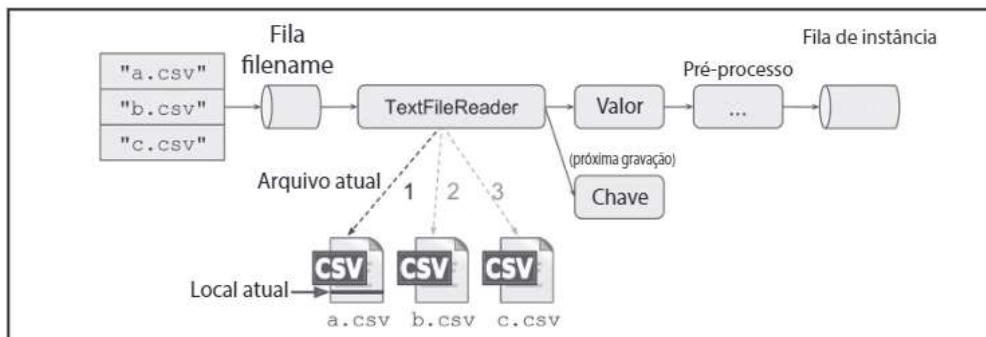


Figura 12-9. Um grafo dedicado à leitura de instâncias de treinamento de arquivos CSV

Você precisa criar a fila de instâncias compartilhadas e desenfileirar os minilotes no grafo do treinamento:

```

instance_queue = tf.RandomShuffleQueue([...], shared_name="shared_instance_q")
mini_batch_instances, mini_batch_targets = instance_queue.dequeue_up_to(2)
[...] # use as instâncias de minilote e alvos para criar o grafo de treinamento
training_op = [...]

with tf.Session(...) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
        pass # não há mais instâncias de treinamento

```

Neste exemplo, o primeiro minilote conterá as duas primeiras instâncias do arquivo CSV e o segundo minilote conterá a última instância.



As filas do TensorFlow não lidam bem com tensores esparsos, portanto, se as instâncias de treinamento forem esparsas, você deverá analisar os registros após a fila de instâncias.

Essa arquitetura utilizará apenas uma thread para ler registros e empurrá-los para a fila de instâncias. Você obtém uma taxa de transferência muito maior ao ter várias threads lidas simultaneamente com a utilização de vários leitores a partir de vários arquivos. Vejamos como.

### Leitores Multithreaded utilizando as classes Coordinator e QueueRunner

Para que várias threads leiam instâncias simultaneamente, você pode criar threads Python (utilizando o módulo `threading`) e gerenciá-las você mesmo. No entanto, o TensorFlow fornece algumas ferramentas que facilitam isso: a classe `Coordinator` e a classe `QueueRunner`.

Um `Coordinator` é um objeto muito simples cujo único propósito é coordenar a interrupção de várias threads. Primeiro você cria um `Coordinator`:

```
coord = tf.train.Coordinator()
```

Então você o fornece a todas as threads que precisam parar juntas, e seu loop principal se parecerá com isso:

```
while not coord.should_stop():
    [...] # faça alguma coisa
```

Ao recorrermos ao método `Coordinator request_stop()`, qualquer thread pode solicitar que todos os segmentos parem:

```
coord.request_stop()
```

Cada thread será interrompida assim que terminar sua iteração atual e você pode esperar que todas finalizem ao chamar o método do `Coordinator join()`:

```
coord.join(list_of_threads)
```

Um `QueueRunner` executa várias threads, cada uma executando repetidamente uma operação de enfileiramento, preenchendo uma fila o mais rápido possível. Assim que a fila é fechada, a próxima thread que tenta enviar um item para a fila receberá um `OutOfRangeException`; este segmento captura o erro e imediatamente informa outras threads para pararem de utilizar um `Coordinator`. O código a seguir mostra como você pode usar um `QueueRunner` para que cinco threads leiam instâncias simultaneamente e as empurrem para uma fila de instâncias:

```
[...] # mesma fase de construção de antes
queue_runner = tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)

with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads = queue_runner.create_threads(sess, coord=coord, start=True)
```

A primeira linha cria o `QueueRunner` e pede que ele execute cinco threads, todas executando repetidamente a mesma operação `enqueue_instance`. Então iniciamos uma sessão e enfileiramos o nome dos arquivos para a leitura (neste caso, apenas “`my_test.csv`”). A seguir, como acabamos de explicar, criamos um `Coordinator` que o `QueueRunner` utilizará para parar suavemente, conforme explicado. Finalmente, dizemos ao `QueueRunner` para criar as threads e iniciá-las. As threads lerão todas as instâncias de treinamento, as enviarão para a fila de instâncias e, em seguida, todas pararão.

Isso será um pouco mais eficiente do que antes, mas podemos melhorar. Atualmente, todas as threads estão lendo do mesmo arquivo. Podemos fazê-las ler simultaneamente a partir de arquivos separados (veja a Figura 12-10) ao criar vários leitores (assumindo que os dados de treinamento são partitionados em vários arquivos CSV).

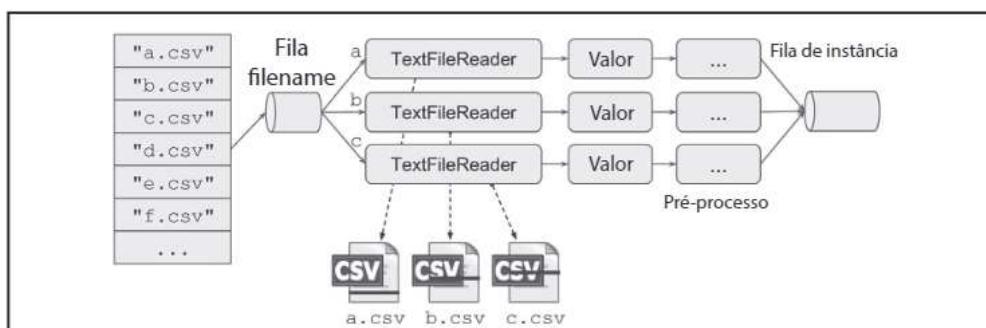


Figura 12-10. Leitura simultânea de vários arquivos

Para isso, precisamos escrever uma pequena função para criar um leitor e os nós que lerão e enviarão uma instância para a fila de instâncias:

```
def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1.]])
    features = tf.stack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance
```

Em seguida, definimos as filas:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string], shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()

instance_queue = tf.RandomShuffleQueue([...])
```

E, finalmente, criamos o `QueueRunner`, mas, desta vez, damos uma lista de diferentes operações de enfileiramento. Cada operação utilizará um leitor diferente para que as threads sejam lidas simultaneamente a partir de arquivos diferentes:

```
read_and_enqueue_ops =
    read_and_push_instance(filename_queue, instance_queue)
    for i in range(5)]
queue_runner = tf.train.QueueRunner(instance_queue, read_and_enqueue_ops)
```

A fase de execução é a mesma de antes: primeiro inserimos os nomes dos arquivos para leitura, criamos um `Coordinator` e criamos e iniciamos as threads `QueueRunner`. Desta vez, todas lerão diferentes arquivos simultaneamente até que todos sejam lidos por completo e, então, o `QueueRunner` fechará a fila de instâncias para que outras operações puxando a partir dele não sejam bloqueadas.

## Outras funções de conveniências

O TensorFlow também oferece algumas funções convenientes para simplificar certas tarefas comuns na leitura de instâncias de treinamento. Veremos apenas alguns exemplos (consulte a documentação da API para obter a lista completa).

O `string_input_producer()` utiliza um tensor 1D contendo uma lista de nomes de arquivos, cria uma thread que envia um nome de arquivo de cada vez para a fila dos nomes e a fecha. Se você especificar um número de épocas, ele percorrerá os nomes dos arquivos uma vez por época antes de fechar a fila. Por padrão, ele embaralha os nomes dos arquivos em cada época, cria um `QueueRunner` para gerenciar sua thread e o adiciona à coleção `Graph Keys.QUEUE_RUNNERS`. Para iniciar cada `QueueRunner` nesta coleção, você pode chamar a função `tf.train.start_queue_runners()`. Observe que,

se você se esquecer de iniciar o `QueueRunner`, a fila do nome do arquivo estará aberta e vazia, e seus leitores estarão bloqueados para sempre.

Para executar uma operação de enfileiramento, existem algumas outras funções `producer` que, de maneira semelhante, criam uma fila e uma `QueueRunner` correspondente (por exemplo, `input_producer()`, `range_input_producer()`, e `slice_input_producer()`).

A função `shuffle_batch()` pega uma lista de tensores (por exemplo, `[features, target]`) e cria:

- Uma `RandomShuffleQueue`;
- Uma `QueueRunner` para enviar os tensores para a fila (adicionado a coleção `GraphKeys.QUEUE_RUNNERS`);
- Uma operação `dequeue_many` para extrair um minilote da fila.

Isso facilita o gerenciamento em um processo único de um pipeline de entrada multithread alimentando uma fila e um pipeline de treinamento lendo minilotes dessa fila. Verifique também as funções `batch()`, `batch_join()` e `shuffle_batch_join()`, que fornecem funcionalidade semelhante.

Ok! Agora você tem todas as ferramentas necessárias em um cluster do TensorFlow para começar a treinar e executar redes neurais eficientemente em vários dispositivos e servidores. Vamos rever o que você aprendeu:

- Utilizar vários dispositivos GPU;
- Configurar e iniciar um cluster do TensorFlow;
- Distribuir cálculos entre vários dispositivos e servidores;
- Compartilhar variáveis (e outras operações de estado como filas e leitores em sessões com a utilização de contêineres);
- Coordenar vários grafos trabalhando de forma assíncrona com a utilização de filas;
- Ler entradas eficientemente utilizando readers, queue runners e coordinators.

Agora, utilizaremos tudo isso na paralelização das redes neurais!

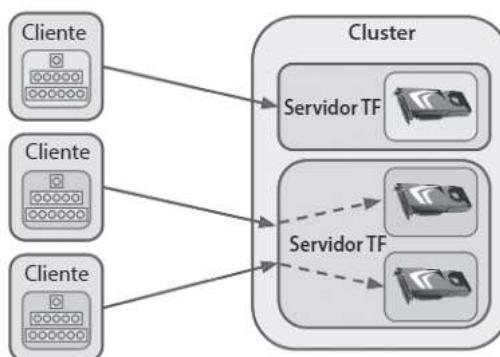
## Paralelizando Redes Neurais em um Cluster do TensorFlow

Nesta seção, primeiro veremos como paralelizar várias redes neurais posicionando cada uma em um dispositivo diferente. Em seguida, veremos um problema muito mais complicado ao treinar uma única rede neural em vários dispositivos e servidores.

### Uma Rede Neural por Dispositivo

A maneira mais trivial de treinar e executar redes neurais em um cluster do TensorFlow é pegar exatamente o mesmo código que você utilizaria para um único dispositivo em uma única máquina e especificar o endereço do servidor principal ao criar a sessão. É só isso! Seu código será executado no dispositivo padrão do servidor. Você pode alterar o dispositivo que executará seu grafo ao simplesmente posicionar a fase de construção do seu código em um bloco de dispositivo.

Ao executar várias sessões do cliente em paralelo (em diferentes threads ou diferentes processos), você pode facilmente treinar ou executar muitas redes neurais em paralelo em todos os dispositivos e em todas as máquinas em seu cluster (veja a Figura 12-11) conectando-os a diferentes servidores e configurando-os para utilizar diferentes dispositivos. A aceleração é quase linear<sup>4</sup>. Treinar 100 redes neurais em 50 servidores com 2 GPUs cada não levará muito mais tempo do que treinar apenas 1 rede neural em 1 GPU.



*Figura 12-11. Treinando uma rede neural por dispositivo*

Esta solução é perfeita para o ajuste do hiperparâmetro: cada dispositivo no cluster treinará um modelo diferente com seu próprio conjunto de hiperparâmetros. Quanto

<sup>4</sup> Não é 100% linear se você esperar que todos os dispositivos terminem, já que o tempo total será o tempo gasto pelo dispositivo mais lento.

mais poder de computação você tiver, maior será o espaço de hiperparâmetro que você pode explorar.

Ela também funcionará perfeitamente se você hospedar um serviço da Web que receba um grande *número de consultas por segundo* (QPS) e precisar da rede neural para fazer uma previsão para cada consulta. Basta replicar a rede neural em todos os dispositivos no cluster e enviar consultas em todos os dispositivos. Ao adicionar mais servidores, você manipula um número ilimitado de QPS (no entanto, isso não reduzirá o tempo necessário para processar uma única solicitação pois ela ainda terá que esperar que uma rede neural faça uma previsão).



Outra opção é utilizar o *TensorFlow Serving* para atender suas redes neurais. É um sistema de código aberto lançado pelo Google em fevereiro de 2016 projetado para atender a um grande volume de consultas a modelos de Aprendizado de Máquina (normalmente construído com o TensorFlow). Ele lida com versões do modelo para que você possa implantar facilmente uma nova versão de sua rede na produção ou experimentar com vários algoritmos sem interromper seu serviço, e ele pode sustentar uma carga pesada ao adicionarmos mais servidores. Para mais detalhes, confira <https://www.tensorflow.org/serving/>.

## Replicação em Grafo Versus Replicação Entre Grafos

Você também pode parallelizar o treinamento de um grande ensemble de redes neurais colocando cada rede neural em um dispositivo diferente (os ensembles foram introduzidos no Capítulo 7). No entanto, quando você quiser executar o ensemble, precisará agregar as previsões individuais feitas por cada rede neural para produzir a previsão do ensemble, e isso requer um pouco de coordenação.

Existem duas abordagens principais para lidar com um ensemble de redes neurais (ou qualquer outro grafo que contenha grandes blocos de cálculos independentes):

- Você pode criar um grande grafo contendo cada rede neural, cada uma fixada em um dispositivo diferente, além dos cálculos necessários para agregar as previsões individuais de todas as redes neurais (veja a Figura 12-12). Em seguida, basta criar uma sessão para qualquer servidor no cluster e deixar que ela cuide de tudo (incluindo aguardar que todas as previsões individuais estejam disponíveis antes de agregá-las). Essa abordagem é conhecida como *in-graph replication*.

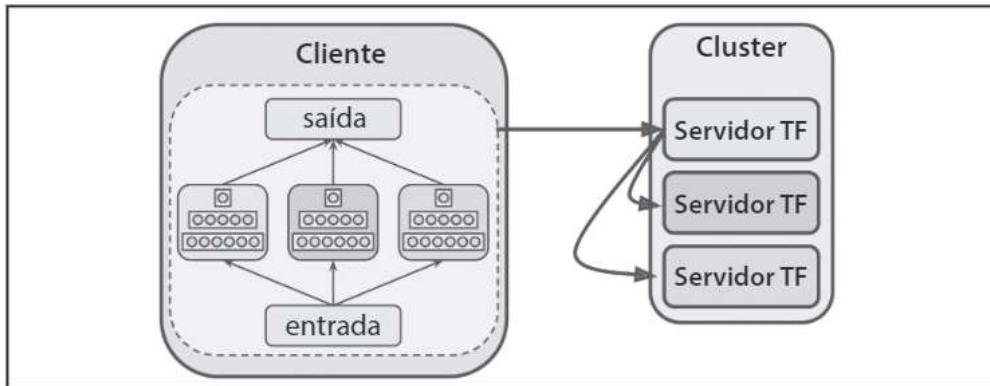


Figura 12-12. In-graph replication

- Como alternativa, você pode criar um grafo separado para cada rede neural e lidar com a sincronização entre esses grafos, abordagem esta chamada de *between-graph replication*. Coordenar a execução desses grafos com a utilização de filas é uma implementação típica (veja a Figura 12-13). Um conjunto de clientes manipula uma rede neural cada um, lendo a partir de sua fila de entrada dedicada e gravando em sua fila de previsão dedicada. Outro cliente é responsável por ler as entradas e mandá-las para todas as filas de entrada (copiando todas as entradas para cada fila). Finalmente, um último cliente é encarregado de ler uma previsão de cada fila de previsão e agregá-las para produzir a previsão do ensemble.

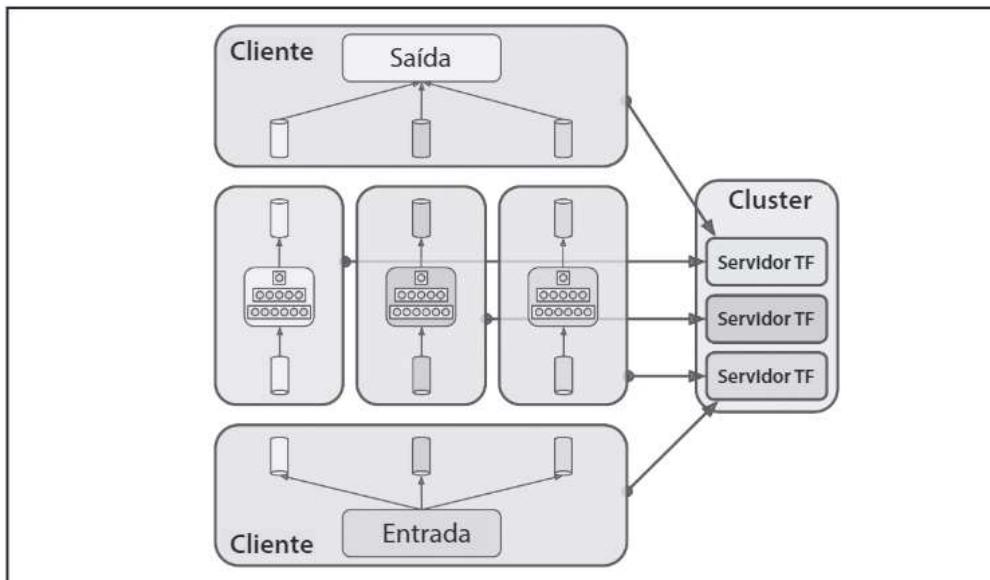


Figura 12-13. Replicação entre grafos

Essas soluções têm seus prós e contras. A implementação da in-graph replication é um pouco mais simples, pois você não precisa gerenciar vários clientes e filas. No entanto, é um pouco mais fácil de organizar em módulos bem delimitados e testar a replicação entre grafos. Além disso, oferece mais flexibilidade. Por exemplo, você poderia adicionar um timeout de desenfileiramento no cliente agregador para que o ensemble não falhasse, mesmo que um dos clientes da rede neural falhasse ou se uma delas demorasse demais para produzir sua previsão. O TensorFlow permite que você especifique o timeout quando chama a função `run()` passando a `RunOptions` com `timeout_in_ms`:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000 # 1s timeout
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # Houve um timeout na operação dequeue depois de 1s
```

Outra forma de especificar um timeout seria definir a opção de configuração `operation_timeout_out_in_ms`, mas, neste caso, a função `run()` expira se *qualquer* operação demorar mais do que o timeout delay:

```
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000 # timeout de 1s para toda operação

with tf.Session(..., config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
    except tf.errors.DeadlineExceededError as ex:
        [...] # houve um timeout na operação dequeue depois de 1s
```

## Paralelismo do Modelo

Até agora, rodamos cada rede neural em um único dispositivo. E se quisermos executar uma única rede neural em vários dispositivos? Isso requer que você recorte seu modelo em pedaços e execute cada um deles em um dispositivo diferente, o que é chamado de *paralelismo do modelo*. Infelizmente, é bem complicado, e depende muito da arquitetura de sua rede neural. Não há muito a ganhar com essa abordagem para redes totalmente conectadas (veja a Figura 12-14). Intuitivamente, pode parecer que uma forma fácil de dividir o modelo seria posicionar cada camada em um dispositivo diferente, mas isso não funciona, já que cada camada precisa esperar pela saída da camada anterior antes de poder fazer qualquer coisa. Então, talvez você pudesse fatiá-la verticalmente — por exemplo, com a metade esquerda de cada camada em um dispositivo e a direita em outro. Isso melhoraria um pouco, já que as duas metades de cada camada podem trabalhar em paralelo, mas o problema é que cada metade da próxima camada vai requerer a saída de

ambas as metades, portanto haverá muita comunicação entre dispositivos (representada pelas setas tracejadas). É provável que isso cancele completamente o benefício da computação paralela, já que a comunicação entre dispositivos é lenta (especialmente se for através de máquinas separadas).

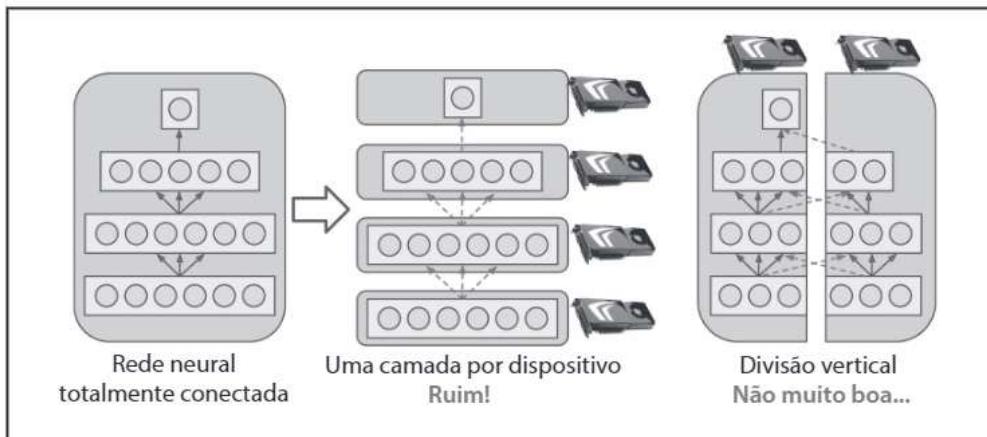


Figura 12-14. Divisão de uma rede neural totalmente conectada

Como veremos no Capítulo 13, algumas arquiteturas de rede neural, como redes neurais convolucionais, contêm camadas que são conectadas apenas parcialmente às camadas inferiores, então têm muito mais facilidade de distribuir os pedaços entre dispositivos de uma forma eficiente.

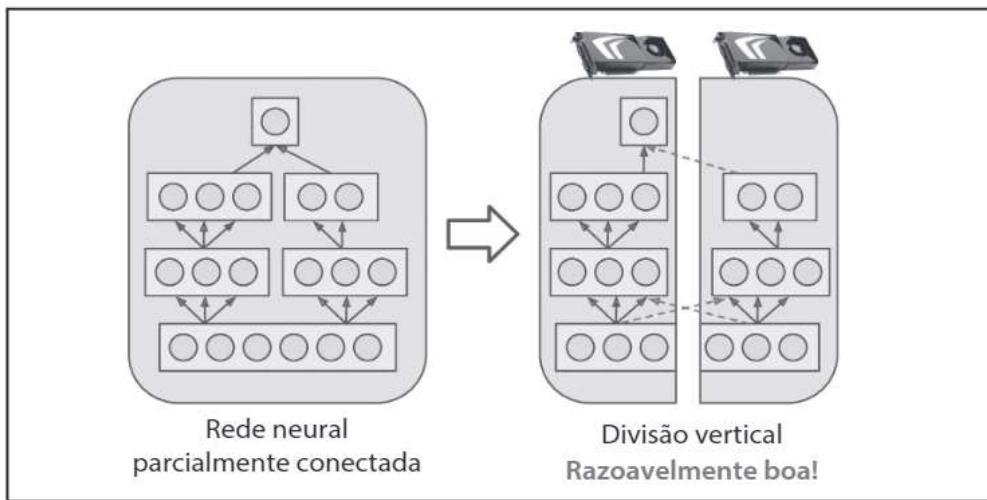


Figura 12-15. Dividindo uma rede neural parcialmente conectada

Além disso, conforme veremos no Capítulo 14, algumas redes neurais recorrentes profundas são compostas de várias camadas de *células de memória* (veja o lado esquerdo da Figura 12-16). A saída de uma célula no momento  $t$  é realimentada para sua entrada no tempo  $t + 1$  (como você pode ver mais claramente no lado direito da Figura 12-16). Se dividirmos tal rede horizontalmente, posicionando cada camada em um dispositivo diferente, então apenas um deles estará ativo no primeiro passo, no segundo passo dois estarão ativos, e, no momento em que o sinal se propaga para a camada de saída, todos os dispositivos estarão ativos simultaneamente. Ainda há muita comunicação acontecendo entre dispositivos, mas, como cada célula pode ser bastante complexa, o benefício de executar várias células em paralelo geralmente supera o ônus da comunicação.

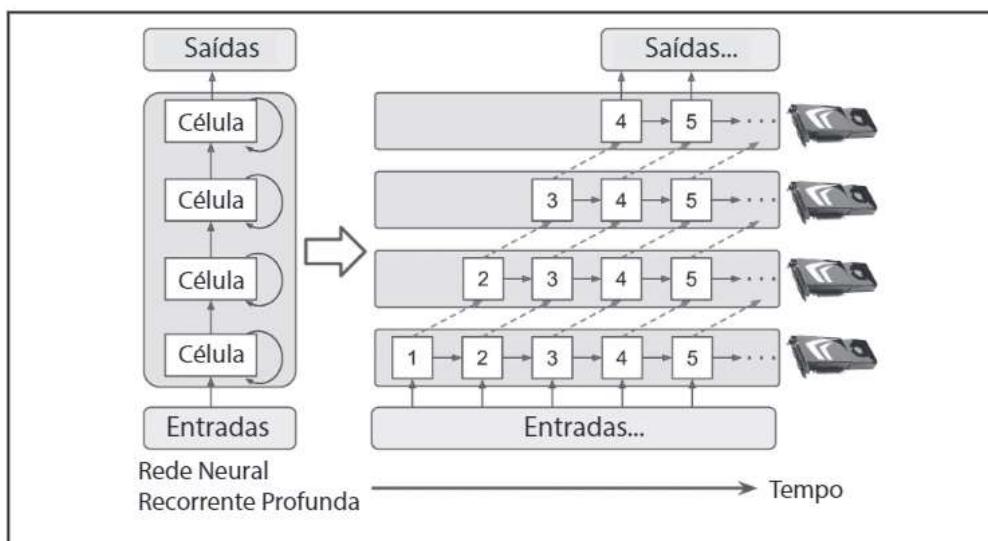


Figura 12-16. Divisão de uma rede neural recorrente profunda

Em suma, o paralelismo do modelo pode acelerar a execução ou o treinamento de alguns tipos de redes neurais, mas não de todos, e requer cuidados e ajustes especiais, como garantir que os dispositivos que precisam se comunicar com mais frequência sejam executados na mesma máquina.

## Paralelismo de Dados

Outra forma de paralelizar o treinamento de uma rede neural é replicá-la em cada dispositivo, executar simultaneamente uma etapa de treinamento utilizando um minilote diferente para cada uma e então agregar os gradientes para atualizar os parâmetros do modelo. Isto é chamado de *paralelismo de dados* (veja a Figura 12-17).

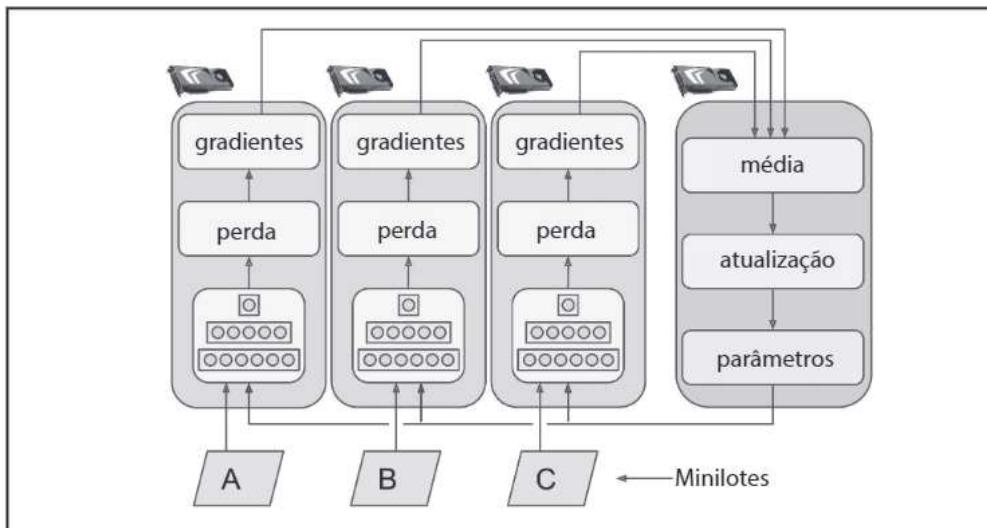


Figura 12-17. Paralelismo de Dados

Existem duas variantes desta abordagem: *atualizações síncronas* e *atualizações assíncronas*.

### Atualizações síncronas

Com atualizações síncronas, o agregador espera que todos os gradientes estejam disponíveis antes de calcular a média e aplicar o resultado (isto é, utilizando os gradientes agregados para atualizar os parâmetros do modelo). Depois que uma réplica termina de calcular seus gradientes, ela deve aguardar a atualização dos parâmetros antes de prosseguir para o próximo minilote. A desvantagem é que alguns dispositivos podem ser mais lentos do que outros, então todos os outros dispositivos terão que esperar por eles a cada etapa. Além disso, os parâmetros serão copiados quase ao mesmo tempo para cada dispositivo (imediatamente após os gradientes serem aplicados), o que pode saturar a largura de banda dos servidores de parâmetros.



Para reduzir o tempo de espera em cada etapa, você poderia ignorar os gradientes em algumas réplicas mais lentas (normalmente ~ 10%). Por exemplo, você poderia executar 20 réplicas, mas agragar apenas os gradientes das 18 réplicas mais rápidas em cada etapa e simplesmente ignorar os gradientes das 2 últimas. Assim que os parâmetros forem atualizados, as primeiras 18 réplicas poderão começar a funcionar novamente de imediato, sem ter que esperar pelas 2 réplicas mais lentas. Essa configuração é descrita como tendo 18 réplicas mais 2 réplicas *sobressalentes*.<sup>5</sup>

<sup>5</sup> Esse nome é um pouco confuso, pois parece que algumas réplicas são especiais, sem fazer nada. Na realidade, todas as réplicas são equivalentes: todas trabalham duro para estarem entre as mais rápidas em cada etapa de treinamento, e os perdedores variam a cada passo de treinamento (a menos que alguns dispositivos sejam realmente mais lentos do que outros).

## Atualizações assíncronas

Com atualizações assíncronas, sempre que uma réplica termina de calcular os gradientes, ela os utiliza imediatamente para atualizar os parâmetros do modelo. Não há agregação (remova a etapa “média” na Figura 12-17) e não há sincronização. As réplicas funcionam independentemente e, como não há espera pelas outras réplicas, essa abordagem executa mais etapas de treinamento por minuto. Além disso, embora os parâmetros ainda precisem ser copiados para cada dispositivo em cada etapa, isso acontece em momentos diferentes para cada réplica, de modo que o risco de saturação da largura de banda é reduzido.

O paralelismo de dados com atualizações assíncronas é uma opção atrativa devido à sua simplicidade, ausência de atraso na sincronização e melhor uso da largura de banda. No entanto, embora funcione razoavelmente bem, na prática é quase surpreendente que funcione! De fato, quando uma réplica termina de calcular os gradientes com base em alguns valores de parâmetros, esses parâmetros terão sido atualizados várias vezes por outras réplicas (em média,  $N - 1$  vezes se houver  $N$  réplicas) e não há garantia de que os gradientes ainda estarão apontando na direção certa (veja a Figura 12-18). Eles serão chamados de *gradientes obsoletos* quando estiverem severamente desatualizados: introduzindo efeitos de ruído e oscilação, podem retardar a convergência (a curva de aprendizado pode conter oscilações temporárias) ou até fazer com que o algoritmo de treinamento possa divergir.

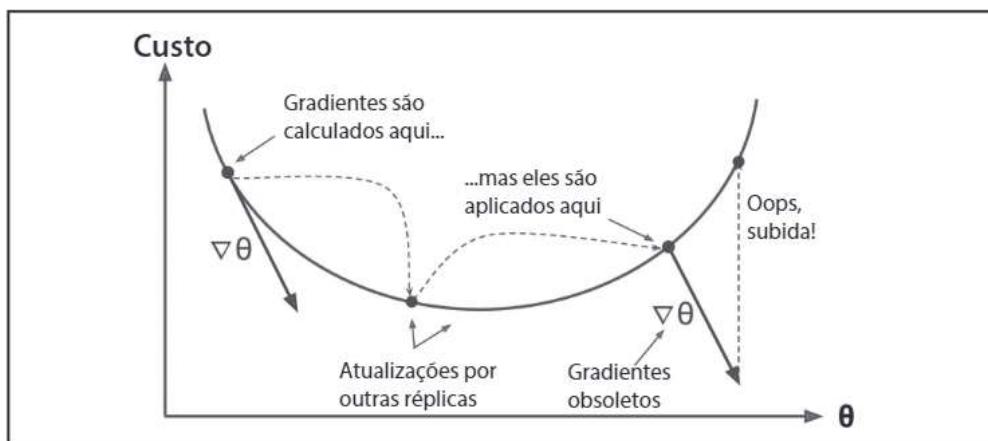


Figura 12-18. Gradientes obsoletos quando utilizam atualizações assíncronas

Existem algumas maneiras de reduzir o efeito dos gradientes obsoletos:

- Reduzir a taxa de aprendizado;
- Descartar gradientes obsoletos ou reduzi-los;
- Ajustar o tamanho do minibatch;

- Começar as primeiras épocas utilizando apenas uma réplica, isso é chamado de *fase de aquecimento*. Gradientes obsoletos tendem a ser mais prejudiciais no início do treinamento, quando os gradientes são tipicamente grandes e os parâmetros ainda não se estabeleceram em um vale da função de custo, portanto réplicas diferentes podem empurrar os parâmetros em direções bem diferentes.

Um artigo publicado pela equipe do *Google Brain* em abril de 2016 (<http://goo.gl/9GCiPb>) comparou várias abordagens e descobriu que o paralelismo de dados, com atualizações síncronas que utilizam algumas réplicas sobressalentes, era o mais eficiente não apenas por convergir mais rápido, mas também por produzir um modelo melhor. No entanto, esta ainda é uma área ativa de pesquisa, portanto você ainda não deve descartar as atualizações assíncronas.

### Saturação da largura de banda

Quer você utilize atualizações síncronas ou assíncronas, o paralelismo de dados ainda requer a comunicação dos parâmetros do modelo vindos dos servidores de parâmetros para cada réplica ao início de cada etapa de treinamento, e os gradientes na outra direção ao final de cada etapa. Infelizmente, isso significa que sempre haverá um ponto em que adicionar uma GPU extra não melhorará o desempenho, pois o tempo gasto movendo os dados para dentro e fora da GPU RAM (e possivelmente através da rede) será maior do que a aceleração obtida pela divisão da carga computacional. Nesse ponto, adicionar mais GPUs aumentará a saturação e retardará o treinamento.



É melhor treinar alguns modelos, geralmente relativamente pequenos e treinados em um conjunto de treinamento muito grande, em uma única máquina com uma única GPU.

A saturação é mais severa para modelos grandes e densos, pois eles têm muitos parâmetros e gradientes para transferir. Ela será menos severa para modelos pequenos (mas o ganho da paralelização é pequeno) e também em grandes modelos esparsos, já que os gradientes geralmente são zeros e podem ser comunicados de forma eficiente. Jeff Dean, iniciador e líder do projeto do *Google Brain*, relatou (<http://goo.gl/E4ypxo>) acelerações típicas de 25 a 40x ao distribuir cálculos em 50 GPUs para modelos densos, e uma aceleração de 300x para modelos mais esparsos treinados em 500 GPUs. Como você pode ver, os modelos esparsos realmente escalonam melhor. Veja alguns exemplos concretos:

- Tradução da máquina neural: aceleração de 6x em 8 GPUs;
- Inception/ImageNet: aceleração de 32x em 50 GPUs;

- RankBrain: 300x aceleração em 500 GPUs.

Esses números representam o que há de melhor no primeiro trimestre de 2016. Além de algumas dúzias de GPUs, a saturação entra em ação e o desempenho diminui para um modelo denso ou algumas centenas de GPUs para um modelo esparsos. Há muitas pesquisas em andamento para resolver esse problema (explorando arquiteturas ponto a ponto em vez de servidores de parâmetros centralizados, utilizando compactação do modelo de perdas, otimizando quando e do que as réplicas precisam para se comunicar etc.), portanto haverá muito progresso na paralelização de redes neurais nos próximos anos.

Enquanto isso, siga estes simples passos para reduzir o problema da saturação:

- Agrupe suas GPUs em alguns servidores em vez de dispersá-las em vários, e isso evitará saltos desnecessários da rede;
- Particione os parâmetros em vários servidores de parâmetros (conforme discutido anteriormente);
- Troque a precisão de flutuação dos parâmetros do modelo de 32 bits (`tf.float32`) para modelos 16 bits (`tf.bfloat16`), reduzindo pela metade a quantidade de dados a transferir sem muito impacto na taxa de convergência ou no desempenho do modelo.



Embora a precisão de 16 bits seja o mínimo para treinar uma rede neural, você pode diminuir essa precisão para 8 bits após o treinamento para reduzir o tamanho do modelo e acelerar os cálculos. Isso é chamado de *quantizar* a rede neural. É particularmente útil para implantar e executar modelos pré-treinados em telefones móveis. Veja o ótimo post de Pete Warden (<http://goo.gl/09Cb6v>) sobre o assunto.

## Implementação do TensorFlow

Ao utilizar o TensorFlow para implementar o paralelismo de dados, você primeiro precisa escolher se deseja a replicação no grafo ou a replicação entre grafos e se deseja atualizações síncronas ou assíncronas. Vejamos como você implementaria cada combinação (veja os exercícios e os notebooks do Jupyter para exemplos completos de código).

Com a replicação no grafo + atualizações síncronas, você constrói um grande grafo contendo todas as réplicas do modelo (posicionadas em diferentes dispositivos) e alguns nós para agregar todos os seus gradientes e fornecê-los para um otimizador. Seu código abre uma sessão para o cluster e executa a operação de treinamento repetidamente.

Com replicação no grafo + atualizações assíncronas, você também cria um grafo grande, mas com um otimizador por réplica, e roda uma thread por réplica, executando repetidamente o otimizador da réplica.

Com replicação entre grafos + atualizações assíncronas, você executa vários clientes independentes (normalmente em processos separados), cada um treinando a réplica do modelo como se estivesse sozinho no mundo, mas os parâmetros são realmente compartilhados com outras réplicas (utilizando um contêiner de recursos).

Com a replicação entre grafos + atualizações síncronas, mas uma vez você executa vários clientes, cada um treinando uma réplica de modelo com base em parâmetros compartilhados, mas, desta vez, você envolve o otimizador (por exemplo, um `MomentumOptimizer`) em um `SyncReplicasOptimizer`. Cada réplica utiliza esse otimizador como utilizaria qualquer outro, mas nos bastidores ele envia os gradientes para um conjunto de filas (uma por variável) que é lido por um dos `SyncReplicasOptimizer` da réplica, chamado de *chefe*. O chefe agrupa os gradientes e os aplica, em seguida grava um token em uma *fila de tokens* para cada réplica sinalizando que ele pode ir adiante e calcular os gradientes seguintes. Essa abordagem suporta *réplicas sobressalentes*.

Se fizer os exercícios, você implementará cada uma dessas quatro soluções. Poderá facilmente aplicar o que aprendeu para treinar grandes redes neurais profundas em dezenas de servidores e GPUs! Nos próximos capítulos, falaremos de algumas das arquiteturas de redes neurais mais importantes antes de abordar o Aprendizado por Reforço.

## Exercícios

1. O que provavelmente está acontecendo se você receber um `CUDA_ERROR_OUT_OF_MEMORY` ao iniciar seu TensorFlow? O que pode ser feito em relação a isso?
2. Qual é a diferença entre fixar uma operação em um dispositivo e posicionar uma operação em um dispositivo?
3. Se você estiver executando uma instalação do TensorFlow habilitada para GPU e utilizar apenas o posicionamento padrão, todas as operações serão posicionadas na primeira GPU?
4. Se você fixar uma variável na “`/gpu:0`”, ela pode ser utilizada por operações posicionadas na `/gpu:1`? Ou por operações posicionadas na “`/cpu:0`”? Ou por operações fixadas em dispositivos localizados em outros servidores?
5. Duas operações posicionadas no mesmo dispositivo podem rodar em paralelo?
6. O que é uma dependência de controle e quando você a utiliza?

7. Suponha que você treine uma DNN por dias em um cluster do TensorFlow e, imediatamente após o término do seu programa de treinamento, percebe que esqueceu de salvar o modelo ao utilizar um `Saver`. Seu modelo treinado foi perdido?
8. Treine várias DNNs em paralelo em um cluster do TensorFlow com a utilização de diferentes valores dos hiperparâmetros. Podem ser DNNs para classificação MNIST ou qualquer outra tarefa do seu interesse. A opção mais simples seria escrever um único programa cliente que treine apenas uma DNN e execute esse programa em vários processos em paralelo com diferentes valores dos hiperparâmetros para cada cliente. O programa deve ter opções de linha de comando para controlar em qual servidor e dispositivo a DNN deve ser posicionada e quais contêineres de recursos e valores dos hiperparâmetros devem ser utilizados (certifique-se de utilizar um contêiner de recursos diferente para cada DNN). Utilize um conjunto de validação ou validação cruzada para selecionar os três principais modelos.
9. Crie um ensemble utilizando os três principais modelos do exercício anterior. Defina-o em um único grafo garantindo que cada DNN seja executada em um dispositivo diferente. Avalie no conjunto de validação: o ensemble tem um desempenho melhor do que as DNNs individuais?
10. Treine uma DNN utilizando replicação entre grafos e paralelismo de dados com atualizações assíncronas, calculando quanto tempo leva para atingir um desempenho satisfatório. Em seguida, tente novamente utilizando atualizações síncronas. Atualizações síncronas produzem um modelo melhor? O treinamento está mais rápido? Divida a DNN verticalmente, coloque cada fatia vertical em um dispositivo diferente e treine o modelo novamente. O treinamento ficou mais rápido? O desempenho ficou diferente?

Soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 13

# Redes Neurais Convolucionais (CNN)

Embora o supercomputador Deep Blue da IBM tenha vencido o campeão mundial de xadrez Garry Kasparov em 1996, até recentemente os computadores não conseguiam executar tarefas aparentemente triviais como detectar um filhote em uma foto ou reconhecer palavras faladas de forma confiável. Por que essas tarefas são tão fáceis para nós humanos? A resposta está no fato de a percepção ocorrer em grande parte fora do domínio de nossa consciência, dentro de módulos sensoriais auditivos e visuais especializados em nossos cérebros. Quando a informação sensorial chega à nossa consciência, ela já está adornada com características de alto nível; por exemplo, quando você olha para uma foto de um filhote fofo, você não pode escolher *não* ver o filhote ou *não* notar sua fofura, assim como também não consegue explicar *como* reconhece um filhote fofo; é apenas óbvio para você. Assim, não podemos confiar em nossa experiência subjetiva: a percepção não é nada trivial e, para entendê-la, devemos observar como os módulos sensoriais funcionam.

As *Redes Neurais Convolucionais* (CNNs, em inglês) emergiram do estudo do córtex visual do cérebro e têm sido utilizadas no reconhecimento de imagens desde os anos 1980. Nos últimos anos, graças ao aumento do poder computacional, a quantidade de dados de treinamento disponível e os truques apresentados no Capítulo 11 para treinamento de redes profundas, as CNNs conseguiram alcançar um desempenho sobre-humano em algumas tarefas visuais complexas. Elas fornecem serviços de pesquisa de imagens, carros autônomos, sistemas automáticos de classificação de vídeo e muito mais. Além disso, as CNNs não estão restritas à percepção visual: sendo também bem-sucedidas em outras tarefas, como *reconhecimento de voz* ou *processamento de linguagem natural* (PLN); no entanto, nos concentraremos apenas em aplicações visuais por enquanto.

Neste capítulo, apresentaremos de onde vieram as CNNs, como são seus blocos de construção e como implementá-las com a utilização do TensorFlow. Em seguida, apresentaremos algumas das melhores arquiteturas das CNNs.

## A Arquitetura do CórTEX Visual

David H. Hubel e Torsten Wiesel realizaram uma série de experimentos com gatos em 1958 (<http://goo.gl/VLxXf9>)<sup>1</sup> e 1959 (<http://goo.gl/OYuFUZ>)<sup>2</sup> (e alguns anos depois em macacos (<http://goo.gl/95F7QH>)<sup>3</sup>), que proporcionou aprendizados cruciais sobre a estrutura do córtex visual (os autores receberam o Prêmio Nobel de Fisiologia e Medicina em 1981 por seu trabalho). Em particular, eles mostraram que muitos neurônios no córtex visual têm um pequeno *campo receptivo local*, o que significa que eles reagem apenas a estímulos visuais localizados em uma região limitada do campo visual (veja a Figura 13-1, na qual os campos receptivos locais de cinco neurônios são representados por círculos tracejados). Os campos receptivos de diferentes neurônios podem se sobrepor e, juntos, formam todo o campo visual. Além disso, os autores mostraram que alguns neurônios reagem apenas a imagens nas linhas horizontais, enquanto outros reagem apenas a linhas com diferentes orientações (dois neurônios podem ter o mesmo campo receptivo, mas reagem a diferentes orientações da linha). Eles também notaram que alguns neurônios têm campos receptivos maiores e reagem a padrões mais complexos que são combinações dos padrões de nível inferior. Essas observações levaram à ideia de que os neurônios de nível superior são baseados nas saídas dos neurônios vizinhos de nível inferior (na Figura 13-1, observe que cada neurônio está conectado a apenas alguns neurônios da camada anterior). Esta poderosa arquitetura é capaz de detectar todos os tipos de padrões complexos em qualquer área do campo visual.

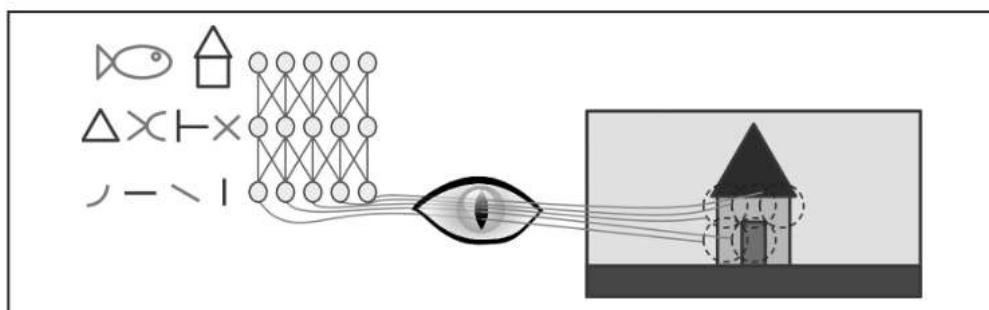


Figura 13-1. Campos receptivos locais no córtex visual

Esses estudos do córtex visual inspiraram o *neocognitron*, introduzido em 1980 (<http://goo.gl/XwiXs9>)<sup>4</sup>, que gradualmente evoluiu para o que hoje chamamos de *redes neurais*

1 “Single Unit Activity in Striate Cortex of Unrestrained Cats,” D. Hubel e T. Wiesel (1958).

2 “Receptive Fields of Single Neurones in the Cat’s Striate Cortex,” D. Hubel e T. Wiesel (1959).

3 “Receptive Fields and Functional Architecture of Monkey Striate Cortex,” D. Hubel e T. Wiesel (1968).

4 “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”, K. Fukushima (1980).

*convolucionais*. Um marco importante foi um artigo de 1998 (<http://goo.gl/A347S4>)<sup>5</sup> de Yann LeCun, Léon Bottou, Yoshua Bengio e Patrick Haffner, que introduziu a famosa arquitetura LeNet-5 amplamente utilizada para reconhecer números em cheques manuscritos. Essa arquitetura tem alguns blocos de construção que você já conhece, como camadas totalmente conectadas e funções de ativação sigmoide, mas também apresenta dois novos blocos de construção: *camadas convolucionais* e *camadas de pooling*. Vamos conhecê-las agora



Por que não simplesmente utilizar uma rede neural profunda regular, com camadas totalmente conectadas, para tarefas de reconhecimento de imagem? Infelizmente, embora isso funcione bem para pequenas imagens (por exemplo, MNIST), falha em imagens maiores devido ao grande número de parâmetros necessários. Por exemplo, uma imagem  $100 \times 100$  tem 10 mil pixels e, se a primeira camada tiver apenas 1 mil neurônios (o que já restringe severamente a quantidade de informações transmitidas para a próxima camada), isso significa um total de 10 milhões de conexões. E essa é apenas a primeira camada. As CNNs resolvem este problema ao utilizar camadas parcialmente conectadas.

## Camada Convolucional

O bloco de construção mais importante de uma CNN é a *camada convolucional*:<sup>6</sup> os neurônios na primeira camada convolucional não estão conectados a cada pixel na imagem de entrada (como estavam nos capítulos anteriores), mas apenas a pixels em seus campos receptivos (veja a Figura 13-2). Por sua vez, cada neurônio na segunda camada convolucional está conectado apenas a neurônios localizados dentro de um pequeno retângulo na primeira camada. Essa arquitetura permite que a rede se concentre em características de baixo nível na primeira camada oculta e, em seguida, os reúna em características de nível superior na próxima camada oculta, e assim por diante. Essa estrutura hierárquica é comum em imagens do mundo real, e é uma das razões pelas quais as CNNs funcionam tão bem para o reconhecimento de imagens.

5 “Gradient-Based Learning Applied to Document Recognition”, Y.LeCun *et al.* (1998).

6 Uma convolução é uma operação matemática que desliza uma função sobre a outra e mede a integral da sua multiplicação pontual. Possui conexões profundas com a transformada de Fourier e a transformada de Laplace e é muito utilizada no processamento de sinais. As camadas convolucionais realmente utilizam correlações cruzadas, que são muito semelhantes às convoluções (consulte <http://goo.gl/HAfxXd> para obter mais detalhes).

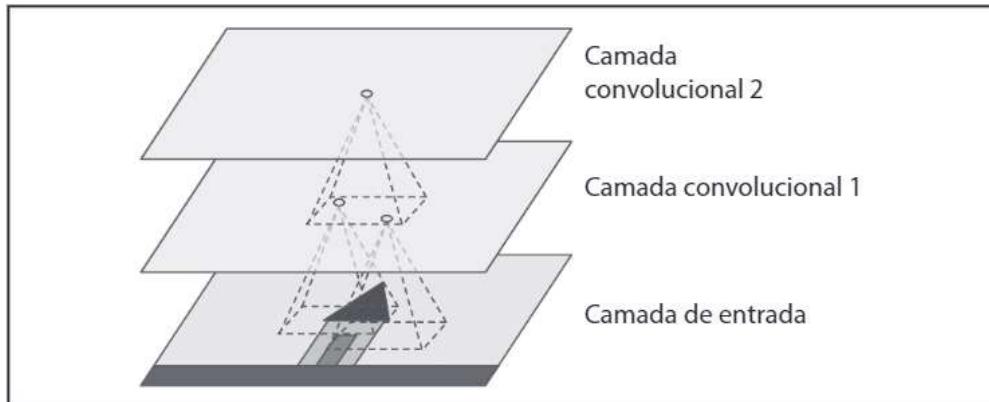


Figura 13-2. Camadas CNN com campos receptivos locais retangulares



Até agora, todas as redes neurais multicamadas que analisamos tinham camadas compostas por uma longa linha de neurônios, e tivemos que nivelar as imagens de entrada para 1D antes de fornecê-las à rede neural. Agora, cada camada é representada em 2D, o que facilita a combinação dos neurônios com suas entradas correspondentes.

Um neurônio localizado na linha  $i$ , coluna  $j$  de uma determinada camada está conectado às saídas dos neurônios na camada anterior localizada nas linhas  $i$  a  $i + f_h - 1$ , colunas  $j$  a  $j + f_w - 1$ , sendo que  $f_h$  e  $f_w$  são a altura e a largura do campo receptivo (veja a Figura 13-3). É comum adicionar zeros ao redor das entradas para que uma camada tenha a mesma altura e largura da camada anterior, conforme mostrado no diagrama, o que é chamado de *zero padding*.

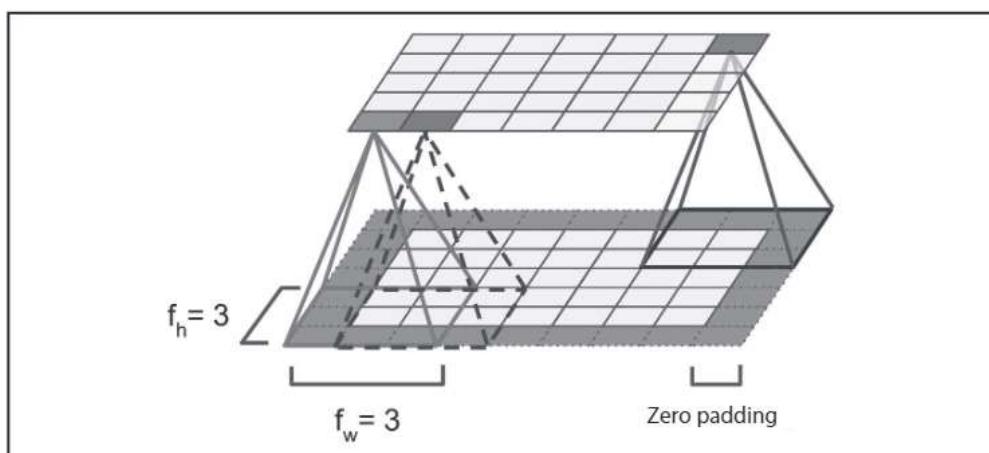


Figura 13-3. Conexões entre camadas e zero padding

Também é possível conectar uma grande camada de entrada a uma camada muito menor ao espaçar os campos receptivos, como mostrado na Figura 13-4. A distância entre dois campos receptivos consecutivos é chamada de *stride*. No diagrama, uma camada de entrada  $5 \times 7$  (mais preenchimento com zeros) é conectada a uma camada  $3 \times 4$  usando campos receptivos  $3 \times 3$  e um stride de 2 (neste exemplo, o stride é o mesmo em ambas as direções, mas não precisa ser assim). Um neurônio localizado na linha  $i$ , coluna  $j$  na camada superior está ligado às saídas dos neurônios na camada anterior localizada nas linhas  $i \times s_h$  a  $i \times s_h + f_h - 1$ , colunas  $j \times s_w$  a  $j \times s_w + f_w - 1$ , em que  $s_h$  e  $s_w$  são os strides verticais e horizontais.

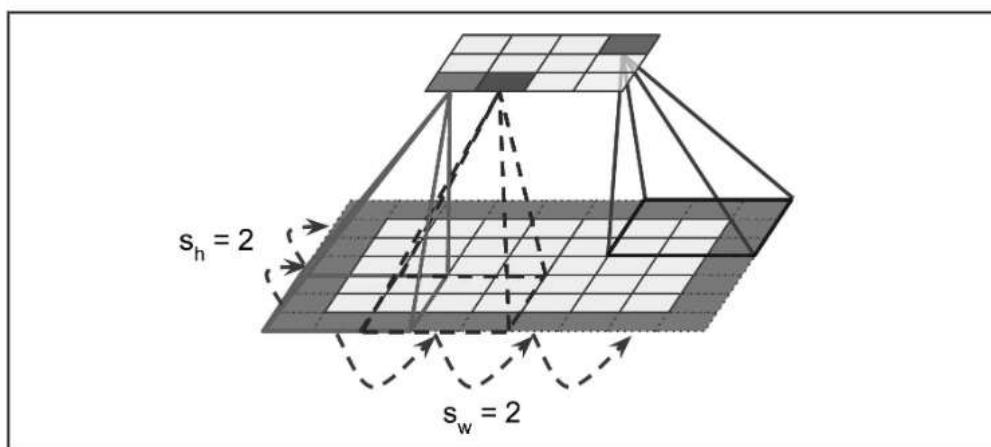


Figura 13-4. Reduzindo a dimensionalidade utilizando um stride de 2

## Filtros

Os pesos de um neurônio podem ser representados como uma pequena imagem do tamanho do campo receptivo. Por exemplo, a Figura 13-5 mostra dois conjuntos possíveis de pesos chamados *filtros* (ou *kernels de convolução*). O primeiro é representado como um quadrado preto com uma linha branca vertical no meio (é uma matriz  $7 \times 7$  cheia de 0s, exceto pela coluna central, que é cheia de 1s); os neurônios que utilizam esses pesos ignorarão tudo em seu campo receptivo, exceto a linha vertical central (já que todas as entradas serão multiplicadas por 0 exceto aquelas localizadas na linha vertical central). O segundo filtro é um quadrado preto com uma linha branca horizontal no meio. Mais uma vez, os neurônios que utilizam esses pesos ignorarão tudo em seu campo receptivo, exceto a linha horizontal central.

Agora, se todos os neurônios na camada utilizarem o mesmo filtro de linha vertical (e o mesmo termo de polarização), e se você alimentar a rede com a imagem de entrada mostrada na Figura 13-5, a camada exibirá a imagem superior esquerda. Observe que as

linhas brancas verticais são aprimoradas enquanto o restante fica desfocado. Da mesma forma, se todos os neurônios utilizarem o filtro de linha horizontal, a imagem superior direita é o que você obtém; observe que as linhas brancas horizontais são aprimoradas enquanto o restante é desfocado. Assim, uma camada cheia de neurônios, com a utilização do mesmo filtro, fornece um *mapa de características* que destaca as áreas de uma imagem mais semelhantes ao filtro. Durante o treinamento, uma CNN encontra os filtros mais úteis para sua tarefa e aprende a combiná-los em padrões mais complexos (por exemplo, uma cruz é uma área em uma imagem em que o filtro vertical e o filtro horizontal estão ativos).

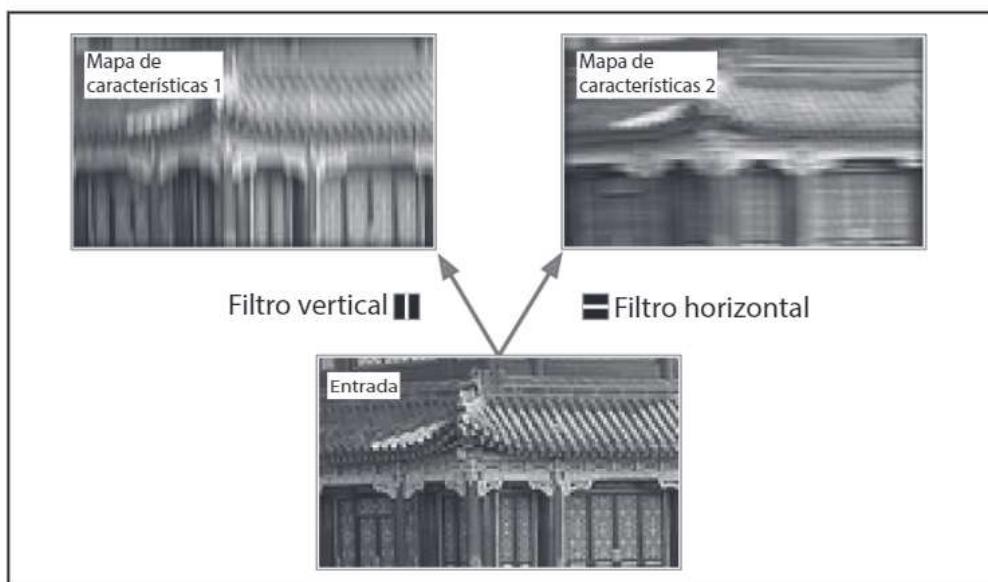


Figura 13-5. Aplicando dois filtros diferentes para obter dois mapas de características

## Empilhando Múltiplos Mapas de Características

Até agora, para simplificar, representamos cada camada convolucional como uma fina camada 2D, mas na realidade ela é composta por vários mapas de tamanhos iguais, então ela é representada com mais precisão em 3D (veja a Figura 13-6). Em um mapa de características, todos os neurônios compartilham os mesmos parâmetros (pesos e termos de polarização), mas diferentes mapas podem ter parâmetros diferentes. O campo receptivo de um neurônio é o mesmo conforme descrito anteriormente, e se estende a todos os mapas das camadas anteriores. Em suma, uma camada convolucional aplica simultaneamente vários filtros às suas entradas, tornando-a capaz de detectar várias características em qualquer lugar em suas entradas.



O fato de que todos os neurônios em um mapa de características compartilham os mesmos parâmetros reduz drasticamente o número de parâmetros no modelo, mas o mais importante é que assim que uma CNN aprende a reconhecer um padrão em um local, ela poderá reconhecê-lo em qualquer outro local. Em contraste, uma vez que uma DNN normal aprende a reconhecer um padrão em um local, ela só consegue reconhecê-lo neste local específico.

Além disso, as imagens de entrada também são compostas de múltiplas subcamadas: uma por *canal de cor*. Normalmente, existem três: vermelho, verde e azul (RGB). As imagens em escala de cinza têm apenas um canal, mas algumas podem ter muito mais, por exemplo, imagens de satélite que capturam frequências extras de luz (como o infravermelho).

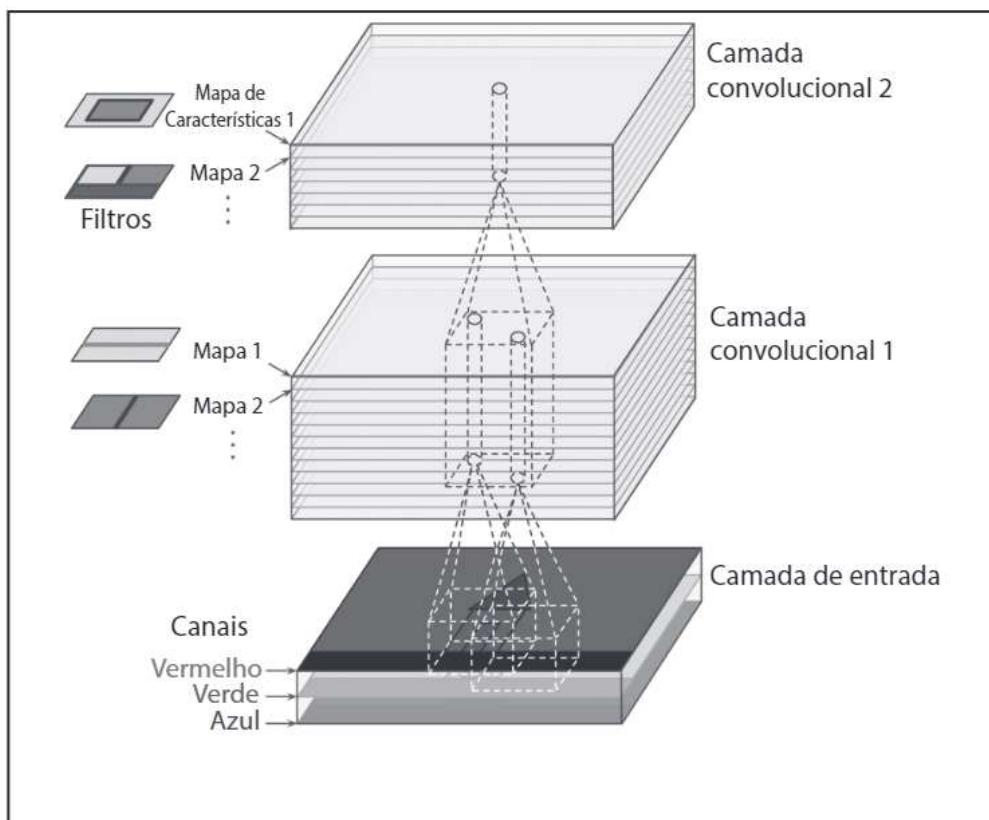


Figura 13-6. Camadas de convolução com vários mapas de características e imagens com três canais

Especificamente, um neurônio localizado na linha  $i$ , coluna  $j$  do mapa de característica  $k$  em uma determinada camada convolucional  $l$  está conectado às saídas dos neurônios na camada anterior  $l - 1$ , localizado nas linhas  $i \times s_h$  a  $i \times s_h + f_h - 1$  e colunas  $j \times s_w$  a  $j \times s_w + f_w - 1$  em todos os mapas de características (na camada  $l - 1$ ). Note que todos os

neurônios localizados na mesma linha  $i$  e coluna  $j$ , mas em mapas de características diferentes, são conectados às saídas dos mesmos neurônios da camada anterior.

A Equação 13-1 resume as explicações precedentes em uma grande equação matemática que mostra como calcular a saída de um dado neurônio em uma camada convolucional.

Devido aos diferentes índices, ela é um pouco estranha, mas tudo o que ela faz é calcular a soma ponderada de todas as entradas, mais o termo de polarização.

*Equação 13-1. Calculando a saída de um neurônio em uma camada convolucional*

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{com} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$  é a saída do neurônio localizado na linha  $i$ , coluna  $j$  no mapa de características  $k$  da camada convolucional (camada  $l$ );
- Como explicado anteriormente,  $s_h$  e  $s_w$  são os strides verticais e horizontais,  $f_h$  e  $f_w$  são a altura e a largura do campo receptivo e  $f_{n'}$  é o número de mapas de características na camada anterior (camada  $l - 1$ );
- $x_{i',j',k'}$  é a saída do neurônio localizado na camada  $l - 1$ , linha  $i'$ , coluna  $j'$ , mapa de características  $k'$  (ou canal  $k'$  se a camada anterior for a camada de entrada);
- $b_k$  é o termo de polarização para o mapa de características  $k$  (na camada  $l$ ). Você pode encarar isso como um botão que altera o brilho geral do mapa de características  $k$ ;
- $w_{u,v,k',k}$  é o peso da conexão entre qualquer neurônio no mapa de características  $k$  da camada  $l$  e sua entrada localizada na linha  $u$ , coluna  $v$  (relativa ao campo receptivo dos neurônios) e ao mapa de características  $k'$ .

## Implementação do TensorFlow

Cada imagem de entrada no TensorFlow é representada como um tensor de formato 3D [`height, width, channels`] e um minilote é representado como um tensor de formato 4D [`mini-batch size, height, width, channels`]. Os pesos de uma camada convolucional são representados como um tensor de formato 4D [ $f_h, f_w, f_{n'}, f_n$ ], já seus termos de polarização são representados como um tensor de formato 1D [ $f_n$ ].

Analisaremos um exemplo simples. O código a seguir carrega duas imagens de amostra do Scikit-Learn com a utilização do `load_sample_images()` (que carrega duas imagens coloridas, sendo uma de um templo chinês e outra de uma flor). Ele cria dois filtros  $7 \times 7$  (um com uma linha branca vertical no meio e outro com uma linha branca horizontal no meio) e aplica a ambas as imagens usando uma camada convolucional construída

utilizando a função `tf.nn.conv2d()` do TensorFlow (com zero padding e um stride de 2). Por fim, plota um dos mapas de características resultantes (semelhante à imagem superior direita na Figura 13-5).

```
import numpy as np
from sklearn.datasets import load_sample_images

# Carrega imagens amostrais
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")
dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Cria 2 filtros
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

# Cria um grafo com entrada X mais uma camada convolucional aplicando os 2 filtros
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1,2,2,1], padding="SAME")

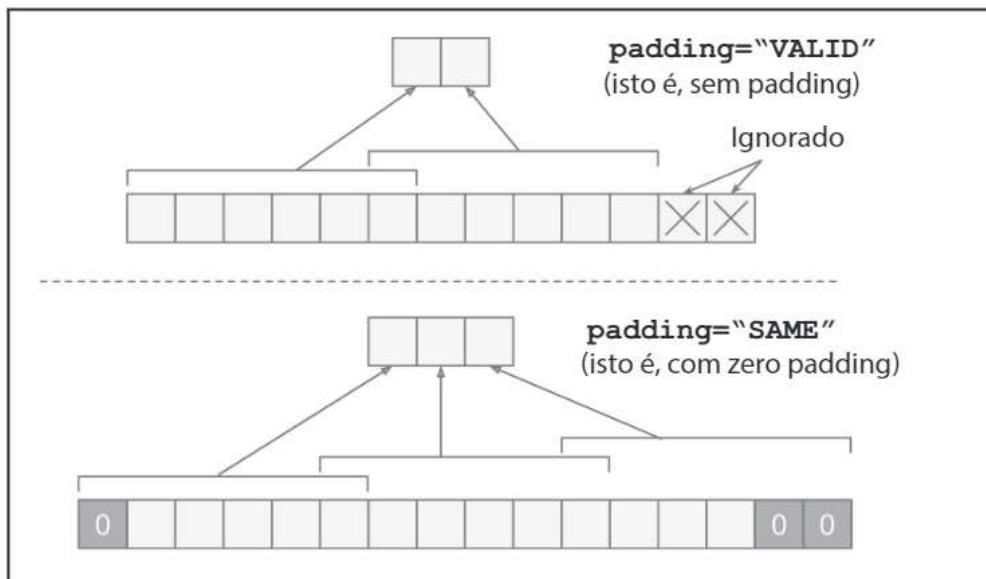
with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1], cmap="gray") # plota o mapa da segunda característica da primeira imagem
plt.show()
```

A maior parte deste código é autoexplicativa, mas a linha `tf.nn.conv2d()` merece uma pequena explicação:

- `X` é o minilote de entrada (um tensor 4D, como explicado anteriormente);
- `filters` é o conjunto de filtros a se aplicar (também um tensor 4D, como explicado anteriormente);
- `strides` é um array 1D de quatro elementos nos quais os dois elementos centrais são os strides vertical e horizontal ( $s_h$  e  $s_w$ ). O primeiro e o último elementos devem ser iguais a 1. Eles podem um dia ser utilizados para especificar um incremento em lote (para pular algumas instâncias) e um incremento do pipeline (para pular alguns dos mapas de características ou canais da camada anteriores);
- `padding` deve ser tanto “VALID” ou “SAME”:
  - Se configurada para “VALID”, a camada convolucional *não* utiliza o zero padding e pode ignorar algumas linhas e colunas na parte inferior e direita da imagem de entrada, dependendo do incremento, conforme mostrado na Figura 13-7 (para simplificar, apenas a dimensão horizontal é mostrada aqui, mas é claro que a mesma lógica se aplica à dimensão vertical);
  - Caso seja configurada para “SAME”, a camada convolucional utiliza o zero padding se necessário. Nesse caso, o número de neurônios de saída é igual ao número de neurônios de entrada dividido pelo stride, arredondado (neste

exemplo,  $\text{ceil}(13/5) = 3$ ). Os zeros são adicionados em torno das entradas o mais uniformemente possível.



*Figura 13-7. Opções de padding — largura da entrada: 13, largura do filtro: 6, stride: 5*

Criamos manualmente os filtros neste exemplo simples, mas em uma CNN real você permitiria que o algoritmo de treinamento descobrisse automaticamente os melhores filtros. O TensorFlow tem uma função `tf.layers.conv2d()` que cria a variável de filtros para você (denominado `kernel`) e a inicializa aleatoriamente, além de criar também a variável de polarização (denominada `bias`) e a inicializar com zeros. Por exemplo, o código a seguir cria um placeholder de entrada seguido por uma camada convolucional com dois mapas de características  $7 \times 7$  usando strides  $2 \times 2$  (note que esta função espera somente strides verticais e horizontais) e padding “SAME”:

```
X = tf.placeholder(shape=(None, height, width, channels), dtype=tf.float32)
conv = tf.layers.conv2d(X, filters=2, kernel_size=7, strides=[2,2],
                      padding="SAME")
```

Infelizmente, as camadas convolucionais têm muitos hiperparâmetros: você deve escolher o número de filtros, sua altura e largura, os incrementos e o tipo de padding. Como sempre, você pode utilizar a validação cruzada para encontrar os valores corretos do hiperparâmetro, mas isso consome muito tempo. Discutiremos as arquiteturas de CNN comuns posteriormente para dar uma ideia de quais valores dos hiperparâmetros funcionam melhor na prática.

## Requisitos de Memória

Outro problema com as CNNs é que as camadas convolucionais requerem uma quantidade enorme de RAM, especialmente durante o treinamento, porque o reverse pass da retropropagação requer todos os valores intermediários calculados durante o forward pass.

Por exemplo, considere uma camada convolucional com filtros  $5 \times 5$  produzindo 200 mapas de tamanho  $150 \times 100$ , com stride de 1 e padding SAME. Se a entrada for uma imagem RGB de  $150 \times 100$  (três canais), então o número de parâmetros será  $(5 \times 5 \times 3 + 1) \times 200 = 15.200$  (o +1 corresponde aos termos de polarização), o que é minúsculo se comparado com uma camada totalmente conectada.<sup>7</sup> No entanto, cada um dos 200 mapas contém  $150 \times 100$  neurônios e cada um desses neurônios precisa calcular uma soma ponderada de suas  $5 \times 5 \times 3 = 75$  entradas: um total de 225 milhões de multiplicações de floats. Não é tão ruim quanto uma camada totalmente conectada, mas ainda bastante intensa em termos computacionais. Além disso, se os mapas forem representados utilizando floats de 32 bits, a saída da camada convolucional ocupará  $200 \times 150 \times 100 \times 32 = 96$  milhões de bits (aproximadamente 11,4 MB) de RAM.<sup>8</sup> E isso somente para uma instância! Se um lote de treinamento contiver 100 instâncias, essa camada utilizará mais de 1 GB de RAM!

A RAM ocupada por uma camada pode ser liberada assim que a próxima camada tiver sido calculada durante a inferência (ou seja, ao fazer uma previsão para uma nova instância), portanto você somente precisará da quantidade máxima de RAM exigida por duas camadas consecutivas. Mas, durante o treinamento, tudo o que foi calculado ao longo do forward pass precisa ser preservado para o reverse pass, então a quantidade de RAM necessária é (pelo menos) a quantidade total de RAM requerida por todas as camadas.



Se o treinamento falhar devido a um erro de falta de memória, você pode tentar reduzir o tamanho do minilote. Alternativas: reduzir a dimensionalidade utilizando um stride ou removendo algumas camadas; tentar utilizar floats de 16 bits em vez de 32 bits; distribuir a CNN por vários dispositivos.

Agora veremos o segundo bloco comum de construção de CNNs: a *camada pooling*.

<sup>7</sup> Uma camada totalmente conectada com  $150 \times 100$  neurônios, cada um conectado a todas as entradas de  $150 \times 100 \times 3$ , teria  $150^2 \times 100^2 \times 3 = 675$  milhões de parâmetros!

<sup>8</sup> 1 MB = 1,024 kB =  $1,024 \times 1,024$  bytes =  $1,024 \times 1,024 \times 8$  bits.

## Camada Pooling

É muito fácil entender as camadas pooling depois que entendemos como as camadas convolucionais funcionam. Seu objetivo é *subamostrar* (ou seja, diminuir) a imagem de entrada para reduzir a carga computacional, o uso de memória e o número de parâmetros (limitando assim o risco de sobreajuste).

Assim como nas camadas convolucionais, cada neurônio na camada pooling é conectado às saídas de um número limitado de neurônios na camada anterior, localizados dentro de um pequeno campo receptivo retangular. Como antes, você deve definir seu tamanho, o stride e o tipo de preenchimento. No entanto, um neurônio de pooling não tem pesos; com a utilização de uma função de agregação, ele apenas une as entradas, como a max ou a mean. A Figura 13-8 mostra uma max pooling layer que é o tipo mais comum de camada pooling. Neste exemplo, utilizamos um pooling kernel  $2 \times 2$ , um stride de 2 e nenhum preenchimento. Note que apenas o valor máximo de entrada em cada kernel chega à próxima camada e as outras entradas são descartadas.

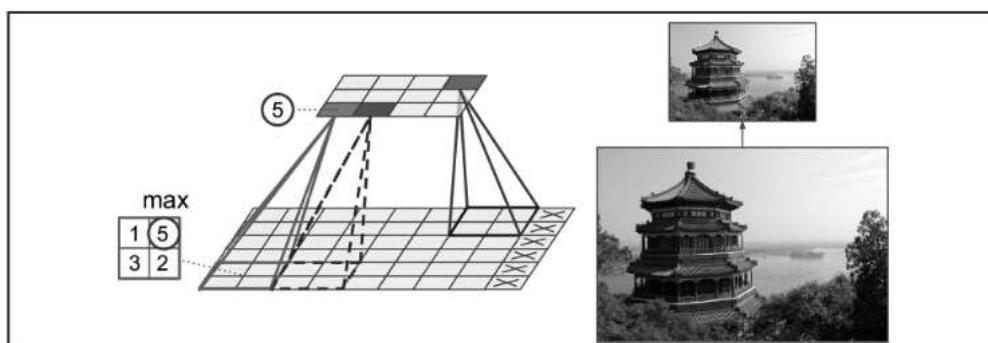


Figura 13-8. Camada max pooling (kernel pooling  $2 \times 2$ , stride de 2, sem padding)

Este é claramente um tipo muito destrutivo de camada: sua saída será duas vezes menor em ambas as direções (sua área será quatro vezes menor), descartando 75% dos valores de entrada, mesmo com um minúsculo kernel  $2 \times 2$  e um incremento de 2.

A camada de pooling funciona de forma independente em todos os canais de entrada, portanto, a profundidade de saída é a mesma de entrada. Você pode alternativamente agrupar a dimensão de profundidade, como veremos a seguir, caso em que as dimensões espaciais da imagem (altura e largura) permanecem inalteradas, mas o número de canais é reduzido.

É muito fácil a implementação de uma camada max pooling no TensorFlow. O código a seguir cria uma camada max pooling utilizando um kernel  $2 \times 2$ , stride de 2 e sem padding e, em seguida, a aplica a todas as imagens no conjunto de dados:

```
[...] # carregue o conjunto de dados das imagens, como acima

# Crie um grafo com uma entrada X mais uma camada max pooling
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1,2,2,1], strides=[1,2,2,1], padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})

plt.imshow(output[0].astype(np.uint8)) # plota a saída da primeira imagem
plt.show()
```

O argumento `ksize` contém o formato do kernel ao longo de todas as quatro dimensões do tensor de entrada: `[batch size, height, width, channels]`. O TensorFlow atualmente não suporta o agrupamento em múltiplas instâncias, então o primeiro elemento `ksize` deve ser igual a 1. Além disso, tanto nas dimensões espaciais (altura e largura) quanto na de profundidade, ele não suporta o agrupamento, portanto `ksize[1]` e `ksize[2]` devem ser iguais a 1, ou `ksize[3]` deve ser igual a 1.

Para criar uma *camada average pooling*, utilize a função `avg_pool()` em vez da `max_pool()`.

Agora você conhece todos os blocos de construção para criar uma rede neural convolucionial. Veremos como montá-las.

## Arquiteturas CNN

As arquiteturas CNN típicas empilham algumas camadas convolucionais (cada uma geralmente seguida por uma camada ReLU), depois uma camada pooling, depois outras camadas convolucionais (+ ReLU), depois outra camada pooling e assim por diante. A imagem fica cada vez menor à medida que avança pela rede, mas também fica mais e mais profunda (isto é, com mais mapas de características) graças às camadas convolucionais (consulte a Figura 13-9). No topo da pilha, uma rede neural feedforward regular é adicionada, composta de algumas camadas totalmente conectadas (+ReLUs) e a camada final gera a previsão (por exemplo, uma camada softmax que gera probabilidades estimadas de classe).

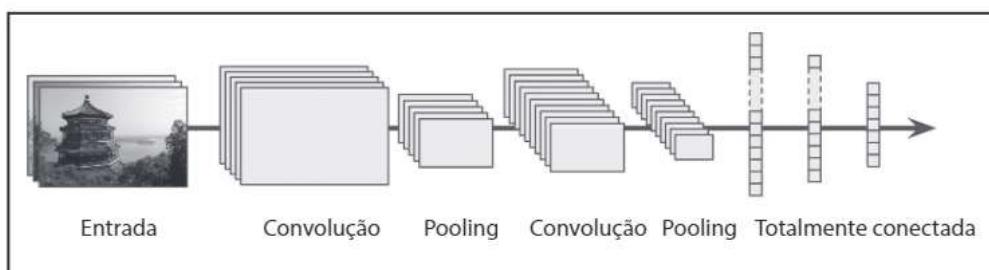


Figura 13-9. Arquitetura CNN típica



É um erro comum utilizar kernels de convolução muito grandes. Você pode obter o mesmo efeito de uma camada convolucional com um kernel  $9 \times 9$  ao empilhar duas camadas com kernels  $3 \times 3$  com muito menos cálculos e parâmetros.

Ao longo dos anos, foram desenvolvidas variantes dessa arquitetura fundamental, levando a avanços surpreendentes no campo. Uma boa medida desse progresso é a taxa de erro em competições, como o desafio ILSVRC ImageNet (<http://image-net.org/>), no qual as taxas de erro top-5 para a classificação de imagens caiu de 26% para menos de 3% em apenas seis anos. As cinco primeiras taxas de erros são o número de imagens de teste para as quais as cinco principais previsões do sistema não incluíram a resposta correta. As imagens são grandes (256 pixels de altura) e há 1 mil classes, algumas das quais realmente sutis (tente distinguir 120 raças de cães). Uma boa maneira de entender como as CNNs funcionam é observar a evolução das entradas vencedoras.

Observaremos primeiro a arquitetura clássica LeNet-5 (1998), depois veremos três dos vencedores do desafio ILSVRC: AlexNet (2012), GoogLeNet (2014) e ResNet (2015).

### Outras Tarefas Visuais

Houve um progresso impressionante também em outras tarefas visuais, como a detecção e localização de objetos e a segmentação de imagens. Na detecção e localização de objetos, a rede neural produz uma sequência de caixas delimitadoras em torno de vários objetos na imagem. Por exemplo, veja o artigo de 2015 de Maxine Oquab *et al.* (<https://goo.gl/ZKuDt>), que produz um mapa de calor para cada classe do objeto, ou o artigo de 2015 de Russell Stewart *et al.* (<https://goo.gl/upuHl2>), que utiliza uma combinação de uma CNN para detectar faces e uma rede neural recorrente para gerar uma sequência de caixas delimitadoras em torno delas. Na segmentação de imagem, a rede produz uma imagem (geralmente do mesmo tamanho que a entrada) em que cada pixel indica a classe do objeto a qual pertence o pixel correspondente de entrada. Por exemplo, confira o artigo de Evan Shelhamer *et al.* 2016 (<https://goo.gl/7ReZql>).

### LeNet-5

A arquitetura LeNet-5 é talvez a arquitetura CNN mais conhecida e, como mencionado anteriormente, foi criada por Yann LeCun em 1998 e amplamente utilizada para reconhecimento de dígitos manuscritos (MNIST). É composta pelas camadas mostradas na Tabela 13-1.

*Tabela 13-1. Arquitetura LeNet-5*

Camada	Tipo	Mapas	Tamanho	Tamanho do Kernel	Stride	Ativação
Out	Totalmente conectado	–	10	–	–	RBF
F6	Totalmente conectado	–	84	–	–	tanh
C5	Convolução	120	1×1	5×5	1	tanh
S4	Média de Pooling	16	5×5	2×2	2	tanh
C3	Convolução	16	10×10	5×5	1	tanh
S2	Média de Pooling	6	14×14	2×2	2	tanh
C1	Convolução	6	28×28	5×5	1	tanh
In	Entrada	1	32×32	–	–	–

Há alguns detalhes extras a serem observados:

- Imagens MNIST têm  $28 \times 28$  pixels, mas são preenchidas com zeros para chegar a  $32 \times 32$  pixels e normalizadas antes de serem fornecidas à rede. O restante da rede não utiliza nenhum preenchimento, razão pela qual o tamanho continua encolhendo à medida que a imagem avança por ela;
- As camadas comuns de agrupamento são um pouco mais complexas do que o normal: cada neurônio calcula a média de suas entradas, multiplica o resultado por um coeficiente de aprendizado (um por mapa), adiciona um termo de polarização de aprendizado (novamente, um por mapa) e, finalmente, aplica a função de ativação;
- A maioria dos neurônios nos mapas C3 está conectada aos neurônios em apenas três ou quatro mapas S2 (em vez de todos os seis mapas S2). Veja a tabela 1 no documento original para detalhes;
- A camada de saída é especial: em vez de calcular o produto escalar das entradas e do vetor de peso, cada neurônio produz o quadrado da distância Euclidiana entre seu vetor de entrada e seu vetor de peso. Cada saída mede o quanto a imagem pertence a uma determinada classe de dígitos. A função de custo de entropia cruzada é preferida, pois penaliza muito mais as previsões ruins, produzindo gradientes maiores e, portanto, convergindo mais rapidamente.

O site de Yann LeCun (<http://yann.lecun.com/>) (seção “LENET”) apresenta ótimas demonstrações de classificação dos dígitos da LeNet-5.

## AlexNet

A arquitetura CNN *AlexNet* (<http://goo.gl/mWRBRp>)<sup>9</sup> venceu o desafio 2012 ImageNet ILSVRC com vantagem: alcançou 17% de taxa de erro no top-5 enquanto o segundo melhor alcançou apenas 26%! Foi desenvolvida por Alex Krizhevsky (daí o nome), Ilya Sutskever e Geoffrey Hinton, e foi a primeira a empilhar camadas convolucionais diretamente umas sobre as outras em vez de empilhar uma camada pooling no topo de cada camada convolucional. Se assemelha muito ao LeNet-5, apenas é muito maior e mais profunda. A Tabela 13-2 apresenta essa arquitetura.

Tabela 13-2. Arquitetura AlexNet

Camada	Tipo	Mapas	Tamanho	Tamanho do Kernel	Incremento	Preenchimento	Ativação
Out	Totalmente conectado	—	1.000	—	—	—	Softmax
F9	Totalmente conectado	—	4.096	—	—	—	ReLU
F8	Totalmente conectado	—	4.096	—	—	—	ReLU
C7	Convolução	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolução	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolução	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	—
C3	Convolução	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	—
C1	Convolução	96	55 × 55	11 × 11	4	SAME	ReLU
In	Entrada	3 (RGB)	224 × 224	—	—	—	—

Os autores utilizaram duas técnicas de regularização que discutimos nos capítulos anteriores para a redução do sobreajuste. Para as saídas das camadas F8 e F9, primeiro eles aplicaram o dropout (com uma taxa de descarte de 50%) durante o treinamento e, em seguida, realizaram o aumento de dados através do deslocamento aleatório das imagens de treinamento por vários desvios, invertendo-as horizontalmente e alterando as condições de iluminação.

A AlexNet também utiliza uma etapa de normalização competitiva imediatamente após a etapa ReLU das camadas C1 e C3, chamada *normalização de resposta local* (LRN, em inglês). Essa forma de normalização faz com que os neurônios ativados mais fortemente inibam neurônios no mesmo local, mas em mapas de características vizinhos (tal ativação

<sup>9</sup> “ImageNet Classification with Deep Convolutional Neural Networks”, A. Krizhevsky *et al.* (2012).

competitiva foi observada em neurônios biológicos). Isso incentiva os diferentes mapas a se especializarem, separando-os e forçando-os a explorar uma gama mais ampla das características, melhorando a generalização. A Equação 13-2 mostra como aplicar a LRN.

*Equação 13-2. Normalização de resposta local*

$$b_i = a_i \left( k + \alpha \sum_{j=j_{\text{baixo}}}^{j_{\text{alto}}} a_j^2 \right)^{-\beta} \quad \text{com} \quad \begin{cases} j_{\text{alto}} = \min \left( i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{baixo}} = \max \left( 0, i - \frac{r}{2} \right) \end{cases}$$

- $b_i$  é a saída normalizada do neurônio localizado no mapa de características  $i$ , em alguma linha  $u$  e coluna  $v$  (observe que nesta equação consideramos apenas neurônios localizados nesta linha e coluna, então  $u$  e  $v$  não são exibidos);
- $a_i$  é a ativação daquele neurônio após o passo ReLU, mas antes da normalização;
- $k$ ,  $\alpha$ ,  $\beta$  e  $r$  são hiperparâmetros.  $k$  é chamado de *bias* e  $r$  é chamado de *depth radius*;
- $f_n$  é o número de mapas de características.

Por exemplo, se  $r = 2$  e um neurônio tiver uma forte ativação, ele inibirá a ativação dos neurônios localizados no mapa de características imediatamente acima e abaixo.

Na AlexNet, os hiperparâmetros são definidos da seguinte forma:  $r = 2$ ,  $\alpha = 0,00002$ ,  $\beta = 0,75$  e  $k = 1$ . Com a utilização da operação `tf.nn.local_response_normalization()` do TensorFlow, esta etapa pode ser implementada.

Uma variante do AlexNet chamada *ZF Net* foi desenvolvida por Matthew Zeiler e Rob Fergus e venceu o desafio ILSVRC 2013. É essencialmente a AlexNet com alguns hiperparâmetros ajustados (número de mapas de características, tamanho do kernel, stride, etc.).

## GoogLeNet

A arquitetura GoogLeNet (<http://goo.gl/tCFzVs>) foi desenvolvida por Christian Szegedy *et al.* da Google Research,<sup>10</sup> e venceu o desafio ILSVRC 2014 deixando a taxa de erro do top-5 abaixo de 7%. Esse ótimo desempenho se deu em grande parte pelo fato de que a rede era muito mais profunda que as CNNs anteriores (veja a Figura 13-11). Isso foi possível graças a subredes chamadas *módulos inception*,<sup>11</sup> que possibilitam que o GoogLeNet utilize parâmetros muito mais eficientemente do que as arquiteturas ante-

10 “Going Deeper with Convolutions”, C. Szegedy *et al.* (2015).

11 No filme *A Origem* (do inglês, Inception), de 2010, os personagens continuam se aprofundando em camadas múltiplas de sonhos, daí o nome desses módulos.

riores: o GoogLeNet na verdade tem 10 vezes menos parâmetros que a AlexNet (cerca de 6 milhões em vez de 60 milhões).

A Figura 13-10 mostra a arquitetura de um módulo de criação. A notação “ $3 \times 3 + 2(S)$ ” significa que a camada utiliza um kernel  $3 \times 3$ , stride de 2 e padding SAME. Primeiro o sinal de entrada é copiado e alimentado em quatro camadas diferentes. Todas as camadas convolucionais utilizam a função de ativação ReLU. Observe que o segundo conjunto de camadas convolucionais utilizam diferentes tamanhos de kernel ( $1 \times 1$ ,  $3 \times 3$  e  $5 \times 5$ ), permitindo que eles capturem padrões em diferentes escalas. Observe também que cada camada utiliza um stride de 1 e padding SAME (até mesmo a camada max pooling), portanto todas as suas saídas têm a mesma altura e largura de suas entradas. Isso possibilita a concatenação de todas as saídas ao longo da dimensão de profundidade na *camada final depth concat* (isto é, empilhar os mapas de todas as quatro camadas convolucionais superiores). Com a utilização da operação `tf.concat()` com `axis=3` (eixo 3 é a profundidade) essa camada de concatenação pode ser implementada no TensorFlow.

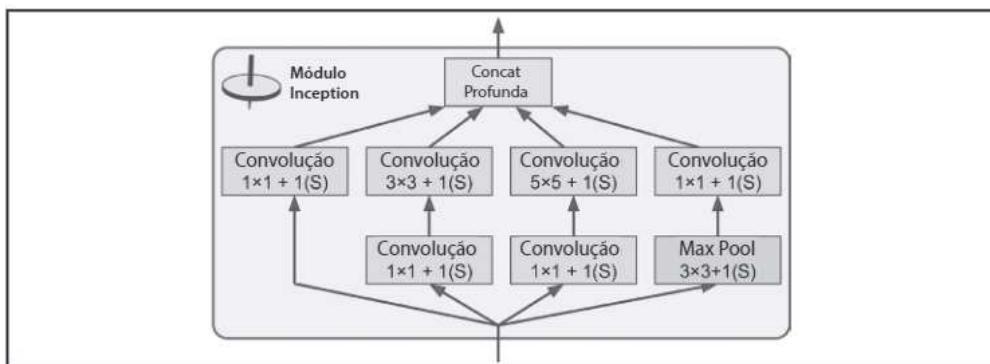


Figura 13-10. Módulo Inception

Você pode se perguntar por que os módulos de criação têm camadas convolucionais com núcleos  $1 \times 1$ . Já que olham apenas um pixel por vez, certamente essas camadas não podem capturar nenhuma característica. Na verdade, essas camadas servem dois propósitos:

- Primeiro, para que sirvam como *camadas de gargalo*, elas são configuradas para exibir muito menos mapas do que suas entradas, o que significa que reduzem a dimensionalidade. Isso é particularmente útil antes das convoluções  $3 \times 3$  e  $5 \times 5$ , já que essas camadas são muito dispendiosas em termos computacionais;
- Segundo, cada par de camadas convolucionais ( $[1 \times 1, 3 \times 3]$  e  $[1 \times 1, 5 \times 5]$ ) age como uma única e poderosa camada convolucional capaz de capturar padrões mais complexos. De fato, em vez de varrer um classificador linear simples pela imagem (como faz uma única camada convolucional), este par de camadas convolucionais varre uma rede neural de duas camadas pela imagem.

Em suma, você pode pensar em todo o módulo de criação como uma camada convolucional bombada, capaz de produzir mapas que capturem padrões complexos em várias escalas.



O número de kernels convolucionais para cada camada convolucionar é um hiperparâmetro. Infelizmente, isso significa que você terá mais seis hiperparâmetros para ajustar para cada camada de criação adicionada.

Agora, veremos a arquitetura CNN do GoogLeNet (veja a Figura 13-11). O GoogLeNet é tão profundo que tivemos que representá-lo em três colunas, mas, na verdade, ele é uma pilha alta que inclui nove módulos inception (as caixas com os peões). O número de mapas produzidos por cada camada convolucionar e cada camada pooling é mostrado antes do tamanho do kernel. O número de mapas gerados por cada camada convolucionar no módulo (na mesma ordem da Figura 13-10) representa os seis números nos módulos iniciais. Observe que todas as camadas convolucionais utilizam a função de ativação ReLU.

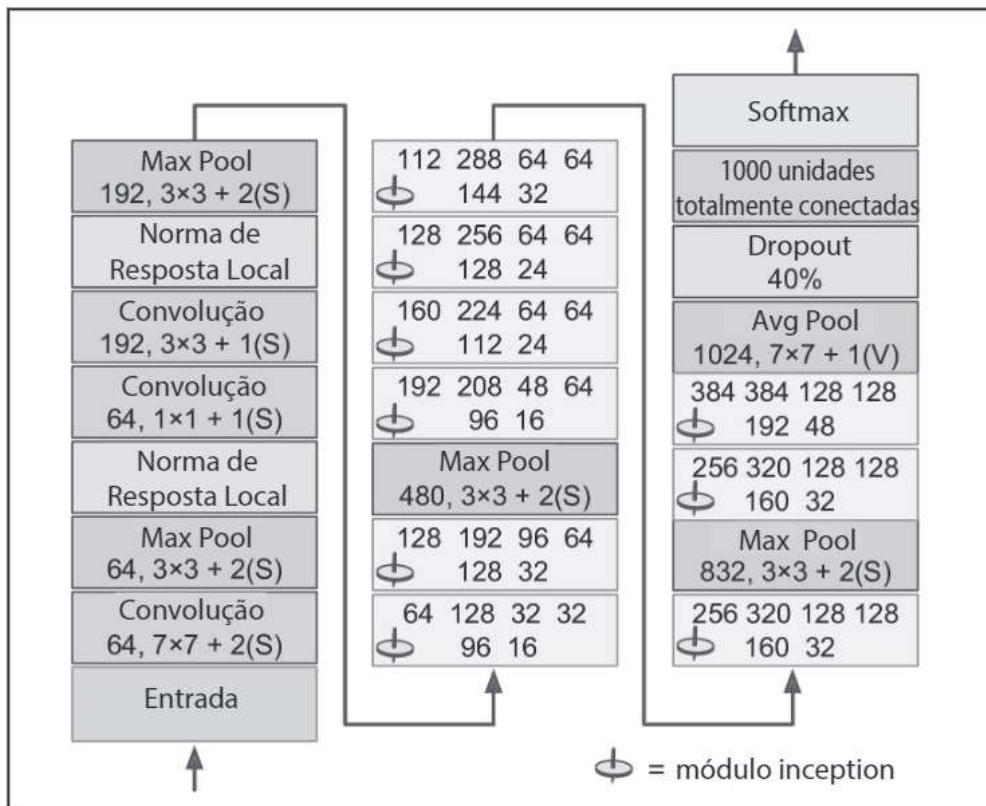


Figura 13-11. Arquitetura GoogLeNet

Analisaremos essa rede:

- As duas primeiras camadas dividem a altura e a largura da imagem por 4 (portanto sua área é dividida por 16) para reduzir a carga computacional;
- Em seguida, a camada de normalização de resposta local garante que as camadas anteriores aprendam uma ampla variedade de características (conforme discutido anteriormente);
- Seguem-se duas camadas convolucionais onde a primeira atua como uma camada de gargalo. Você pode pensar neste par como uma única camada convolucional mais inteligente, como explicado anteriormente;
- Novamente, uma camada de normalização de resposta local garante que as camadas anteriores capturem uma ampla variedade de padrões;
- Em seguida, novamente para acelerar os cálculos, uma camada max pooling reduz a altura e a largura da imagem em 2;
- Em seguida vem a pilha alta de nove módulos de criação, intercalados com um par de camadas max pooling para reduzir a dimensionalidade e acelerar a rede;
- Então, a camada média de pooling utiliza um kernel do tamanho dos mapas com padding VALID produzindo mapas  $1 \times 1$ : essa estratégia surpreendente é chamada de *global average pooling*. Ela efetivamente força as camadas anteriores a produzir mapas que são, na verdade, mapas de confiança para cada classe de destino (já que outros tipos de características seriam destruídos pela etapa média). Isso torna desnecessário ter várias camadas totalmente conectadas no topo da CNN (como na AlexNet), reduzindo consideravelmente o número de parâmetros na rede e limitando o risco de sobreajuste;
- As últimas camadas são autoexplicativas: dropout para regularização, depois uma camada totalmente conectada com uma função de ativação softmax para gerar probabilidades estimadas de classe.

Este diagrama é um pouco simplificado: a arquitetura GoogLeNet original também incluiu dois classificadores auxiliares conectados ao terceiro e ao sexto módulos inception. Ambos foram compostos por uma camada average pooling, uma camada convolucional, duas camadas totalmente conectadas e uma camada de ativação softmax. Sua perda (reduzida em 70%) foi adicionada à perda total durante o treinamento. O objetivo era combater o problema dos vanishing gradients e regularizar a rede. No entanto, foi demonstrado que o seu efeito foi relativamente pequeno.

## ResNet

Por último, mas não menos importante, o vencedor do desafio ILSVRC 2015, foi a *Residual Network* (<http://goo.gl/4puHU5>) (ou ResNet), desenvolvida por Kaiming He *et al*<sup>12</sup>, que apresentou uma surpreendente taxa de erro nos top-5 abaixo de 3,6%, usando uma CNN extremamente profunda composta por 152 camadas. A chave para poder treinar uma rede tão profunda é utilizar *conexões skip* (também chamadas de *shortcuts connections*): o sinal fornecido a uma camada também é adicionado à saída de uma camada localizada um pouco acima da pilha. Veremos por que isso é útil.

Ao treinar uma rede neural, o objetivo é fazer com que ela modele uma função alvo  $h(x)$ . Se você adicionar a entrada  $x$  à saída da rede (isto é, adicionar uma conexão skip), então a rede será forçada a modelar  $f(x) = h(x) - x$  em vez de  $h(x)$ , o que é chamado *residual learning* (veja a Figura 13-12).

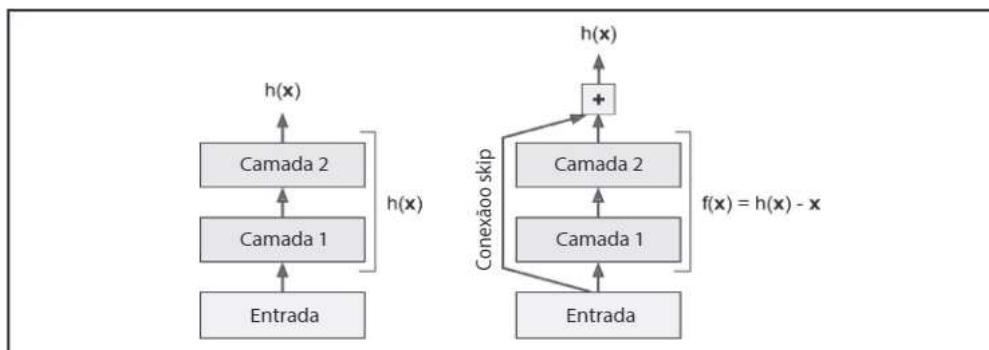


Figura 13-12. Residual Learning

Quando você inicializa uma rede neural regular, os seus pesos estão próximos de zero, então a rede apenas produz valores próximos de zero. Se você adicionar uma conexão skip, a rede resultante exibirá apenas uma cópia de suas entradas; em outras palavras, ela inicialmente modela a função de identidade, acelerando consideravelmente o treinamento se a função de destino estiver razoavelmente próxima da função identidade (o que geralmente é o caso).

Além disso, se você adicionar muitas conexões skip, a rede pode começar a fazer progresso mesmo se várias camadas ainda não começaram a aprender (veja a Figura 13-13). Graças às conexões skip, o sinal pode passar facilmente por toda a rede. A rede residual profunda pode ser vista como uma pilha de *unidades residuais*, sendo que cada uma delas é uma pequena rede neural com uma conexão skip.

12 “Deep Residual Learning for Image Recognition,” K. He (2015).

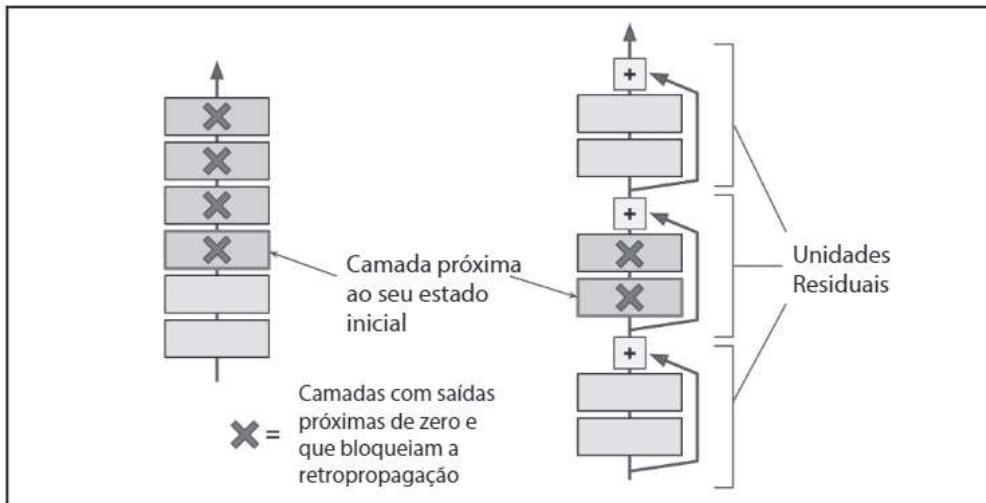


Figura 13-13. Rede neural profunda regular (esquerda) e rede residual profunda (direita)

Agora veremos a arquitetura ResNet (veja a Figura 13-14). Na verdade, ela é surpreendentemente simples. Ela começa e termina exatamente como o GoogLeNet (mas sem uma camada de dropout) e entre elas há apenas uma pilha muito profunda de unidades residuais simples. Cada unidade residual é composta por duas camadas convolucionais com Normalização por Lote (BN), ativação ReLU com a utilização de kernels  $3 \times 3$  e com a preservação de dimensões espaciais (stride de 1 e padding SAME )

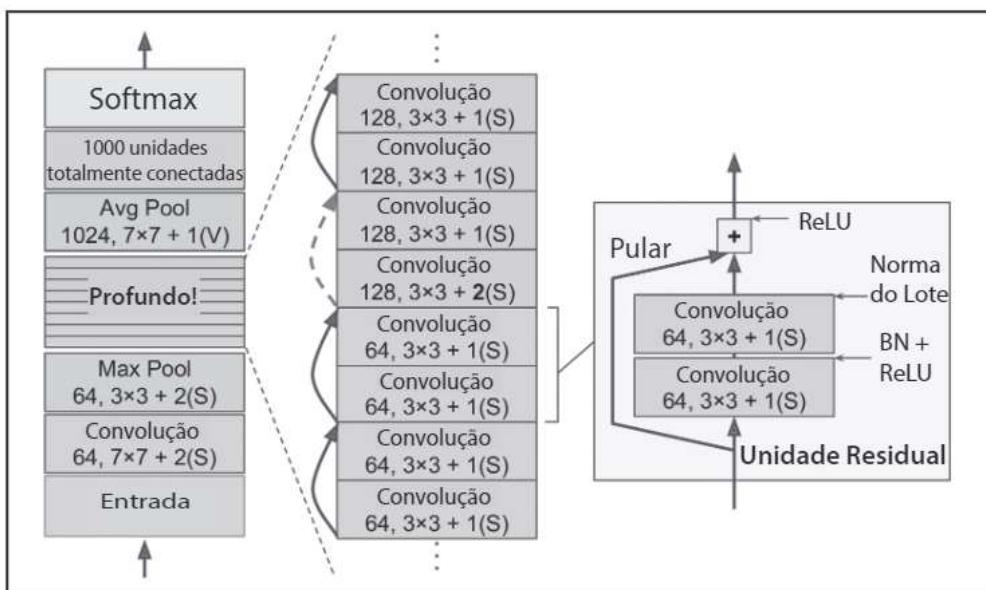


Figura 13-14. Arquitetura ResNet

Note que o número de mapas é dobrado em poucas unidades residuais, ao mesmo tempo em que sua altura e largura são reduzidas pela metade (utilizando uma camada convolucional com stride de 2). As entradas não podem ser adicionadas diretamente às saídas da unidade residual quando isso acontece, pois elas não têm o mesmo formato (por exemplo, esse problema afeta a conexão skip representada pela seta tracejada na Figura 13-14). Para resolver este problema, as entradas são passadas através de uma camada convolucional de  $1 \times 1$  com stride de 2 e o número certo de mapas de saída (veja a Figura 13-15).

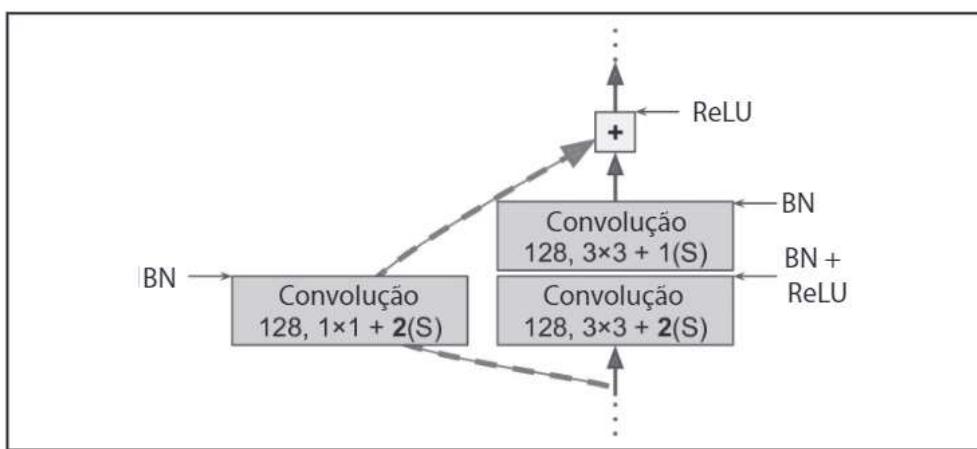


Figura 13-15. Conexão skip ao alterar o tamanho e a profundidade do mapa de características

A ResNet-34 é a ResNet com 34 camadas (contando apenas as camadas convolucionais e uma camada totalmente conectada) contendo três unidades residuais que geram 64 mapas, 4 RUs com 128 mapas, 6 RUs com 256 mapas e 3 RUs com 512 mapas.

ResNets mais profundas do que isso, como a ResNet-152, utilizam unidades residuais ligeiramente diferentes. Em vez de duas camadas convolucionais  $3 \times 3$  com (digamos) 256 mapas de características, elas utilizam três camadas convolucionais: primeiro uma camada convolucional  $1 \times 1$  com apenas 64 mapas de características (4 vezes menos) que atua como uma camada de gargalo (como já discutido), em seguida uma camada  $3 \times 3$  com 64 mapas e, finalmente, outra camada convolucional  $1 \times 1$  com 256 mapas (4 vezes 64) que restaura a profundidade original. A ResNet-152 contém três dessas RUs que geram 256 mapas, em seguida 8 RUs com 512 mapas, uma impressionante 36 RUs com 1.024 mapas e finalmente, 3 RUs com 2.048 mapas.

Como você pode ver, com vários tipos de arquiteturas surgindo a cada ano, a área está mudando rapidamente. Uma tendência clara é que as CNNs se aprofundem cada vez mais, além de também estarem ficando mais leves e exigindo menos parâmetros. No

momento, a arquitetura ResNet é a mais poderosa, e, sem dúvida, a mais simples, então é realmente a que você deve utilizar por enquanto, mas continue checando o desafio da ILSVRC a cada ano. Com uma impressionante taxa de erro de 2,99% a vencedora de 2016 foi a equipe Trimps-Soushen da China. Para tanto, eles treinaram combinações dos modelos anteriores e os uniram em um ensemble. A taxa de erro reduzida pode, ou não, dependendo da tarefa, valer a complexidade extra.

Verifique outras arquiteturas, em particular a VGGNet (<http://goo.gl/QcMjXQ>)<sup>13</sup> (vice-campeã do desafio ILSVRC 2014) e a Inception-v4 (<http://goo.gl/Ak2vBp>)<sup>14</sup> (que mescla as ideias da GoogLeNet e da ResNet e alcança uma taxa de erro de 3% no top-5 na classificação do ImageNet).



Não há realmente nada de especial na implementação das várias arquiteturas CNN que acabamos de discutir. Vimos anteriormente como construir todos os blocos de construção individuais, então você só precisa montá-los para criar a arquitetura desejada. Construiremos uma CNN completa nos próximos exercícios e você encontrará um código de trabalho completo nos notebooks da Jupyter

## Operações de Convolução do TensorFlow

O TensorFlow também oferece alguns outros tipos de camadas convolucionais:

- `tf.layers.conv1d()` cria uma camada convolucional para entradas 1D, o que é útil, por exemplo, em uma sentença que pode ser representada como um array 1D de palavras e o campo receptivo abrange algumas palavras vizinhas no processamento de linguagem natural;
- `tf.layers.conv3d()` cria uma camada convolucional para entradas 3D, como o scan 3D PET;
- `tf.nn.atrous_conv2d()` cria a *atrous convolutional layer* (“à trous” significa “com buracos” em Francês). Inserir linhas e colunas de zeros (isto é, buracos) equivale a utilizar uma camada convolucional regular com um filtro dilatado. Por exemplo, o filtro  $1 \times 3$  igual a  $[[1, 2, 3]]$  pode ser dilatado com uma *tarda de dilatação* de 4, resultando em um *filtro dilatado*  $[[1, 0, 0, 0, 2, 0, 0, 0, 3]]$ , permitindo que a camada convolucional tenha um campo receptivo maior sem nenhum preço computacional e sem utilizar parâmetros extras;

<sup>13</sup> “Very Deep Convolutional Networks for Large-Scale Image Recognition”, K. Simonyan and A. Zisserman (2015)

<sup>14</sup> “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”, C. Szegedy *et al.* (2016)

- `tf.layers.conv2d_transpose()` cria uma *camada convolucional de transposição*, às vezes chamada de *camada desconvolucional*,<sup>15</sup> que aumenta a amostragem (*upsample*) de uma imagem inserindo zeros entre as entradas, então você pode pensar nisso como uma camada convolucional comum usando um stride fracionário. A upsampling tem sua utilidade, por exemplo, na segmentação da imagem: em uma CNN típica, os mapas ficam cada vez menores à medida em que você avança na rede. Então, para produzir uma imagem do mesmo tamanho de sua entrada, você precisará de uma camada de upsampling;
- `tf.nn.depthwise_conv2d()` cria uma *camada convolucional em profundidade* que aplica cada filtro para cada canal de entrada individual de forma independente. Assim, se houver filtros  $f_n$  e canais de entrada  $f_{n'}$  ela produzirá  $f_n \times f_{n'}$  mapas;
- `tf.layers.separable_conv2d()` cria uma *camada convolucional separável* que primeiro age como uma camada convolucional profunda, então aplica uma camada convolucional de  $1 \times 1$  aos mapas resultantes, possibilitando a aplicação de filtros a conjuntos arbitrários de canais de entrada.

## Exercícios

1. Quais são as vantagens de uma CNN sobre uma DNN totalmente conectada para a classificação de imagens?
2. Considere uma CNN composta por três camadas convolucionais, cada uma com kernels  $3 \times 3$ , um stride de 2 e padding SAME. A camada inferior gera 100 mapas, a do meio gera 200 e a de cima 400. As imagens de entrada são RGB de  $200 \times 300$  pixels. Qual é o número total de parâmetros na CNN? Se estivermos utilizando floats de 32 bits, quanto de RAM essa rede exigirá ao fazer uma previsão para uma única instância? E quando treinamos em um minilote de 50 imagens?
3. Se sua GPU ficar sem memória durante o treinamento de uma CNN, quais são cinco coisas que você poderia tentar para resolver o problema?
4. Por que você adicionaria uma camada max pooling em vez de uma camada convolucional com o mesmo stride?
5. Quando você adicionaria uma camada de *normalização de resposta local*?
6. Você pode citar as principais inovações na AlexNet em comparação com a LeNet-5? E as principais inovações da GoogLeNet e da ResNet?
7. Construa sua própria CNN e tente atingir a maior precisão possível no MNIST.

<sup>15</sup> Este nome é bastante enganador, já que esta camada *não* realiza uma desconvolução, que é uma operação matemática bem definida (o inverso de uma convolução).

8. Classificando imagens grandes utilizando o Inception v3.
  - a. Baixe algumas imagens de vários animais. Carregue-as no Python, por exemplo, utilizando a função `matplotlib.image.imread()` ou a função `scipy.misc.imread()`. Redimensione e/ou corte-as em  $299 \times 299$  pixels e garanta que elas tenham apenas três canais (RGB), sem o de transparência. As imagens que foram treinadas no modelo Inception foram pré-processadas para que seus valores variem de -1,0 a 1,0, então você deve garantir isso também para as suas.
  - b. Faça o download do modelo mais recente do Inception v3: o ponto de verificação está disponível em <https://goo.gl/25uDF7>. A lista dos nomes das classes está disponível em <https://goo.gl/brXRtZ>, mas você deve inserir uma classe "background" no início.
  - c. Crie o modelo Inception v3 chamando a função `inception_v3()`, como mostrado abaixo. Isto deve ser feito dentro de um escopo do argumento criado pela função `inception_v3_arg_scope()`. Você também deve configurar `is_training=False` e `num_classes=1001` da seguinte forma:

```
from tensorflow.contrib.slim.nets import inception
import tensorflow.contrib.slim as slim

X = tf.placeholder(tf.float32, shape=[None, 299, 299, 3], name="X")
with slim.arg_scope(inception.inception_v3_arg_scope()):
    logits, end_points = inception.inception_v3(
        X, num_classes=1001, is_training=False)
predictions = end_points["Predictions"]
saver = tf.train.Saver()
```
  - d. Abra uma sessão e utilize o `Saver` para restaurar o ponto de verificação do modelo pré-treinado que você baixou anteriormente.
  - e. Execute o modelo para classificar as imagens que você preparou. Exiba as cinco principais previsões para cada imagem junto com a probabilidade estimada. Qual é a acurácia do modelo?
9. Transferência de aprendizado para classificação de imagens grandes.
  - a. Crie um conjunto de treinamento contendo pelo menos 100 imagens por classe. Por exemplo, você pode classificar suas próprias imagens com base no local (praia, montanha, cidade etc.) ou, como alternativa, utilizar apenas um conjunto de dados existente como o conjunto de dados das flores (<https://goo.gl/EgJVXZ>) ou o dos locais do MIT (<http://places.csail.mit.edu/>) (requer registro e é enorme).
  - b. Escreva uma etapa de pré-processamento que redimensionará e recortará a imagem em  $299 \times 299$  com alguma aleatoriedade para o aumento de dados.

- c. Utilizando o modelo Inception v3 pré-treinado do exercício anterior, congele todas as camadas até a camada de gargalo (ou seja, a última camada antes da camada de saída) e substitua a camada de saída pelo número apropriado de saídas para sua nova tarefa de classificação (por exemplo, o conjunto de dados das flores tem cinco classes mutuamente exclusivas, então a camada de saída deve ter cinco neurônios e utilizar a função de ativação softmax).
  - d. Divilde seu conjunto de dados em um conjunto de treinamento e um conjunto de testes. Treine o modelo no conjunto de treinamento e o avalie no conjunto de testes.
10. Acesse o tutorial DeepDream do TensorFlow (<https://goo.gl/4b2s6g>). É uma forma divertida de se familiarizar com várias formas de visualizar os padrões aprendidos por uma CNN e de gerar arte com a utilização do Aprendizado Profundo.

Soluções para estes exercícios estão disponíveis no Apêndice A.



## Capítulo 14

# Redes Neurais Recorrentes (RNN)

O taco rebate a bola. Imediatamente, você começa a correr antecipando sua trajetória e, ao segui-la, adapta seus movimentos para, finalmente, capturar a bola (sob uma chuva de aplausos). Prever o futuro é o que você faz o tempo todo, seja terminando a frase de um amigo ou antecipando o cheiro do café da manhã. Neste capítulo, discutiremos redes neurais recorrentes (RNN, do inglês), uma classe de redes que pode prever o futuro (bem, até certo ponto, é claro). Elas podem analisar *dados de séries temporais*, como preços de ações, e informar quando comprar ou vender, podem também antecipar trajetórias de carros e ajudar na prevenção de acidentes em sistemas autônomos de direção. Mais geralmente, em vez de em entradas de tamanho fixo, elas podem trabalhar em *sequências* de comprimentos arbitrários, como todas as redes que discutimos até agora. Por exemplo, podem utilizar frases, documentos ou amostras de áudio como entrada, tornando-as extremamente úteis para sistemas de processamento de linguagem natural (PNL) como a tradução automática, voz para texto ou *análise comportamental* (por exemplo, lendo resenhas de filmes e extraíndo a impressão do avaliador quanto ao filme).

Além disso, a capacidade de antecipação das RNNs também as torna capazes de uma criatividade surpreendente. Você pode pedir para prever quais serão as próximas notas mais prováveis de uma melodia, escolher depois aleatoriamente uma dessas notas e tocá-la. Em seguida, peça à rede para reproduzir as próximas notas mais prováveis e repita várias vezes o processo. Antes que você perceba, sua rede comporá uma melodia, como a que foi produzida pelo projeto Magenta do Google (<http://goo.gl/IxIL1V>) (<https://magenta.tensorflow.org/>). Da mesma forma, as RNNs podem gerar sentenças (<http://goo.gl/onkPNd>), legendas de imagens (<http://goo.gl/Nwx7Kh>) e muito mais. O resultado não é ainda exatamente um Shakespeare ou Mozart, mas quem sabe o que produzirão daqui a alguns anos?

Neste capítulo, examinaremos os conceitos fundamentais subjacentes às RNNs, o principal problema que elas enfrentam (a saber, gradientes vanishing/exploding discutidos no Capítulo 11) e as soluções amplamente utilizadas para combatê-lo: células LSTM e

GRU. Ao longo do caminho, como sempre utilizando o TensorFlow, mostraremos como implementar RNNs. Finalmente, daremos uma olhada na arquitetura de um sistema de tradução automática.

## Neurônios Recorrentes

Até agora, vimos principalmente redes neurais feedforward nas quais as ativações fluem apenas em uma direção, da camada de entrada à camada de saída (exceto algumas redes no Apêndice E). Uma rede neural recorrente é muito parecida com uma rede neural feedforward, exceto por também conter conexões apontando para trás. Vejamos a RNN mais simples possível, composta de apenas um neurônio recebendo entradas, produzindo uma saída e enviando essa saída de volta para si mesmo, conforme mostrado na Figura 14-1 (esquerda). A cada *intervalo de tempo t* (também chamado de *frame*) este *neurônio recorrente* recebe as entradas  $x_{(t)}$ , bem como sua própria saída do intervalo de tempo anterior,  $y_{(t-1)}$ . Podemos representar essa minúscula rede em relação ao eixo do tempo, conforme mostrado na Figura 14-1 (direita). Isto é chamado *desenrolando a rede ao longo do tempo*.

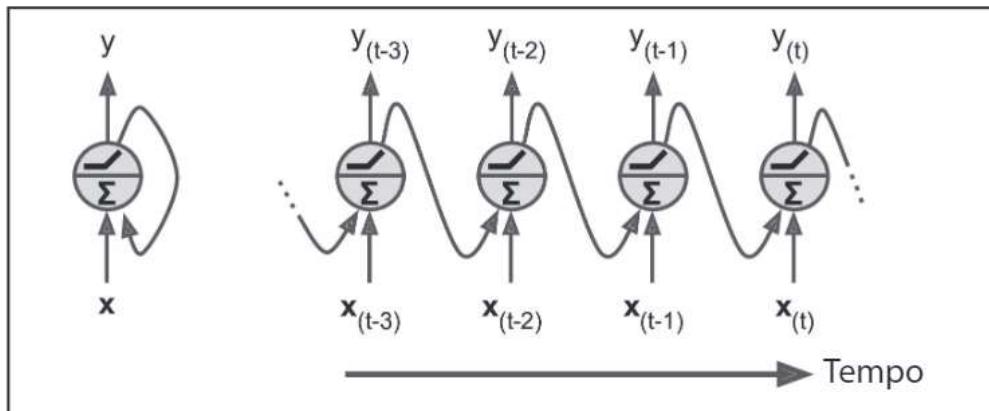


Figura 14-1. Um neurônio recorrente (esquerda), desenrolado através do tempo (direita)

Você pode criar facilmente uma camada de neurônios recorrentes. A cada etapa de tempo  $t$ , cada neurônio recebe tanto o vetor de entrada  $x_{(t)}$  quanto o vetor de saída da etapa de tempo anterior  $y_{(t-1)}$ , como mostrado na Figura 14-2. Note que, agora, tanto as entradas quanto as saídas são vetores (quando havia apenas um único neurônio a saída era um escalar).

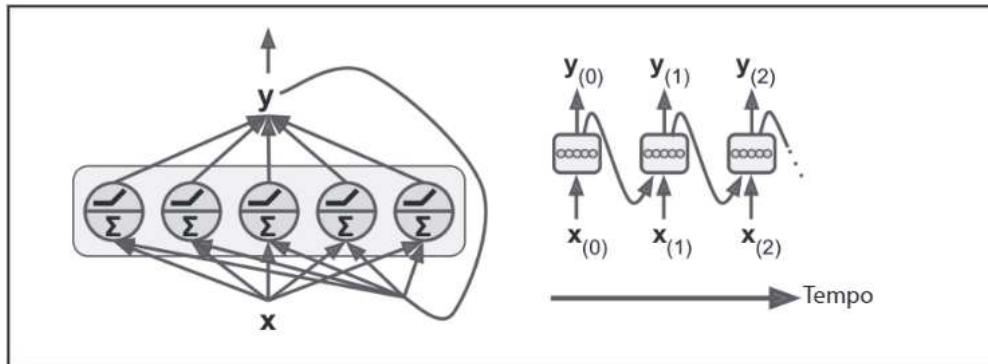


Figura 14-2. Uma camada de neurônios recorrentes (esquerda) desenrolados através do tempo (direita)

Cada neurônio recorrente tem dois conjuntos de pesos: um para as entradas  $x_{(t)}$  e outro para as saídas do intervalo de tempo anterior,  $y_{(t-1)}$ , chamaremos estes vetores de peso de  $w_x$  e  $w_y$ . Se considerarmos toda a camada recorrente em vez de apenas um neurônio recorrente, podemos colocar todos os vetores de peso em duas matrizes de peso,  $W_x$  e  $W_y$ . Como você poderia esperar, o vetor de saída de toda a camada recorrente pode, então, ser calculado ( $b$  é o vetor de polarização e  $\phi(\cdot)$  é a função de ativação, por exemplo, ReLU<sup>1</sup>) como mostrado na Equação 14-1.

*Equação 14-1. Saída de uma camada recorrente para uma única instância*

$$y_{(t)} = \phi(W_x^T \cdot x_{(t)} + W_y^T \cdot y_{(t-1)} + b)$$

Assim como para redes neurais feedforward, podemos calcular a saída de uma camada recorrente em uma só tomada para um minilote inteiro colocando todas as entradas no intervalo de tempo em uma matriz de entrada  $X_{(t)}$  (veja a Equação 14-2).

*Equação 14-2. Saídas de uma camada de neurônios recorrentes para todas as instâncias em um minilote*

$$\begin{aligned} Y_{(t)} &= \phi(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b) \\ &= \phi([X_{(t)} \quad Y_{(t-1)}] \cdot W + b) \text{ com } W = \begin{bmatrix} W_x \\ W_y \end{bmatrix} \end{aligned}$$

1 Observe que muitos pesquisadores preferem a utilização da função de ativação tangente hiperbólica (tanh) em RNNs em vez da função de ativação ReLU. Por exemplo, dê uma olhada no artigo de Vu Pham *et al.* "Dropout Improves Recurrent Neural Networks for Handwriting Recognition" (<https://goo.gl/2WSnaj>). No entanto, RNNs baseados em ReLU também conseguem, conforme mostrado no artigo de Quoc V. Le *et al* "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units" (<https://goo.gl/NrKAP0>).

- $\mathbf{Y}_{(t)}$  é uma matriz  $m \times n_{\text{neurônios}}$  que contém as saídas das camadas em um intervalo de tempo  $t$  para cada instância no minilote ( $m$  é o número de instâncias no minilote e  $n_{\text{neurônios}}$  é o número de neurônios);
- $\mathbf{X}_{(t)}$  é uma matriz  $m \times n_{\text{entradas}}$  que contém as entradas para todas as instâncias ( $n_{\text{entradas}}$  é o número de características de entrada);
- $\mathbf{W}_x$  é uma matriz  $n_{\text{entradas}} \times n_{\text{neurônios}}$  que contém os pesos de conexão para as entradas do intervalo de tempo atual;
- $\mathbf{W}_y$  é uma matriz  $n_{\text{neurônios}} \times n_{\text{neurônios}}$  que contém os pesos de conexão para as saídas do intervalo de tempo anterior;
- $\mathbf{b}$  é um vetor de tamanho  $n_{\text{neurônios}}$  que contém cada termo de polarização do neurônio;
- As matrizes de peso  $\mathbf{W}_x$  e  $\mathbf{W}_y$  são frequentemente concatenadas verticalmente em uma única matriz de peso  $\mathbf{W}$  no formato  $(n_{\text{entradas}} + n_{\text{neurônios}}) \times n_{\text{neurônios}}$  (veja a segunda linha da Equação 14-2);
- A notação  $[\mathbf{X}_{(t)} \ \mathbf{Y}_{(t-1)}]$  representa a concatenação horizontal das matrizes  $\mathbf{X}_{(t)}$  e  $\mathbf{Y}_{(t-1)}$ .

Observe que  $\mathbf{Y}_{(t)}$  é uma função de  $\mathbf{X}_{(t)}$  e  $\mathbf{Y}_{(t-1)}$ , que é uma função de  $\mathbf{X}_{(t-1)}$  e  $\mathbf{Y}_{(t-2)}$ , que é uma função de  $\mathbf{X}_{(t-2)}$  e  $\mathbf{Y}_{(t-3)}$ , e assim por diante, tornando  $\mathbf{Y}_{(t)}$  uma função de todas as entradas, pois o intervalo é  $t = 0$  (ou seja,  $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \dots, \mathbf{X}_{(t)}$ ). No primeiro intervalo de tempo,  $t = 0$ , não existem saídas anteriores, então elas tipicamente assumem como sendo zeros.

## Células de Memória

Como a saída de um neurônio recorrente no intervalo  $t$  é uma função de todas as entradas dos intervalos de tempos anteriores, você poderia dizer que ele tem uma aparência de *memória*. Uma parte de uma rede neural que preserva algum estado nos intervalos de tempo é chamada de *célula de memória* (ou simplesmente *célula*). Um único neurônio recorrente, ou uma camada de neurônios recorrentes, é uma *célula muito básica*, porém mais adiante neste capítulo veremos alguns tipos de células mais complexas e poderosas.

No geral, o estado de uma célula em um intervalo de tempo  $t$ , denominado  $\mathbf{h}_{(t)}$  (o “ $h$ ” é de “*hidden*” [escondido]), é uma função de algumas entradas naquele intervalo de tempo e seu estado no intervalo de tempo anterior:  $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$ . Sua saída no intervalo de tempo  $t$ , denominada  $y_{(t)}$ , também é uma função do estado anterior e das entradas atuais. No caso das células básicas, que discutimos até agora, a saída é igual ao estado, mas em células mais complexas este nem sempre é o caso, como mostrado na Figura 14-3.

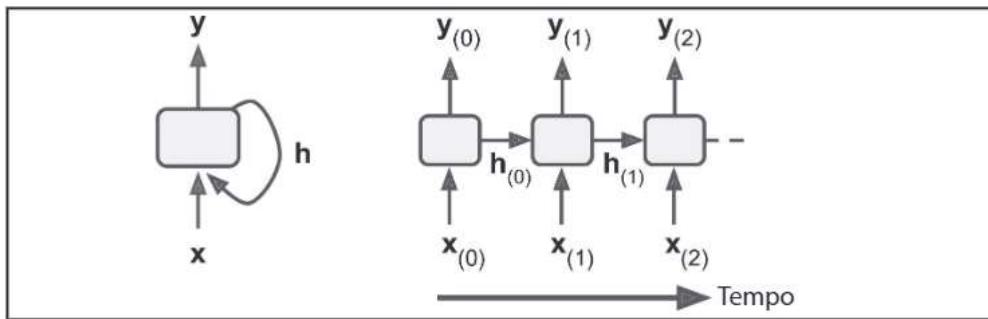


Figura 14-3. O estado oculto de uma célula e sua saída podem ser diferentes

## Sequências de Entrada e Saída

Uma RNN pode simultaneamente pegar uma sequência de entradas e produzir uma sequência de saídas (veja a Figura 14-4, rede acima à esquerda). Por exemplo, esse tipo de rede é útil para prever séries temporais como cotações de ações: você fornece os preços dos últimos  $N$  dias e ela exibe os preços deslocados em um dia no futuro (ou seja, de  $N - 1$  dias antes para amanhã).

Como alternativa, você poderia alimentar a rede com uma sequência de entradas e ignorar todas as saídas, exceto a última (veja a rede superior direita). Em outras palavras, esta é uma rede sequência-para-vetor. Por exemplo, você poderia alimentá-la com uma sequência de palavras correspondente a uma resenha de filme e a rede produziria uma pontuação de sentimento (por exemplo,  $-1$  [ódio] e  $+1$  [amor])

Por outro lado, você poderia alimentar a rede com uma única entrada na primeira vez (e zeros para todos os outros intervalos de tempo) e deixar que ela produza uma saída (veja a rede inferior esquerda). Esta é uma rede vetor-para-sequência. Por exemplo, a entrada poderia ser uma imagem e a saída poderia ser uma legenda para essa imagem.

Por fim, você poderia ter uma rede sequência-a-vetor chamada *codificador*, seguida de uma rede vetor-a-sequência, chamada *decodificador* (veja a rede inferior direita). Por exemplo, isso pode ser utilizado para traduzir uma frase de um idioma para outro. Você alimentaria a rede com uma frase em um idioma, o codificador converteria essa frase em uma única representação vetorial e, em seguida, o decodificador decodificaria esse vetor em uma sentença em outro idioma. Este modelo de duas etapas, chamado de Codificador-Decodificador, funciona muito melhor do que tentar traduzir rapidamente com uma única RNN sequência-a-sequência (como o representado no canto superior esquerdo), pois as últimas palavras de uma frase podem afetar as primeiras palavras da tradução e você precisa esperar até ter ouvido toda a frase antes de traduzi-la.

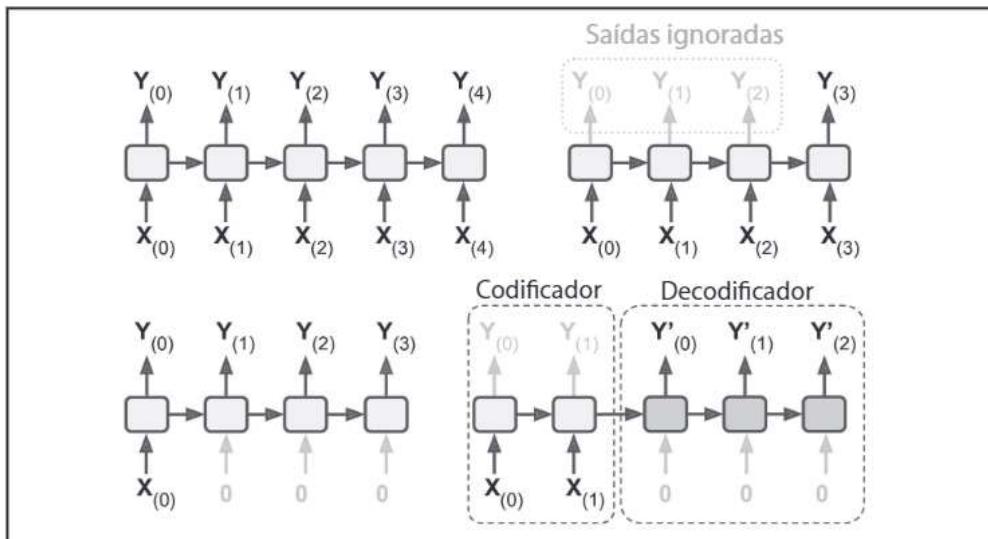


Figura 14-4. Seq para seq (canto superior esquerdo), seq para vetor (canto superior direito), vetor para seq (canto inferior esquerdo), delayed seq para seq (canto inferior direito)

Parece promissor, então começaremos a programar!

## RNNs Básicas no TensorFlow

Primeiro, sem utilizar nenhuma das operações RNN do TensorFlow, implementaremos um modelo RNN muito simples para entender melhor o que acontece nos bastidores. Criaremos uma RNN composta por uma camada com cinco neurônios recorrentes (como a RNN representada na Figura 14-2) usando a função de ativação tanh. Assumiremos que a RNN roda somente em dois intervalos de tempo, tomando vetores de entrada de tamanho 3 em cada intervalo. O código a seguir cria esta RNN desenrolada por meio de dois intervalos de tempo:

```

n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons,n_neurons],dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

```

```
init = tf.global_variables_initializer()
```

Essa rede se parece muito com uma rede neural feedforward de duas camadas com algumas modificações: primeiro, os mesmos pesos e termos de polarização são compartilhados por ambas as camadas e, segundo, alimentamos entradas em cada camada e obtemos saídas de cada camada. Para executar o modelo, precisamos alimentá-lo nas entradas em ambos os intervalos de tempo, assim:

```
import numpy as np

# Minilote:    instância 0,instância 1,instância 2,instância 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t = 0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t = 1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1], feed_dict={X0: X0_batch, X1: X1_batch})
```

Este minilote contém quatro instâncias, cada uma com uma sequência de entrada composta de exatamente duas entradas. Ao final, `Y0_val` e `Y1_val` conterá as saídas da rede em ambos os intervalos para todos os neurônios e todas as instâncias no minilote:

```
>>> print(Y0_val) # saída em t = 0
[[ -0.0664006   0.96257669   0.68105787   0.70918542  -0.89821595] # instância0
 [ 0.9977755  -0.71978885  -0.99657625   0.9673925  -0.99989718] # instância1
 [ 0.99999774  -0.99898815  -0.99999893   0.99677622  -0.99999988] # instância2
 [ 1.          -1.          -1.          -0.99818915   0.99950868]] # instância3
>>> print(Y1_val) # saída em t = 1
[[ 1.          -1.          -1.          0.40200216  -1.          ] # instância0
 [-0.12210433  0.62805319   0.96718419  -0.99371207  -0.25839335] # instância1
 [ 0.99999827  -0.9999994  -0.9999975  -0.85943311  -0.9999879 ] # instância2
 [ 0.99928284  -0.99999815  -0.99990582   0.98579615  -0.92205751]] # instância3
```

Não foi muito difícil, mas é claro que o grafo ficará muito grande se executarmos uma RNN com mais de 100 intervalos de tempo. Agora, utilizando as operações RNN do TensorFlow, veremos como criar o mesmo modelo.

## Desenrolamento Estático Através do Tempo

A função `static_rnn()` cria uma rede RNN desenrolada encadeando células. O código a seguir cria exatamente o mesmo modelo que o anterior:

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
                                                dtype=tf.float32)
Y0, Y1 = output_seqs
```

Como antes, primeiro criamos os placeholders de entrada. Em seguida, criamos uma `BasicRNNCell`, que você pode imaginar como sendo uma fábrica que cria cópias da célula para construir a RNN desenrolada (uma para cada intervalo de tempo). Em seguida, chamamos a `static_rnn()`, dando-lhe a fábrica de células e os tensores de entrada e informando o tipo de dados das entradas (isso é usado para criar a matriz de estado inicial, que por padrão é cheia de zeros). A função `static_rnn()` chama a função `call()` da fábrica de células uma vez por entrada e as encadeia como fizemos anteriormente, criando duas cópias da célula (cada uma contendo uma camada de cinco neurônios recorrentes) com pesos e termos de polarização compartilhados. A função `static_rnn()` retorna dois objetos. O primeiro é uma lista do Python que contém os tensores de saída para cada intervalo de tempo e o segundo é um tensor que contém os estados finais da rede. Quando você utiliza células básicas, o estado final é simplesmente igual ao último resultado.

Se houvesse 50 intervalos de tempo, não seria muito conveniente definir 50 placeholders e 50 tensores de saída. Além disso, você teria que alimentar cada um dos 50 placeholders e manipular as 50 saídas em tempo de execução. Vamos simplificar isso. O código a seguir cria a mesma RNN novamente, mas, dessa vez, é necessário um único placeholder para entrada no formato `[None, n_steps, n_inputs]` sendo que a primeira dimensão é o tamanho do minilote. Em seguida, extraí a lista de sequências de entrada para cada intervalo de tempo. `X_seqs` é uma lista Python de tensores `n_steps` no formato `[None, n_inputs]` e, mais uma vez, a primeira dimensão é do tamanho do minilote. Para fazer isto, primeiro trocamos as duas primeiras dimensões usando a função `transpose()` para que os intervalos de tempo sejam agora a primeira dimensão. Em seguida, extraímos uma lista de tensores Python na primeira dimensão (ou seja, um tensor por intervalo de tempo) usando a função `unstack()`. As próximas duas linhas são as mesmas de antes. Finalmente, mesclamos todos os tensores de saída usando a função `stack()` e trocamos as primeiras duas dimensões para obter um tensor `outputs` final no formato `[None, n_steps, n_neurons]` (de novo, a primeira dimensão é do tamanho de um minilote).

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

Agora podemos rodar a rede alimentando-a com um único tensor que contém todas as sequências do minilote:

```

X_batch = np.array([
    # t = 0      t = 1
    [[0, 1, 2], [9, 8, 7]], # instância0
    [[3, 4, 5], [0, 0, 0]], # instância1
    [[6, 7, 8], [6, 5, 4]], # instância2
    [[9, 0, 1], [3, 2, 1]], # instância3
])

with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})

```

E obtemos um único tensor `outputs_val` para todas as instâncias, todos os intervalos de tempo e todos os neurônios:

```

>>> print(outputs_val)
[[[-0.91279727  0.83698678 -0.89277941  0.80308062 -0.5283336 ]
 [-1.          1.          -0.99794829  0.99985468 -0.99273592]]

 [[-0.99994391  0.99951613 -0.9946925   0.99030769 -0.94413054]
 [ 0.48733309  0.93389565 -0.31362072  0.88573611  0.2424476 ]]

 [[-1.          0.99999875 -0.99975014  0.99956584 -0.99466234]
 [-0.99994856  0.99999434 -0.96058172  0.99784708 -0.9099462 ]]

 [[-0.95972425  0.99951482  0.96938795 -0.969908   -0.67668229]
 [-0.84596014  0.96288228  0.96856463 -0.14777924 -0.9119423 ]]]

```

No entanto, essa abordagem ainda cria um grafo que contém uma célula por intervalo de tempo. O grafo ficaria muito feio se houvesse 50 intervalos de tempo, seria como escrever um programa sem nunca utilizar loops (por exemplo,  $Y_0=f(0, X_0)$ ;  $Y_1=f(Y_0, X_1)$ ;  $Y_2=f(Y_1, X_2)$ ; ...;  $Y_{50}=f(Y_{49}, X_{50})$ ). Com um grafo tão grande, durante a retropropagação você poderia obter erros de falta de memória (OOM, do inglês) (especialmente com a memória limitada das placas GPU), pois ela deve armazenar todos os valores do tensor durante o forward pass e utilizá-los para calcular os gradientes durante o reverse pass.

Felizmente, existe uma solução melhor: a função `dynamic_rnn()`.

## Desenrolamento Dinâmico Através do Tempo

A função `dynamic_rnn()` utiliza uma operação `while_loop()` para executar o número apropriado de vezes na célula, e você pode definir a `swap_memory=True` se quiser que ela troque a memória GPU para a memória CPU durante a retropropagação a fim de evitar erros do tipo OOM. Convenientemente, em cada intervalo de tempo ela também aceita um único tensor para todas as entradas (formato `[None, n_steps, n_inputs]`) e exibe um

único tensor para todas as saídas a cada intervalo de tempo (formato `[None, n_steps, n_neurons]`); não há necessidade de empilhar, desempilhar ou transpor. Ao utilizar a função `dynamic_rnn()`, o código a seguir cria a mesma RNN de antes. É muito melhor!

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```



Durante a retropropagação, a operação `while_loop()` faz a mágica certa: armazena os valores do tensor para cada iteração durante o forward pass para que possa utilizá-los para calcular gradientes durante o reverse pass.

## Manipulando Sequências de Entrada de Comprimento Variável

Até agora, utilizamos apenas sequências de entrada de tamanho fixo (todas com exatamente dois passos). E se as sequências de entrada tiverem comprimentos variáveis (por exemplo, sentenças)? Nesse caso, você deve definir o argumento `sequence_length` quando chamar a função `dynamic_rnn()` (ou `static_rnn()`); deve ser um tensor 1D indicando o comprimento da sequência de entrada para cada instância. Por exemplo:

```
seq_length = tf.placeholder(tf.int32, [None])

[...]
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                     sequence_length=seq_length)
```

Por exemplo, suponha que a segunda sequência de entrada contenha apenas uma entrada em vez de duas. Ela deve ser preenchida com um vetor zero para encaixar no tensor de entrada `X` (porque a segunda dimensão do tensor de entrada é do tamanho da sequência mais longa — ou seja, 2).

```
X_batch = np.array([
    # passo 0      passo 1
    [[0, 1, 2], [9, 8, 7]], # instância0
    [[3, 4, 5], [0, 0, 0]], # instância 1 (com um vetor zero)
    [[6, 7, 8], [6, 5, 4]], # instância2
    [[9, 0, 1], [3, 2, 1]], # instância3
])
seq_length_batch = np.array([2, 1, 2, 2])
```

Claro, agora você precisa alimentar valores para ambos placeholders `X` e `seq_length`:

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states], feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Agora, a RNN produz vetores zero para cada vez que ultrapassa o comprimento da sequência (observe a saída da segunda instância para o segundo intervalo de tempo):

```
>>> print(outputs_val)
[[[-0.68579948 -0.25901747 -0.80249101 -0.18141513 -0.37491536]
 [-0.99996698 -0.94501185  0.98072106 -0.9689762  0.99966913]] # estado final

[[-0.99099374 -0.64768541 -0.67801034 -0.7415446  0.7719509 ] # estado final
 [ 0.          0.          0.          0.          0.        ]] # vetor zero

[[-0.99978048 -0.85583007 -0.49696958 -0.93838578  0.98505187]
 [-0.99951065 -0.89148796  0.94170523 -0.38407657  0.97499216]] # estado final

[[-0.02052618 -0.94588047  0.99935204  0.37283331  0.9998163 ]
 [-0.91052347  0.05769409  0.47446665 -0.44611037  0.89394671]]] # estado final
```

Além disso, o tensor `states` contém o estado final de cada célula (excluindo os vetores zero):

```
>>> print(states_val)
[[[-0.99996698 -0.94501185  0.98072106 -0.9689762  0.99966913] # t = 1
 [-0.99099374 -0.64768541 -0.67801034 -0.7415446  0.7719509 ] # t = 0 !!!
 [-0.99951065 -0.89148796  0.94170523 -0.38407657  0.97499216] # t = 1
 [-0.91052347  0.05769409  0.47446665 -0.44611037  0.89394671]] # t = 1
```

## Manipulando Sequências de Saída de Comprimento Variável

E se as sequências de saída também tiverem comprimentos variáveis? Se você souber antecipadamente o comprimento de cada sequência (por exemplo, se souber que será o mesmo comprimento que a sequência de entrada), poderá definir o parâmetro `sequence_length` conforme descrito acima. Infelizmente, isso não será possível: por exemplo, o comprimento de uma sentença traduzida é diferente do tamanho da sentença de entrada. Neste caso, a solução mais comum seria definir uma saída especial chamada *end-of-sequence token* (EOS token). Qualquer saída após a EOS deve ser ignorada (discutiremos isso mais adiante neste capítulo).

Ok, agora você sabe como construir uma rede RNN (ou, mais precisamente, uma rede RNN desenrolada através do tempo). Mas como você a treina?

## Treinando RNNs

Para treinar uma RNN, o truque é desenrolá-la através do tempo (como acabamos de fazer) e, então, utilizar a retropropagação comum (veja a Figura 14-5), estratégia chamada de *retropropagação através do tempo* (BPTT, do inglês).

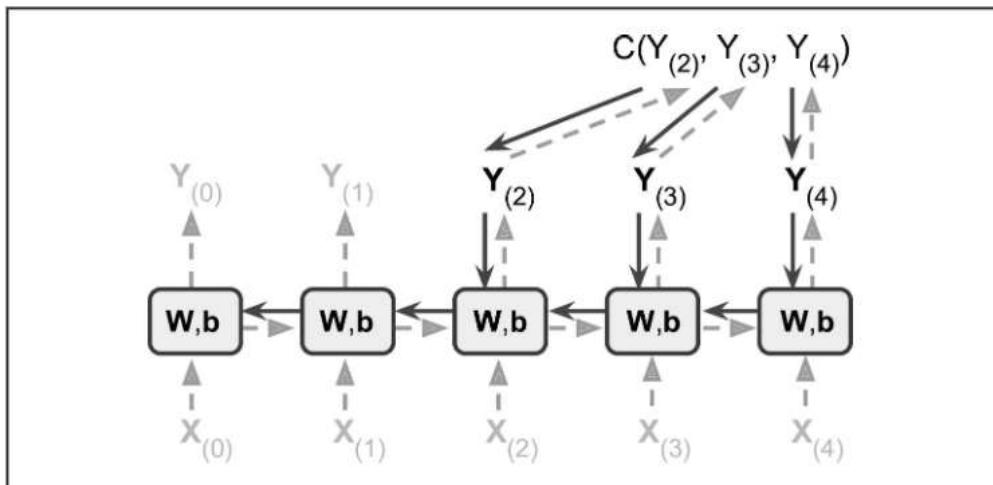


Figura 14-5. Retropropagação ao longo do tempo

Assim como na retropropagação normal há um primeiro forward pass pela rede desenrolada (representado pelas setas tracejadas); então a sequência de saída é avaliada com a utilização de uma função de custo  $c(Y_{t_{\min}}, Y_{t_{\min}+1}, \dots, Y_{t_{\max}})$  (na qual  $t_{\min}$  e  $t_{\max}$  são os primeiros e últimos intervalos de tempo de saída, sem contar as saídas ignoradas) e os gradientes dessa função de custo são propagados para trás através da rede desenrolada (representada pelas setas sólidas); e, finalmente, os parâmetros do modelo são atualizados com a utilização dos gradientes calculados durante a BPTT. Note que os gradientes fluem para trás através de todas as saídas utilizadas pela função de custo, não apenas através da saída final (por exemplo, na Figura 14-5 a função de custo é calculada utilizando as últimas três saídas da rede,  $Y_{(2)}$ ,  $Y_{(3)}$  e  $Y_{(4)}$ ), desta forma gradientes fluem através das três saídas, mas não através de  $Y_{(0)}$  e  $Y_{(1)}$ ). Além disso, como os mesmos parâmetros  $W$  e  $b$  são utilizados em cada etapa de tempo, a retropropagação fará a coisa certa e somará todas as etapas do tempo.

## Treinando um Classificador de Sequência

Treinaremos uma RNN para classificar imagens MNIST. Uma rede neural convolucional seria mais adequada para a classificação de imagens (consulte o Capítulo 13), mas este é um exemplo simples com o qual você já está familiarizado. Trataremos cada imagem como uma sequência de 28 linhas de 28 pixels cada (já que cada imagem MNIST contém  $28 \times 28$  pixels). Utilizaremos células de 150 neurônios recorrentes, além de uma camada totalmente conectada contendo 10 neurônios (um por classe) conectados à saída do último intervalo de tempo, seguidos por uma camada softmax (veja a Figura 14-6).

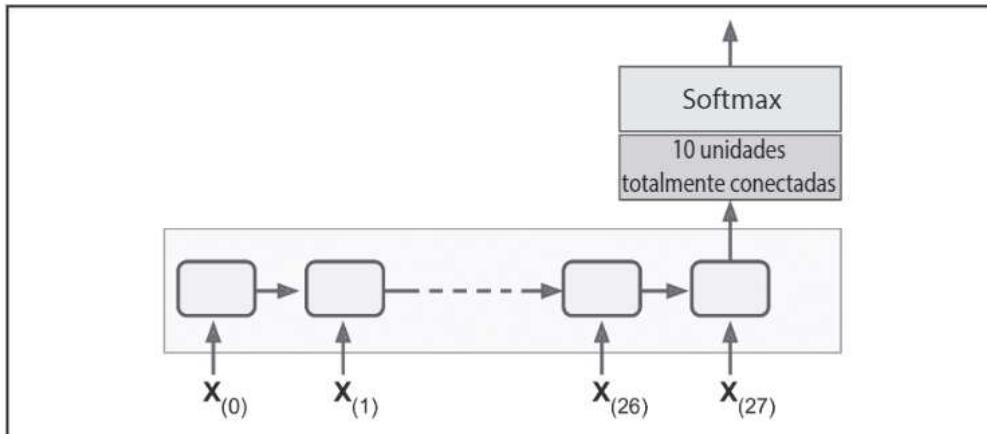


Figura 14-6. Classificador de Sequência

A fase de construção é bastante simples; é praticamente a mesma do classificador MNIST que criamos no Capítulo 10, exceto que uma RNN desenrolada substitui as camadas ocultas. Observe que a camada totalmente conectada está conectada ao tensor de estados que contém apenas o estado final da RNN (ou seja, a saída 28). Observe também que  $y$  é um placeholder para as classes-alvo.

```

n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10

learning_rate = 0.001

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                          logits=logits)
loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()

```

Agora, como é esperado pela rede, carregaremos os dados do MNIST e remodelaremos os dados de teste para `[batch_size, n_steps, n_inputs]`. Reformularemos os dados de treinamento em breve.

```

from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels

```

Agora estamos prontos para treinar a RNN. A fase de execução é igual à do classificador MNIST no Capítulo 10, exceto pelo fato de que reformulamos cada lote de treinamento antes de fornecê-lo à rede.

```

n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

```

A saída se parece com isso:

```

0 Train accuracy: 0.94 Test accuracy: 0.9308
1 Train accuracy: 0.933333 Test accuracy: 0.9431
[...]
98 Train accuracy: 0.98 Test accuracy: 0.9794
99 Train accuracy: 1.0 Test accuracy: 0.9804

```

Nada mal, temos mais de 98% de acurácia! Além disso, inicializando os pesos da RNN com a utilização da inicialização He, você certamente obteria um melhor resultado ajustando os hiperparâmetros, treinando por mais tempo ou adicionando um pouco de regularização (por exemplo, dropout).



Você pode especificar um inicializador para a RNN envolvendo seu código de construção em um escopo variável (por exemplo, utilize `variable_scope("rnn", initializer=variance_scaling_initializer())` para utilizar a inicialização He).

## Treinando para Prever Séries Temporais

Agora daremos uma olhada em como lidar com séries temporais como: preços de ações, temperatura do ar, padrões de ondas cerebrais e assim por diante. Nesta seção, treinaremos uma RNN para prever o próximo valor em uma série temporal. Cada instância de treinamento será uma sequência aleatoriamente selecionada da série temporal de 20

valores consecutivos e a sequência de destino é a mesma da sequência de entrada, com a exceção dela ser mudada em um intervalo de tempo no futuro (veja a Figura 14-7).

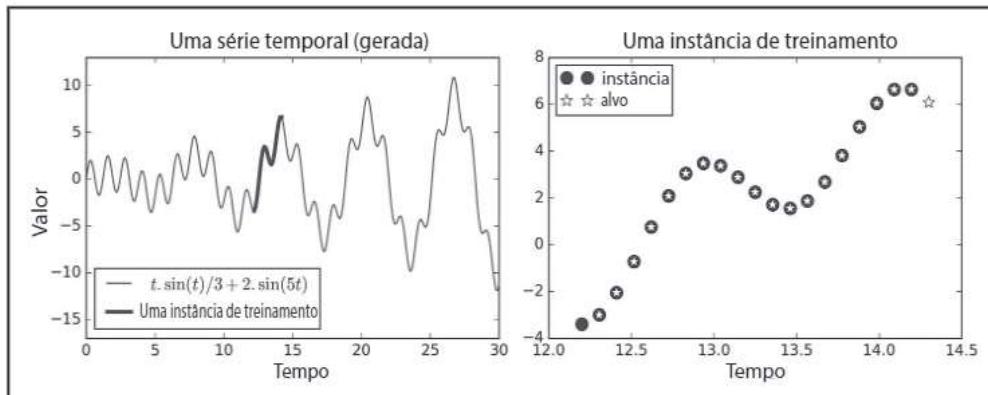


Figura 14-7. Séries temporais (à esquerda) e uma instância de treinamento dessa série (à direita)

Primeiro, criaremos a RNN, que conterá 100 neurônios recorrentes e a desenrolaremos em 20 intervalos de tempo, já que cada instância de treinamento terá 20 entradas. Cada entrada conterá apenas uma característica (o valor naquele momento). Os alvos são também sequências de 20 entradas, cada uma contendo um único valor. O código é quase o mesmo de antes:

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```



No geral, você teria mais do que apenas um recurso de entrada. Por exemplo, se você estivesse tentando prever preços de ações, provavelmente teria muitos outros recursos de entrada em cada intervalo de tempo como preços de ações concorrentes, classificações de analistas ou qualquer outro recurso que ajudasse o sistema a fazer suas previsões.

A cada intervalo de tempo temos agora um vetor de saída de tamanho 100, mas o que realmente queremos é um valor único de saída em cada intervalo de tempo. A solução mais simples é envolver a célula em um `OutputProjectionWrapper`. Uma célula wrapper age como uma célula normal fazendo proxy de cada chamada de método para uma célula subjacente, mas também adiciona alguma funcionalidade. O `OutputProjectionWrapper` adiciona uma camada totalmente conectada de neurônios lineares (ou seja, sem qualquer função

de ativação) no topo de cada saída (mas isso não afeta o estado da célula). Todas essas camadas totalmente conectadas compartilham os mesmos pesos e termos de polarização (treináveis). A RNN resultante é representada na Figura 14-8.

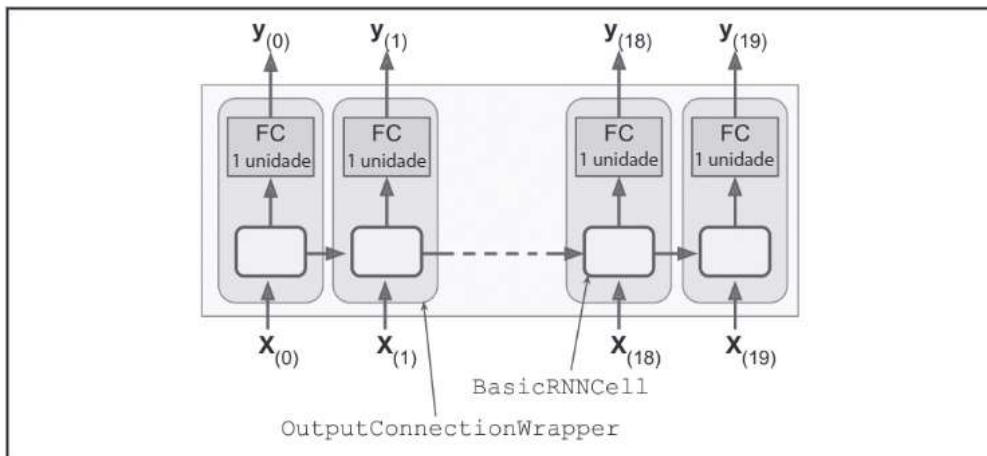


Figura 14-8. Células RNN utilizando projeções de saída

Envolver uma célula é bem fácil. Ajustaremos o código anterior envolvendo o `BasicRNNCell` em um `OutputProjectionWrapper`:

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu),
    output_size=n_outputs)
```

Por enquanto, tudo bem. Agora precisamos definir a função de custo. Utilizaremos o Erro Médio Quadrático (MSE), como fizemos em tarefas de regressão anteriores. Em seguida, como de costume, criaremos um otimizador Adam, o treinamento op e a inicialização da variável op:

```
learning_rate = 0.001

loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Agora para a fase de execução:

```
n_iterations = 1500
batch_size = 50

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = [...] # fetch the next training batch
```

```

sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
if iteration % 100 == 0:
    mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
    print(iteration, "\tMSE:", mse)

```

A saída do programa deve ser parecida com esta:

```

0      MSE: 13.6543
100    MSE: 0.538476
200    MSE: 0.168532
300    MSE: 0.0879579
400    MSE: 0.0633425
[...]

```

Uma vez treinado o modelo, você pode fazer previsões:

```

X_new = [...] # Novas sequências
y_pred = sess.run(outputs, feed_dict={X: X_new})

```

A Figura 14-9 mostra a sequência prevista para a instância que examinamos anteriormente (na Figura 14-7) após apenas 1 mil iterações de treinamento.

Embora a utilização de um `OutputProjectionWrapper` seja a solução mais simples para reduzir a dimensionalidade das sequências de saída da RNN para apenas um valor por intervalo de tempo (por instância), ela não é a mais eficiente. Existe uma solução mais eficiente, porém mais complicada: você pode remodelar as saídas da RNN de `[batch_size, n_steps, n_neurons]` para `[batch_size * n_steps, n_neurons]`, em seguida aplicar uma única camada totalmente conectada com o tamanho de saída apropriada (no nosso caso apenas 1) que resultará em um tensor de saída no formato `[batch_size * n_steps, n_outputs]` e, então, remodelar este tensor para `[batch_size, n_steps, n_outputs]`. Estas operações estão representadas na Figura 14-10.

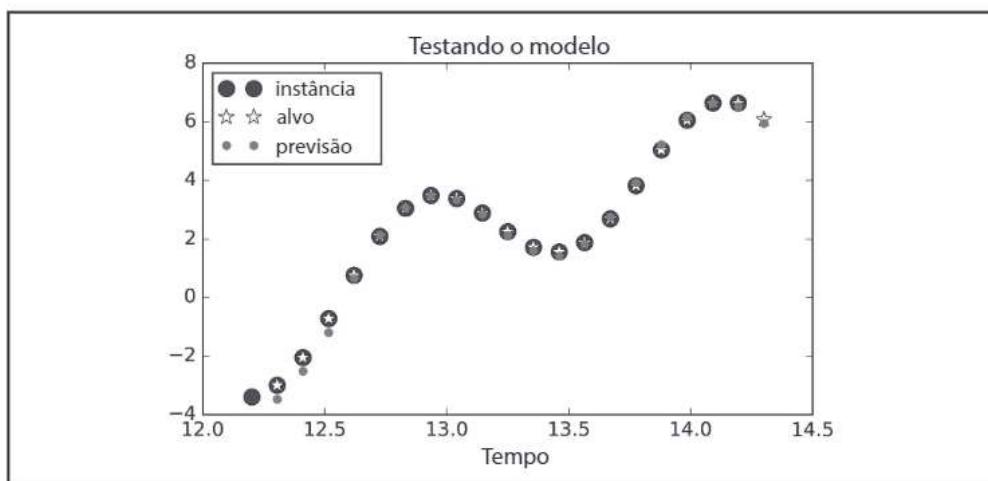


Figura 14-9. Previsões de séries temporais

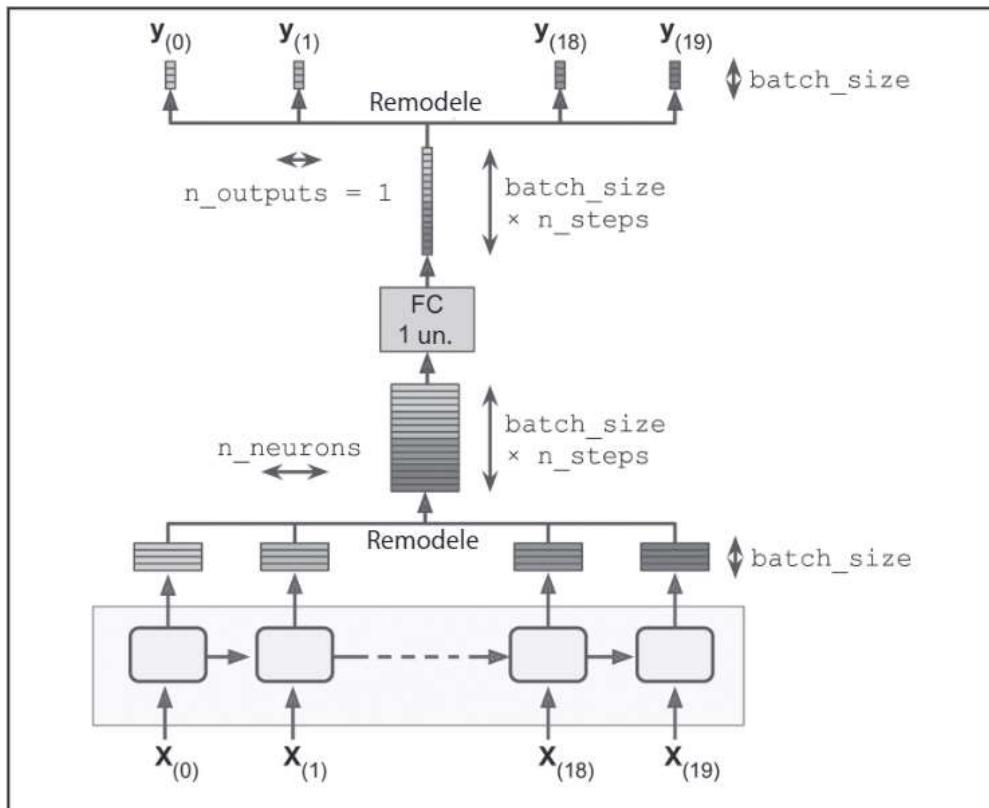


Figura 14-10. Empilhe todas as saídas, aplique a projeção e forneça o resultado

Para implementar esta solução, primeiro revertemos para uma célula básica, sem o `OutputProjectionWrapper`:

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons, activation=tf.nn.relu)
rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

Em seguida, com a utilização da operação `reshape()`, empilhamos todas as saídas, aplicamos a camada linear totalmente conectada (sem usar nenhum função de ativação; isto é, apenas uma projeção) e, finalmente, desempilhamos todas as saídas utilizando `reshape()` novamente:

```
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
```

O restante do código é o mesmo de antes, podendo fornecer um aumento de velocidade significativo, pois há apenas uma camada totalmente conectada, em vez de uma por intervalo de tempo.

## RNN Criativa

Agora que temos um modelo que pode prever o futuro, podemos usá-lo para gerar algumas sequências criativas, como explicado no início do capítulo. Precisamos fornecer apenas uma sequência semente que contenha valores `n_steps` (por exemplo, cheio de zeros), utilizar o modelo para prever o próximo valor, acrescentar este valor previsto à sequência, fornecer os últimos valores `n_steps` para o modelo e, então, prever o próximo valor e assim por diante. Este processo gera uma nova sequência que tem alguma semelhança com a série temporal original (veja a Figura 14-11).

```
sequence = [0.] * n_steps
for iteration in range(300):
    X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    sequence.append(y_pred[0, -1, 0])
```

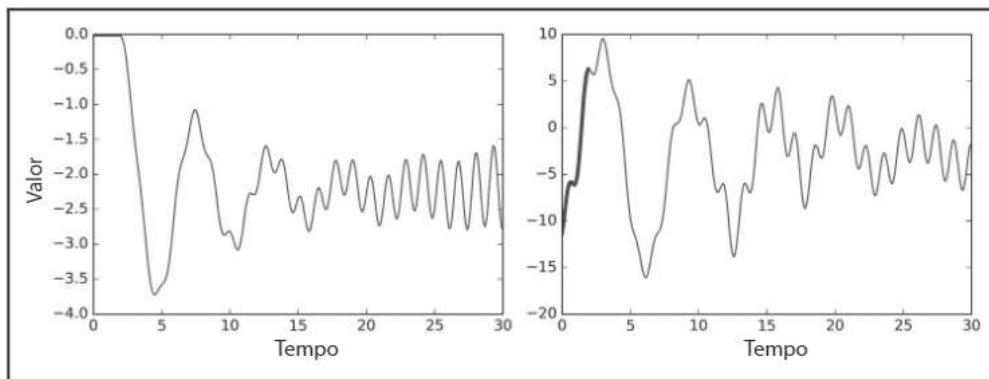


Figura 14-11. Sequências criativas permeadas com zeros (esquerda) ou com uma instância (direita)

Agora você pode tentar fornecer para uma RNN todos os seus álbuns do John Lennon e verificar se ela consegue gerar a próxima “Imagine”. Você provavelmente precisará de uma RNN muito mais poderosa, com mais neurônios, e também muito mais profunda. Daremos uma olhada nas RNNs profundas agora.

## RNNs Profundas

É muito comum, como mostrado na Figura 14-12, empilhar múltiplas camadas de células, resultando em uma *RNN profunda*.

Para implementar uma RNN profunda no TensorFlow, você pode criar várias células e empilhá-las em uma `MultiRNNCell`. No código a seguir, empilharemos três células

idênticas (mas você poderia muito bem utilizar vários tipos de células com um número diferente de neurônios):

```
n_neurons = 100
n_layers = 3

layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
                                         activation=tf.nn.relu)
          for layer in range(n_layers)]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```

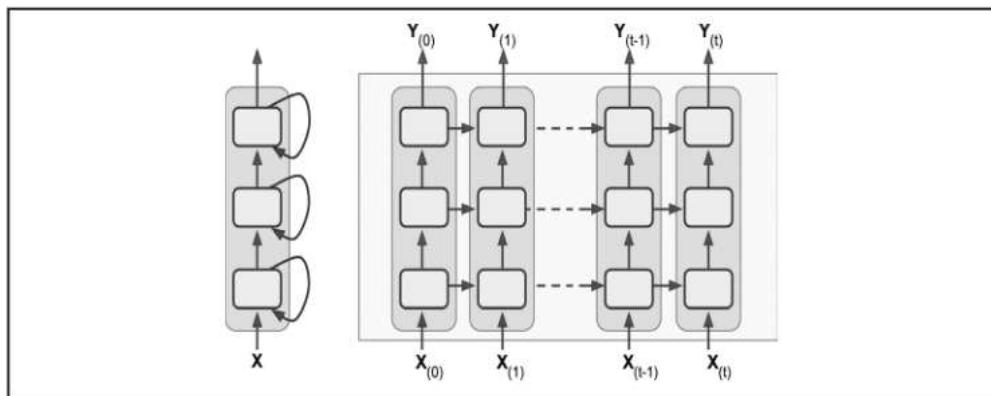


Figura 14-12. RNN profunda (esquerda) desenrolada através do tempo (direita)

Isso é tudo! A variável `states` é uma tupla que contém um tensor por camada, cada um representando o estado final daquela célula da camada (com formato `[batch_size, n_neurons]`). Se você configurar `state_is_tuple=False` ao criar uma `MultiRNNCell`, então `states` se torna um tensor único contendo os estados de cada camada concatenado no eixo coluna (ou seja, seu formato é `[batch_size, n_layers * n_neurons]`). Observe que este comportamento era o padrão antes do TensorFlow 0.11.0.

## Distribuindo uma RNN Profunda Através de Múltiplas GPUs

O Capítulo 12 apontou que, ao fixarmos cada camada a uma GPU diferente, podemos distribuir eficientemente RNNs profundas através de múltiplas GPUs (veja a Figura 12-16). No entanto, não funcionará se você tentar criar cada célula em um bloco `device()` diferente:

```
with tf.device("/gpu:0"): # PÉSSIMO! Foi ignorado.
    layer1 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)

with tf.device("/gpu:1"): # PÉSSIMO! Ignorado novamente.
    layer2 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

Isto falhará porque uma `BasicRNNCell` é uma fábrica de células, não uma célula *per se* (como mencionado anteriormente); nenhuma célula é criada quando você cria a fábrica e, portanto, nenhuma variável também. O bloco do dispositivo é simplesmente ignorado e as células são criadas depois. Quando você chama `dynamic_rnn()`, ela chama a `MultiRNNCell`, que chama cada `BasicRNNCell` individual, que cria as células atuais (incluindo suas variáveis). Infelizmente, nenhuma dessas classes fornece uma maneira de controlar os dispositivos nos quais as variáveis são criadas. Se você tentar colocar a chamada `dynamic_rnn()` dentro de um bloco do dispositivo, toda a RNN será fixada em um único dispositivo. Então, isso significa que você está preso? Felizmente não! O truque é criar seu próprio wrapper de célula (ou utilizar a classe `tf.contrib.rnn.DeviceWrapper`, que foi adicionada no TensorFlow 1.1):

```
import tensorflow as tf

class DeviceCellWrapper(tf.contrib.rnn.RNNCell):
    def __init__(self, device, cell):
        self._cell = cell
        self._device = device

    @property
    def state_size(self):
        return self._cell.state_size

    @property
    def output_size(self):
        return self._cell.output_size

    def __call__(self, inputs, state, scope=None):
        with tf.device(self._device):
            return self._cell(inputs, state, scope)
```

Esse wrapper simplesmente faz o proxy [solicitação] de toda chamada de método para outra célula, exceto que envolve a função `_call_()` dentro de um bloco do dispositivo.<sup>2</sup> Agora, você pode distribuir cada camada em uma GPU diferente:

```
devices = ["/gpu:0", "/gpu:1", "/gpu:2"]
cells = [DeviceCellWrapper(dev, tf.contrib.rnn.BasicRNNCell(num_units=n_neurons))
         for dev in devices]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
```



Não configure `state_is_tuple=False` pois a `MultiRNNCell` concatenará todos os estados da célula em um tensor único em uma única GPU.

---

<sup>2</sup> Usa o padrão de design *decorator*.

## Aplicando o Dropout

Se você construir uma RNN muito profunda, ela pode acabar se sobreajustando ao conjunto de treinamento. Uma técnica comum para evitarmos isso é aplicar o dropout (apresentado no Capítulo 11). Como de costume, podemos adicionar uma camada de dropout antes ou depois da RNN, mas se você também quiser aplicar o dropout entre as camadas da RNN terá que utilizar um `DropoutWrapper`. O código a seguir aplica na RNN o dropout às entradas de cada camada:

```
keep_prob = tf.placeholder_with_default(1.0, shape=())
cells = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
         for layer in range(n_layers)]
cells_drop = [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
              for cell in cells]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
rnn_outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
# O resto da fase de construção é igual à anterior.
```

Você pode fornecer o valor que quiser ao placeholder `keep_prob` (geralmente 0.5) durante o treinamento:

```
n_iterations = 1500
batch_size = 50
train_keep_prob = 0.5

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        _, mse = sess.run([training_op, loss],
                         feed_dict={X: X_batch, y: y_batch,
                                    keep_prob: train_keep_prob})
    saver.save(sess, "./my_dropout_time_series_model")
```

Desativando o dropout durante o teste, você deve deixar o `keep_prob` no padrão 1.0 (lembre-se de que ele deve estar ativo somente durante o treinamento):

```
with tf.Session() as sess:
    saver.restore(sess, "./my_dropout_time_series_model")
    X_new = [...] # alguns dados de teste
    y_pred = sess.run(outputs, feed_dict={X: X_new})
```

Observe que também é possível, desde o TensorFlow 1.1, aplicar o dropout às saídas, e também é possível, com a utilização do `state_keep_prob`, aplicar o dropout ao estado da célula.

Com isso, você deve ser capaz de treinar todos os tipos de RNNs! Infelizmente, as coisas ficarão um pouco mais difíceis se você quiser treinar uma RNN em longas sequências. Veremos por quê, e o quê você pode fazer a respeito.

## A Dificuldade de Treinar Muitos Intervalos de Tempo

Para treinar uma RNN em longas sequências, tornando-a uma rede desenrolada muito profunda, você precisará executá-la ao longo de vários intervalos de tempo. Assim como qualquer rede neural profunda, ela pode demorar uma eternidade no treinamento e sofrer com o problema dos gradientes vanishing/exploding (discutido no Capítulo 11). Muitos dos truques que discutimos para aliviar este problema também podem ser utilizados para RNNs profundas desenroladas: boa inicialização do parâmetro, funções de ativação não saturadas (por exemplo, ReLU), Normalização de Lotes, Recorte de Gradiente e otimizadores mais rápidos. No entanto, o treinamento ainda será muito lento, até mesmo se a RNN precisar lidar com sequências moderadamente longas (por exemplo, 100 entradas).

A solução mais simples e comum para esse problema durante o treinamento é desenrolar a RNN somente durante um número limitado de intervalos de tempo, o que é chamado de *retropropagação truncada através do tempo*. Você pode implementá-lo no TensorFlow simplesmente truncando as sequências de entrada. Por exemplo, você simplesmente reduziria `n_steps` durante o treinamento no problema da previsão de séries temporais. O problema, claro, é que o modelo não será capaz de aprender padrões de longo prazo. Uma solução alternativa seria garantir que essas sequências abreviadas contenham dados antigos e recentes para que o modelo aprenda a utilizar ambos (por exemplo, a sequência pode conter dados mensais dos últimos cinco meses e, em seguida, dados semanais das últimas cinco semanas, em seguida, dados diários dos últimos cinco dias). Mas essa solução alternativa tem seus limites: os dados refinados do ano passado foram realmente úteis? E se houve um evento breve, mas significativo, que absolutamente devia ser levado em consideração, mesmo anos depois (por exemplo, o resultado de uma eleição)?

Um segundo problema enfrentado pelas RNNs de longa duração é o fato de que a memória das primeiras entradas desaparece gradualmente, além do longo tempo de treinamento. De fato, algumas informações são perdidas após cada intervalo de tempo devido às transformações pelas quais os dados passam ao atravessar uma RNN. Depois de um tempo, o estado da RNN praticamente não contém traços das primeiras entradas e isso pode ser um erro severo. Por exemplo, digamos que você queira realizar uma análise de sentimento em um longo artigo que comece com as três palavras "Adorei esse filme", mas o resto da resenha lista muitas coisas que poderiam ter tornado o filme ainda melhor. Se a RNN gradualmente esquece as quatro primeiras palavras, ela interpretará equivocadamente o artigo. Foram introduzidos vários tipos de células com memória de longo prazo para resolver esse problema, e elas se mostraram tão bem sucedidas que as células básicas já não são muito utilizadas. Veremos primeiro a mais popular dessas células de memória longa: a célula LSTM.

## Célula LSTM

A célula *Memória Longa de Curto Prazo* (LSTM, do inglês) foi proposta em 1997 (<https://goo.gl/j39AGv>)<sup>3</sup> por Sepp Hochreiter e Jürgen Schmidhuber e aperfeiçoada por muitos pesquisadores como Alex Graves, Haşim Sak (<https://goo.gl/6BHh81>),<sup>4</sup> Wojciech Zaremba (<https://goo.gl/SZ9kzB>)<sup>5</sup> e muitos outros ao longo dos anos. Ela poderá ser utilizada de maneira muito parecida com uma célula básica se você a considera como uma caixa preta, mas terá um desempenho muito melhor; o treinamento convergirá mais rápido e detectará dependências de longo prazo nos dados. Você pode simplesmente utilizar uma `BasicLSTMCell` em vez de uma `BasicRNNCell` no TensorFlow:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

Células LSTM gerenciam dois vetores de estado e, por razões de desempenho, eles são mantidos separados por padrão. Configurando `state_is_tuple=False` ao criar uma `BasicLSTMCell` você pode mudar este comportamento padrão.

Então, como funciona uma célula LSTM? Sua arquitetura básica é mostrada na Figura 14-13.

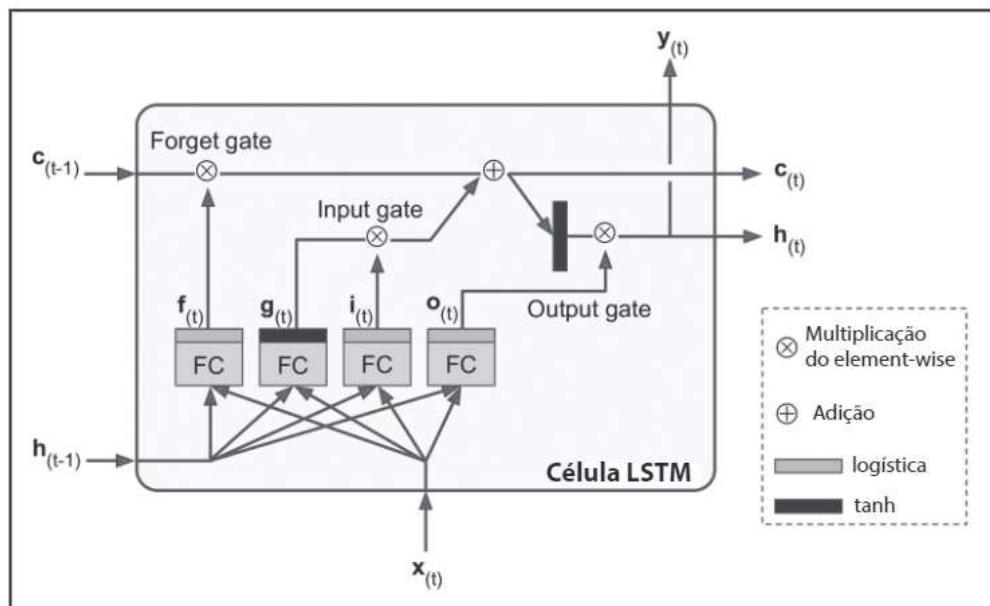


Figura 14-13. Célula LSTM

<sup>3</sup> “Long Short-Term Memory”, S. Hochreiter and J. Schmidhuber (1997).

<sup>4</sup> “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modelling”, H. Sak *et al.* (2014).

<sup>5</sup> “Recurrent Neural Network Regularization”, W. Zaremba *et al.* (2015).

Se você não olhar para o que está dentro da caixa, a célula LSTM se parece exatamente com uma célula comum, exceto pelo seu estado dividido em dois vetores:  $h_{(t)}$  e  $c_{(t)}$  (“c” se refere a “célula”). Você pode pensar em  $h_{(t)}$  como o estado de curto prazo e  $c_{(t)}$  como o estado de longo prazo.

Agora abriremos a caixa! A ideia-chave é que a rede possa aprender o que armazenar no estado de longo prazo, o que jogar fora e o que ler a partir dele. Como o estado de longo prazo  $c_{(t-1)}$  atravessa a rede da esquerda para a direita, você pode ver que ele passa primeiro por um *forget gate*, descartando algumas memórias e adicionando algumas novas memórias através da operação de adição (que acrescenta as memórias que foram selecionadas por um *input gate*). O resultado  $c_{(t)}$  é enviado diretamente, sem qualquer transformação adicional. Então algumas memórias são descartadas e algumas são adicionadas a cada intervalo de tempo. Além disso, após a operação de adição, o estado de longo prazo é copiado e passado pela função  $\tanh$  e, em seguida, seu resultado é filtrado pelo *output gate*, produzindo o estado de curto prazo  $h_{(t)}$  (que é igual à saída da célula para este intervalo de tempo  $y_{(t)}$ ). Agora, veremos de onde vêm as novas memórias e como funcionam as portas.

Primeiro, o vetor de entrada atual  $x_{(t)}$  e o estado de curto prazo anterior  $h_{(t-1)}$  são fornecidos para quatro camadas diferentes totalmente conectadas. Todos servem a um propósito diferente:

- A camada principal é aquela que produz  $g_{(t)}$  e, normalmente, assume o papel de analisar as entradas atuais  $x_{(t)}$  e o estado anterior (de curto prazo)  $h_{(t-1)}$ . Em uma célula básica não há nada além desta camada, e sua saída vai direto para  $y_{(t)}$  e  $h_{(t)}$ . Em contraste, em uma célula LSTM a saída dessa camada não se extingue diretamente, mas é armazenada parcialmente no estado de longo prazo;
- As três outras camadas são *gate controllers*. Como elas utilizam a função de ativação logística, suas saídas variam de 0 a 1. Como você pode ver, suas saídas são fornecidas para operações de multiplicação elemento por elemento, então, se eles produzem 0s, fecham o gate e, se geram 1s, os gates se abrem. Especificamente:
  - O *forget gate* (controlado por  $f_{(t)}$ ) controla quais partes do estado de longo prazo devem ser apagadas;
  - O *input gate* (controlado por  $i_{(t)}$ ) controla quais partes de  $g_{(t)}$  devem ser adicionadas ao estado de longo prazo (é por isso que dissemos que era apenas “parcialmente armazenado”);
  - Finalmente, o *output gate* (controlado por  $o_{(t)}$ ) controla quais partes do estado de longo prazo devem ser lidas e exibidas neste intervalo de tempo (ambos para  $h_{(t)}$  e  $y_{(t)}$ ).

Em resumo, uma célula LSTM pode aprender a reconhecer uma entrada importante (que é o papel do *input gate*), armazená-la no estado de longo prazo, aprender a preservá-la pelo tempo necessário (esse é o papel do *forget gate*) e aprender a extrai-la sempre que for preciso. Isso explica por que elas têm sido surpreendentemente bem-sucedidas em capturar padrões de longo prazo em séries temporais, textos longos, gravações de áudio e muito mais.

A Equação 14-3 resume como calcular em cada intervalo de tempo para uma única instância: o estado de longo prazo da célula, seu estado de curto prazo e sua saída (as equações para um minilote inteiro são muito semelhantes).

#### *Equação 14-3. Cálculos LSTM*

$$\begin{aligned}\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\ \mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\ \mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\ \mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\ \mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)\end{aligned}$$

- $\mathbf{W}_{xi}$ ,  $\mathbf{W}_{xf}$ ,  $\mathbf{W}_{xo}$ ,  $\mathbf{W}_{xg}$  são as matrizes de peso de cada uma das quatro camadas para suas conexões com o vetor de entrada  $\mathbf{x}_{(t)}$ ;
- $\mathbf{W}_{hi}$ ,  $\mathbf{W}_{hf}$ ,  $\mathbf{W}_{ho}$ , e  $\mathbf{W}_{hg}$  são as matrizes de peso de cada uma das quatro camadas para suas conexões com o estado anterior de curto prazo  $\mathbf{h}_{(t-1)}$ ;
- $\mathbf{b}_i$ ,  $\mathbf{b}_f$ ,  $\mathbf{b}_o$ , e  $\mathbf{b}_g$  são os termos de polarização para cada uma das quatro camadas. Observe que o TensorFlow inicializa  $\mathbf{b}_f$  para um vetor cheio de 1s em vez de 0s, evitando esquecer tudo no início do treinamento.

## Conexões Peephole

Em uma célula LSTM básica, os controladores do gate podem olhar apenas para a entrada  $\mathbf{x}_{(t)}$  e o estado anterior de curto prazo  $\mathbf{h}_{(t-1)}$ . Permitir também que observem o estado de longo prazo pode ser uma boa ideia para dar-lhes um pouco mais de contexto. Esta ideia foi proposta por Felix Gers e Jürgen Schmidhuber em 2000 (<https://goo.gl/ch8xz3>),<sup>6</sup> quando propuseram uma variante LSTM com conexões extras chamadas *conexões peephole*: o estado de longo prazo anterior  $\mathbf{c}_{(t-1)}$  é adicionado como uma entrada para os

---

<sup>6</sup> “Recurrent Nets that Time and Count”, F. Gers and J. Schmidhuber (2000).

controladores do forget gate e do input gate e o estado de longo prazo  $c_{(t)}$  é adicionado como entrada para o controlador do output gate.

Para implementar as conexões peephole no TensorFlow você deve utilizar a `LSTMCell` em vez da `BasicLSTMCell` e configurar `use_peepholes=True`:

```
lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons, use_peepholes=True)
```

Existem muitas outras variantes da célula LSTM e uma popular é a célula GRU que veremos agora.

## Célula GRU

A célula *Gated Recurrent Unit* (GRU) foi proposta por Kyunghyun Cho *et al.* em um artigo de 2014 (<https://goo.gl/ZnAEOZ>)<sup>7</sup> que também introduziu a rede Encoder–Decoder que mencionamos anteriormente (veja a Figura 14-14).

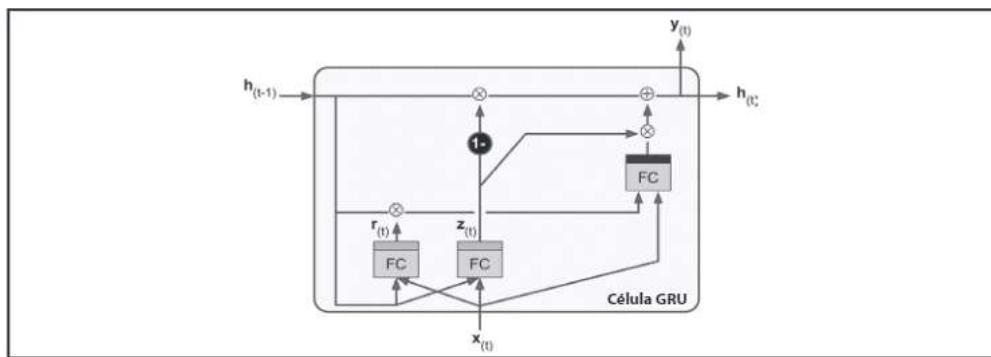


Figura 14-14. Célula GRU

A célula GRU é uma versão simplificada da célula LSTM e parece funcionar muito bem<sup>8</sup> (o que explica sua crescente popularidade). As principais simplificações são:

- Ambos os vetores de estado são mesclados em um único vetor  $h_{(t)}$ ;
- Um único gate controller controla tanto o forget gate quanto o input gate. Se o gate controller gerar um 1 o forget gate é aberto e o input gate é fechado. Se o resultado for 0, o oposto acontece. Em outras palavras, sempre que uma memória precisar ser armazenada, o local onde ela será armazenada será apagado primeiro. Esta é realmente uma variante frequente para a célula LSTM em si;

<sup>7</sup> "Learning Phrase Representations using RNN Encoder – Decoder for Statistical Machine Translation", K. Cho *et al.* (2014).

<sup>8</sup> Um artigo de 2015 por Klaus Greff *et al.*, "LSTM: A Search Space Odyssey", (<http://goo.gl/hZB4KW>) parece mostrar que todas as variantes LSTM têm desempenho semelhante.

- Não há porta de saída; o vetor de estado completo é gerado a cada intervalo de tempo. No entanto, há um novo controlador da porta que controla qual parte do estado anterior será mostrada para a camada principal.

A Equação 14-4 resume como calcular o estado da célula a cada intervalo de tempo para uma única instância.

*Equação 14-4. Cálculos da GRU*

$$\begin{aligned} z_{(t)} &= \sigma\left(\mathbf{W}_{xz}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\ r_{(t)} &= \sigma\left(\mathbf{W}_{xr}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\ g_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot (r_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right) \\ \mathbf{h}_{(t)} &= z_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - z_{(t)}) \otimes g_{(t)} \end{aligned}$$

Criar uma célula GRU no TensorFlow é trivial:

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

As células LSTM ou GRU são uma das principais razões por trás do sucesso das RNNs nos últimos anos, principalmente para aplicações em *processamento de linguagem natural* (PNL).

## Processamento de Linguagem Natural

A maioria dos aplicativos de última geração da PNL, como tradução automática, resumo automático, análise sintática, análise de sentimento e muito mais, agora se baseia (pelo menos em parte) em RNNs. Nesta última seção, daremos uma rápida olhada no modelo de tradução automática. Este tópico é muito bem coberto pelos incríveis tutoriais do TensorFlow Word2Vec (<https://goo.gl/edArdi>) e Seq2Seq (<https://goo.gl/L82gvS>), então você definitivamente deveria conferi-los.

## Word Embeddings

Antes de começarmos, precisamos escolher uma representação de palavras. Uma opção poderia ser representar cada palavra utilizando um vetor one-hot. Suponha que seu vocabulário contenha 50 mil palavras, então a enésima palavra seria representada como um vetor de 50 mil dimensões, cheio de “0” com exceção de um “1” na enésima posição. No entanto, essa representação esparsa não seria eficiente com um vocabulário tão grande. Você quer que palavras semelhantes tenham representações semelhantes, facilitando a generalização do que o modelo aprende sobre uma palavra, para todas as palavras semelhantes.

Por exemplo, se o modelo diz que “eu bebo leite” é uma sentença válida, e se ele sabe que “leite” está perto de “água”, mas longe de “sapatos”, então também saberá que “eu bebo água” provavelmente é uma sentença válida, enquanto “eu bebo sapatos” provavelmente não é. Mas como você pode chegar a uma representação tão significativa?

A solução mais comum é representar cada palavra no vocabulário com a utilização de um vetor razoavelmente pequeno e denso (por exemplo, 150 dimensões), chamado *embedding*, e simplesmente deixar a rede neural aprender uma boa embedding para cada palavra durante o treinamento. No início do treinamento, as word embeddings são escolhidas aleatoriamente, mas durante o treinamento a retropropagação as move automaticamente de forma a ajudar a rede neural a executar sua tarefa. Normalmente, isso significa que palavras semelhantes se agruparão gradualmenteumas às outras e até terminarão organizadas de maneira bastante significativa. Por exemplo, as word embeddings podem ser posicionadas ao longo de vários eixos que representam gênero, singular/plural, adjetivo/substantivo e assim por diante. O resultado pode ser verdadeiramente surpreendente.<sup>9</sup>

Você primeiro precisa criar no TensorFlow a variável que representa as embeddings para cada palavra em seu vocabulário (inicializado aleatoriamente):

```
vocabulary_size = 50000
embedding_size = 150

init_embeds = tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0)
embeddings = tf.Variable(init_embeds)
```

Agora, suponha que você queira alimentar em sua rede neural a frase “eu bebo leite”, comece pré-processando a frase e dividindo-a em uma lista de palavras conhecidas. Por exemplo, você pode remover caracteres desnecessários, substituir palavras desconhecidas por um token predefinido como “[UNK]”, substituir valores numéricos por “[NUM]”, substituir URLs por “[URL]” e assim por diante. Depois de ter uma lista de palavras conhecidas, você pode procurar o identificador de número inteiro de cada palavra (de 0 a 49999) em um dicionário, por exemplo [72, 3335, 288]. Nesse ponto, você está pronto para fornecer esses identificadores de palavras ao TensorFlow com a utilização de um placeholder, e aplicar a função `embedding_lookup()` para obter as embeddings correspondentes:

```
train_inputs = tf.placeholder(tf.int32, shape=[None]) # de ids...
embed = tf.nn.embedding_lookup(embeddings, train_inputs) # ...a embeddings
```

Uma vez que seu modelo tenha aprendido boas embeddings ele pode ser reutilizado de forma bastante eficiente em qualquer aplicação NLP: afinal, “leite” ainda está perto de “água” e longe de “sapatos”, não importa qual seja sua aplicação. Na verdade, em vez de treinar suas próprias word embeddings, você pode querer baixá-las de palavras pré-treinadas.

---

<sup>9</sup> Para mais detalhes, confira o ótimo post de Christopher Olah (<https://goo.gl/5rLNTj>), ou a série de posts de Sebastian Ruder (<https://goo.gl/oJjiE>).

nadas. Assim como quando reutilizamos camadas pré-treinadas (veja o Capítulo 11), você pode escolher congelar a variável embeddings pré-treinada (por exemplo, criando as `embeddings` utilizando `trainable=False`) ou deixando a retropropagação ajustá-las para sua aplicação. A primeira opção acelerará o treinamento, mas a segunda pode levar a um desempenho ligeiramente superior.



Embeddings também são úteis para representar atributos categóricos que podem assumir um grande número de valores diferentes, especialmente quando há semelhanças complexas entre os valores. Por exemplo, considere profissões, hobbies, pratos, espécies, marcas e assim por diante.

Agora você tem quase todas as ferramentas necessárias para implementar um sistema de tradução automática. Vejamos isso agora.

## Uma Rede Codificador–Decodificador para Tradução de Máquina

Daremos uma olhada em um modelo simples de tradução automática (<https://goo.gl/0g-9zWP>)<sup>10</sup> que traduzirá frases do inglês para o francês (veja a Figura 14-15).

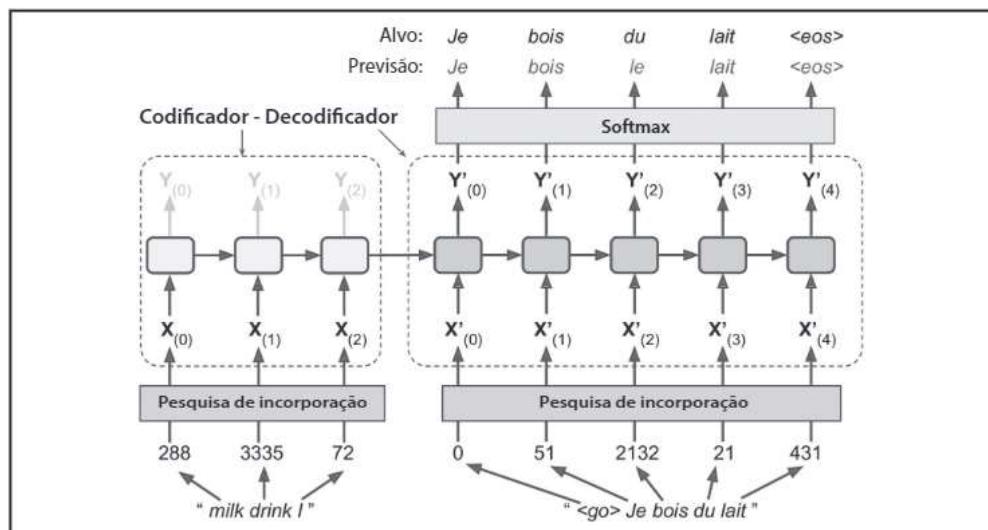


Figura 14-15. Um modelo simples de máquina de tradução

As sentenças em inglês fornecidas ao codificador e o decodificador exibem as traduções em francês. Note que as traduções francesas também são utilizadas como entradas para o decodificador, mas retrocedidas em uma etapa. Em outras palavras, o decodificador

<sup>10</sup> "Sequence to Sequence learning with Neural Networks", I. Sutskever *et al.* (2014).

é tido como a entrada da palavra que *deve* ter sua saída na etapa anterior (independentemente do que realmente exibe). É dado um token que representa o início da sentença (por exemplo, “`<go>`”) para a primeira palavra. Espera-se que o decodificador finalize a sentença com um token de final de sequência (EOS) (por exemplo, “`<eos>`”).

Observe que as sentenças em inglês são invertidas antes de serem fornecidas ao codificador. Por exemplo, “I drink milk” é revertido para “milk drink I”. Isso garante que o início da sentença em inglês seja fornecido por último ao codificador, o que é útil porque geralmente é a primeira coisa que o decodificador precisa traduzir.

Cada palavra é inicialmente representada por um identificador numérico (por exemplo, 288 para a palavra “milk”). Em seguida, uma pesquisa de embedding retorna a palavra incorporada (como explicado anteriormente, trata-se de um vetor denso de baixa dimensionalidade). Estas word embeddings são o que realmente é fornecido para o codificador e o decodificador.

A cada etapa, o decodificador exibe uma pontuação para cada palavra no vocabulário de saída (ou seja, Francês) e, então, a camada Softmax transforma essas pontuações em probabilidades. Por exemplo, na primeira etapa a palavra “Je” pode ter uma probabilidade de 20%, “Tu” pode ter uma probabilidade de 1%, e assim por diante. A palavra com a maior probabilidade é exibida, o que é muito parecido com uma tarefa de classificação regular, então utilizando a função `softmax_cross_entropy_with_logits()`, você pode treinar o modelo.

Observe que você não terá a sentença de destino para alimentar o decodificador no momento da inferência (após o treinamento). Em vez disso, conforme mostrado na Figura 14-16, alimente o decodificador com a palavra que ele emitiu na etapa anterior (isso exigirá uma pesquisa de incorporação que não é mostrada no diagrama).

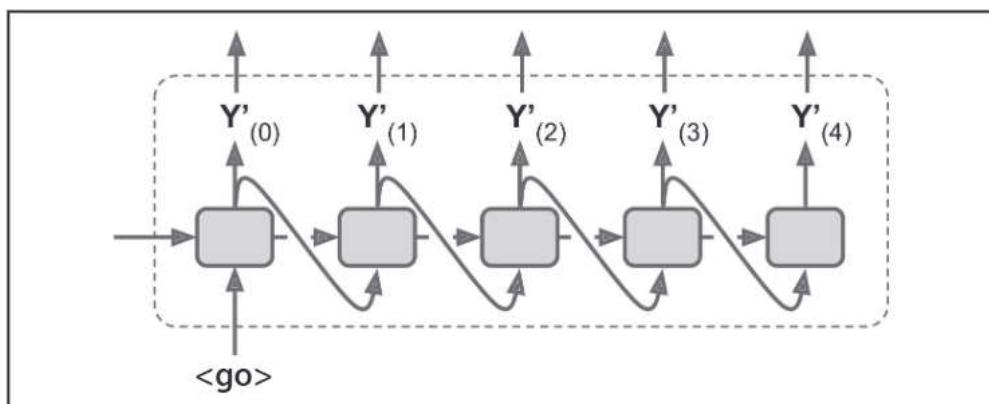


Figura 14-16. Alimentando a palavra de saída anterior como entrada no tempo de inferência

Ok, agora você já consegue ver o quadro geral. No entanto, se passar pelo tutorial do TensorFlow de sequência-para-sequência e observar o código em `rnn/translate/seq2seq_model.py` (nos modelos do TensorFlow [<https://github.com/tensorflow/models>]), você notará algumas diferenças importantes:

- Primeiro, até agora assumimos que todas as sequências de entrada (para o codificador e o decodificador) têm um comprimento constante. Mas obviamente os comprimentos das frases podem variar. Existem várias formas de lidar com isso — por exemplo, utilizar o argumento `sequence_length` para as funções `static_rnn()` ou `dynamic_rnn()` para especificar o comprimento de cada frase (como discutido anteriormente). No entanto, outra abordagem é utilizada no tutorial (presumivelmente por motivos de desempenho): as sentenças são agrupadas em intervalos de comprimento semelhante (por exemplo, um intervalo para as frases de 1 a 6 palavras, outro para as frases de 7 a 12, e assim por diante)<sup>11</sup>, e as sentenças mais curtas são preenchidas com a utilização de um token especial de preenchimento (por exemplo, “<pad>”). Por exemplo, “I drink milk” se torna “<pad> <pad> <pad> milk drink I” e sua tradução se torna “Je bois du lait <eos> <pad>”. Claro, queremos ignorar qualquer saída após o token EOS e a implementação do tutorial utiliza um vetor `target_weights` para isso. Por exemplo, os pesos seriam ajustados para `[1.0, 1.0, 1.0, 1.0, 1.0, 0.0]` para a sentença de destino “Je bois du lait <es> <pad>” (observe o peso 0.0 correspondente ao token de preenchimento na sentença de destino). Multiplicar as perdas pelos pesos-alvo zerará as perdas que correspondem a palavras passadas em tokens EOS;
- Segundo, seria terrivelmente lento produzir uma probabilidade para cada palavra possível quando o vocabulário de saída for grande (o que é o caso aqui). Se o vocabulário-alvo contém, digamos, 50 mil palavras em francês, seria muito intensivo em termos computacionais para se calcular a função softmax sobre um vetor tão grande, então o decodificador produziria vetores de 50 mil dimensões. Uma solução para evitar isso seria deixar o decodificador gerar vetores muito menores e utilizar uma técnica de amostragem para estimar a perda sem precisar calculá-la sobre cada palavra do vocabulário-alvo, como vetores de 1 mil dimensões. Esta técnica Sampled Softmax foi introduzida em 2015 por Sébastien Jean et al. (<https://goo.gl/u0GR8k>).<sup>12</sup> No TensorFlow, você pode utilizar a função `sampled_softmax_loss()`;

---

<sup>11</sup> Os tamanhos de unidades de dados utilizados no tutorial são diferentes.

<sup>12</sup> “On Using Very Large Target Vocabulary for Neural Machine Translation”, S. Jean et al. (2015).

- Em terceiro lugar, a implementação do tutorial utiliza um *mecanismo de atenção* que permite ao decodificador espiar a sequência de entrada. RNNs de atenção aumentadas estão além do escopo deste livro, mas existem artigos úteis sobre a tradução automática (<https://goo.gl/8RCous>),<sup>13</sup> leitura de máquina (<https://goo.gl/X0Nau8>)<sup>14</sup> e legendas de imagens (<https://goo.gl/xmhvfK>);<sup>15</sup>
- Finalmente, a implementação do tutorial faz uso do `tf.nn.legacy_seq2seq`, módulo que fornece ferramentas para construir facilmente vários modelos Encoder–Decoder. Por exemplo, a função `embedding_rnn_seq2seq()` cria um modelo simples Encoder–Decoder que cuida automaticamente das incorporações para você, assim como o representado na Figura 14-15. Este código poderia ser atualizado rapidamente para utilizar o novo módulo `tf.nn.seq2seq`.

Você tem agora todas as ferramentas necessárias para entender a implementação do tutorial sequência-para-sequência. Confira e treine seu próprio tradutor de inglês para francês!

## Exercícios

1. Você pode pensar em algumas aplicações para uma RNN de sequência-a-sequência? Que tal uma RNN sequência-vetor? E uma RNN vetor-a-sequência?
2. Por que as pessoas utilizam RNNs de codificador-decodificador em vez de RNNs de sequência-a-sequência simples para tradução automática?
3. Como você poderia combinar uma rede neural convolucional com uma RNN para classificar vídeos?
4. Quais são as vantagens de se construir uma RNN utilizando `dynamic_rnn()` em vez de `static_rnn()`?
5. Como você pode lidar com sequências de entrada de comprimento variável? E quanto às sequências de saída de tamanho variável?
6. Qual é uma forma comum de se distribuir treinamento e execução de uma RNN profunda em várias GPUs?
7. *Embedded Reber grammars* foram utilizadas por Hochreiter e Schmidhuber em seu artigo sobre LSTMs. São gramáticas artificiais que produzem strings como “BPBTSXXVPSEPE.” Confira a bela introdução de Jenny Orr (<https://goo.gl/7CkNRn>) sobre este tópico. Escolha uma gramática embutida específica (como

<sup>13</sup> “Neural Machine Translation by Jointly Learning to Align and Translate”, D. Bahdanau *et al.* (2014).

<sup>14</sup> “Long Short-Term Memory-Networks for Machine Reading”, J. Cheng (2016).

<sup>15</sup> “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”, K. Xu *et al.* (2015).

a representada na página de Jenny Orr) depois treine uma RNN para identificar se uma string respeita ou não essa gramática. Você primeiro precisará escrever uma função capaz de gerar um lote de treinamento contendo cerca de 50% de strings que respeitem a gramática e 50% que não.

8. Encare a competição Kaggle "How much did it rain? II" (<https://goo.gl/0DS5Xe>). Esta é uma tarefa de previsão de séries temporais: são fornecidas fotografias de valores de radares polarimétricos e é solicitada a previsão do total de precipitação pluviométrica por hora. A entrevista de Luis André Dutra e Silva (<https://goo.gl/fTA90W>) dá algumas informações interessantes sobre as técnicas que ele utilizou para alcançar o segundo lugar na competição. Em particular, ele utilizou uma RNN composta de duas camadas LSTM.
9. Acesse o tutorial Word2Vec (<https://goo.gl/edArdi>) do TensorFlow para criar uma word embedding e, em seguida, leia o tutorial Seq2Seq (<https://goo.gl/L82gvS>) para treinar um sistema de tradução do inglês para o francês.

Soluções para estes exercícios estão disponíveis no Apêndice A.

## Capítulo 15

# Autoencoders

Autoencoders são redes neurais artificiais capazes de aprender representações eficientes dos dados de entrada, conhecidos como codificações, sem supervisão alguma (ou seja, o conjunto de treinamento é não rotulado). Essas codificações geralmente têm uma dimensionalidade muito menor do que os dados de entrada, o que as torna úteis para redução de dimensionalidade (consulte o Capítulo 8). Mais importante, os autoencoders atuam como detectores de características poderosos e podem ser utilizados para pré-treinamento não supervisionado de redes neurais profundas (como discutimos no Capítulo 11). Por último, eles são capazes de gerar novos dados aleatoriamente que são muito semelhantes aos dados de treinamento, o que é chamado de *modelo gerador*. Por exemplo, você poderia treinar um autoencoder em imagens de rostos e, em seguida, ele seria capaz de gerar novos rostos.

Surpreendentemente, o funcionamento dos autoencoders se resume a simplesmente aprender a copiar suas entradas para suas saídas, o que pode soar como uma tarefa trivial, mas veremos que restringir a rede de várias maneiras pode dificultá-la bastante. Por exemplo, você pode limitar o tamanho da representação interna ou adicionar ruído às entradas e treinar a rede para recuperar as entradas originais. Essas restrições impedem que o autoencoder copie as entradas diretamente para as saídas, o que o obriga a aprender maneiras eficientes de representar os dados. Em suma, os códigos são subprodutos da tentativa do autoencoder de aprender a função de identidade, sob algumas restrições.

Neste capítulo, explicaremos com mais profundidade como funcionam os autoencoders, que tipos de restrições podem ser impostas e como implementá-las utilizando o TensorFlow, seja para redução de dimensionalidade, extração de características, pré-treinamento não supervisionado ou como modelos geradores.

## Representações Eficientes de Dados

Quais das seguintes sequências numéricas você acha mais fácil de memorizar?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

À primeira vista, parece que a primeira sequência deve ser mais fácil, pois é muito mais curta. No entanto, se você observar a segunda sequência atentamente, notará que ela segue duas regras simples: os números pares são seguidos pela sua metade e os números ímpares são seguidos pelos triplos, mais um (essa é uma sequência famosa conhecida como *números de granizo*). Ao percebermos esse padrão, a segunda sequência se torna muito mais fácil de memorizar do que a primeira, porque você só precisa memorizar as duas regras, o primeiro número e a duração da sequência. Você não se importaria muito com a existência de um padrão na segunda sequência se pudesse memorizar rapidamente sequências muito longas, apenas aprenderia cada número de cor, e seria isso. É o fato de ser difícil memorizar sequências longas que torna útil o reconhecimento de padrões e esperamos que isso esclareça por que a restrição de um autoencoder durante o treinamento o leva a descobrir e explorar padrões nos dados.

A relação entre memória, percepção e correspondência de padrões foi notoriamente estudada por William Chase e Herbert Simon no início dos anos 1970 (<https://goo.gl/kSNcX0>)<sup>1</sup>, quando observaram que os jogadores de xadrez experientes eram capazes de memorizar as posições de todas as peças em um jogo olhando para o tabuleiro por apenas 5 segundos, uma tarefa que a maioria das pessoas consideraria impossível. No entanto, este foi o caso apenas quando as peças foram posicionadas em posições realistas (de jogos reais), não quando foram posicionadas aleatoriamente. Os especialistas em xadrez não possuem uma memória muito melhor do que você e eu, eles simplesmente enxergam mais facilmente os padrões de xadrez graças à experiência com o jogo. Observar padrões os ajuda a armazenar informações de forma eficiente.

Assim como os jogadores de xadrez neste experimento de memória, um autoencoder examina as entradas, as converte em uma representação interna eficiente e, em seguida, entrega algo que (esperamos) seja muito parecido com as entradas. Um autoencoder sempre é composto de duas partes: um codificador (ou rede de reconhecimento), que converte as entradas para uma representação interna, seguido de um decodificador (ou rede geradora) que converte a representação interna para as saídas (veja a Figura 15-1).

Como você pode ver, um autoencoder geralmente tem a mesma arquitetura de um Perceptron Multicamada (MLP; veja o Capítulo 10), exceto que o número de neurônios

1 "Perception in chess", W. Chase and H. Simon (1973).

na camada de saída deve ser igual ao número de entradas. Neste exemplo, há apenas uma camada oculta composta por dois neurônios (o codificador) e uma camada de saída composta por três neurônios (o decodificador). As saídas geralmente são chamadas de *reconstruções* e a função de custo contém uma *perda de reconstrução* que penaliza o modelo quando as reconstruções são diferentes das entradas, já que o autoencoder tenta reconstruir-las.

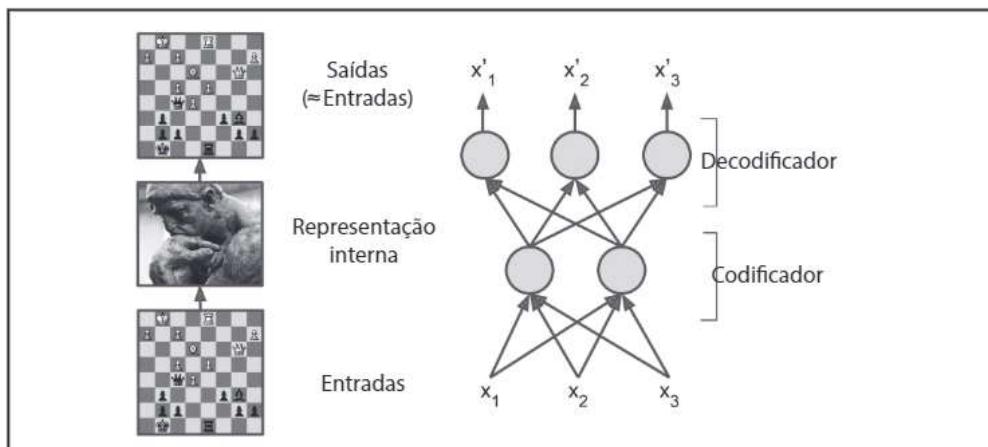


Figura 15-1. O experimento de memória do xadrez (esquerda) e um autoencoder simples (direita)

Dizem que o autoencoder é *incompleto* porque a representação interna tem uma menor dimensionalidade do que os dados de entrada (2D em vez de 3D). Um autoencoder incompleto não pode copiar trivialmente suas entradas para as codificações, mas deve encontrar uma forma de produzir uma cópia delas, portanto, ele é forçado a aprender as características mais importantes nos dados de entrada (e descartar as que não forem importantes).

Vejamos como implementar um autoencoder incompleto bem simples para a redução da dimensionalidade.

## Executando o PCA com um Autoencoder Linear Incompleto

Se o autoencoder utilizar apenas ativações lineares e a função de custo for o erro médio quadrático (MSE), então será possível mostrar que ele acaba executando a análise do componente principal (consulte o Capítulo 8).

O código a seguir cria um autoencoder linear simples para a execução do PCA em um conjunto de dados 3D, projetando-o para 2D:

```

import tensorflow as tf

n_inputs = 3 # entradas 3D
n_hidden = 2 # códigos 2D
n_outputs = n_inputs

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden)
outputs = tf.layers.dense(hidden, n_outputs)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(reconstruction_loss)

init = tf.global_variables_initializer()

```

Este código não é muito diferente de todos os MLPs que construímos nos capítulos anteriores. As duas coisas a notar são:

- O número de saídas é igual ao número de entradas;
- Para executar o PCA simples, não utilizamos nenhuma função de ativação (ou seja, todos os neurônios são lineares) e a função de custo é o MSE. Veremos autoencoders mais complexos em breve.

Agora vamos carregar o conjunto de dados, treinar o modelo no conjunto de treinamento e utilizá-lo para programar o conjunto de teste (ou seja, projetá-lo para 2D):

```

X_train, X_test = [...] # carregue o conjunto de dados

n_iterations = 1000
codings = hidden # a saída das camadas ocultas fornece o código

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train}) # sem rótulos (não supervisionado)
        codings_val = codings.eval(feed_dict={X: X_test})

```

A Figura 15-2 mostra o conjunto de dados 3D original (à esquerda) e a saída da camada oculta do autoencoder (ou seja, a camada de codificação, à direita). Como você pode ver, o autoencoder encontrou o melhor plano 2D para projetar os dados, preservando ao máximo a variação neles (assim como o PCA).

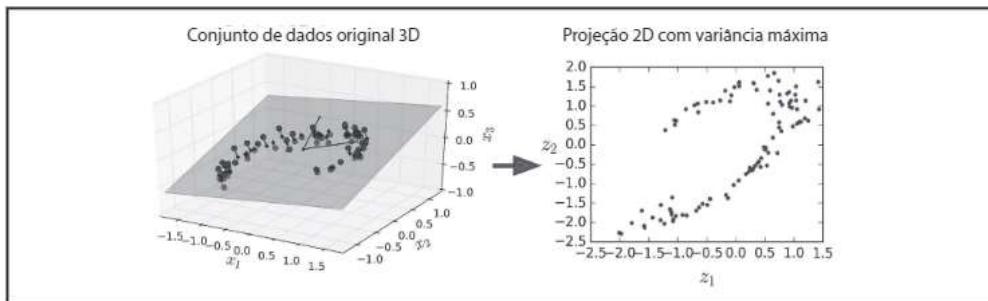


Figura 15-2. PCA realizada por um autoencoder linear incompleto

## Autoencoders Empilhados

Assim como outras redes neurais que discutimos, os autoencoders podem ter várias camadas ocultas, sendo chamados de *autoencoders empilhados* (ou *autoencoders profundos*). Adicionar mais camadas ajuda o codificador automático a aprender codificações mais complexas, mas é preciso ter cuidado para não torná-lo muito poderoso. Imagine um codificador tão poderoso que aprenda a mapear cada entrada para um único número arbitrário (e o decodificador aprende o mapeamento reverso). Obviamente, tal autoencoder reconstruirá os dados de treinamento perfeitamente, mas não terá aprendido nenhuma representação útil de dados no processo (e é improvável que generalize bem para novas instâncias).

A arquitetura de um autoencoder empilhado é simétrica em relação à camada oculta central (a camada de codificação), simplificando: parece um sanduíche. Por exemplo, um autoencoder do MNIST (introduzido no Capítulo 3) pode ter 784 entradas, seguidas por uma camada oculta com 300 neurônios, depois uma camada oculta central de 150 neurônios e, então, outra camada oculta com 300 neurônios e uma camada de saída com 784. Este autoencoder empilhado está representado na Figura 15-3.

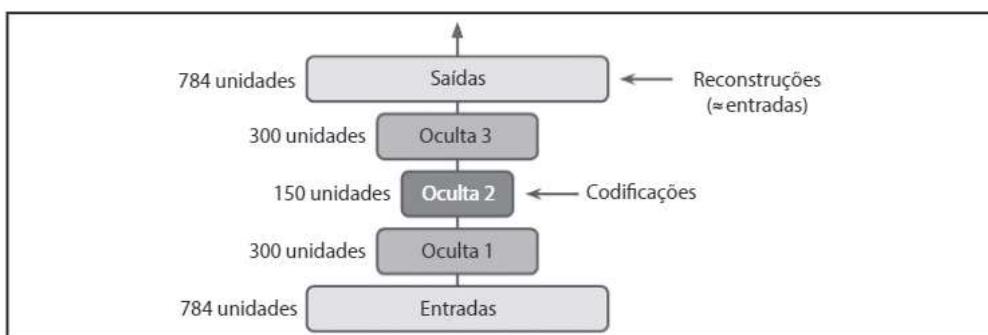


Figura 15-3. Autoencoder Empilhado

## Implementação do TensorFlow

É possível implementar um autoencoder empilhado muito parecido com um MLP profundo normal aplicando as mesmas técnicas que utilizamos no Capítulo 11 para o treinamento de redes profundas. Por exemplo, o código a seguir com a utilização da inicialização He, a função de ativação ELU e a regularização  $\ell_2$  cria um autoencoder empilhado para o MNIST. O código deve parecer bem familiar, porém não existem labels (sem  $y$ ):

```
from functools import partial

n_inputs = 28 * 28 # para MNIST
n_hidden1 = 300
n_hidden2 = 150 # códigos
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.0001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

he_init = tf.contrib.layers.variance_scaling_initializer()
l2_regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
my_dense_layer = partial(tf.layers.dense,
                        activation=tf.nn.elu,
                        kernel_initializer=he_init,
                        kernel_regularizer=l2_regularizer)

hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2) # códigos
hidden3 = my_dense_layer(hidden2, n_hidden3)
outputs = my_dense_layer(hidden3, n_outputs, activation=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Você pode, então, treinar o modelo normalmente. Observe que os rótulos dos dígitos ( $y_{batch}$ ) não são utilizados:

```
n_epochs = 5
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
```

```

n_batches = mnist.train.num_examples // batch_size
for iteration in range(n_batches):
    X_batch, y_batch = mnist.train.next_batch(batch_size)
    sess.run(training_op, feed_dict={X: X_batch})

```

## Amarrando Pesos

Uma técnica comum é *amarrar os pesos* das camadas do decodificador aos pesos das camadas do codificador quando um autoencoder for nitidamente simétrico, como o que construímos, o que diminuirá pela metade o número de pesos no modelo, acelerando o treinamento e limitando o risco de sobreajuste. Especificamente, se o autoencoder tiver um total de  $N$  camadas (sem contar a camada de entrada) e  $W_L$  representa os pesos de conexão da  $L$ -ésima camada (por exemplo, a camada 1 é a primeira camada oculta, camada  $\frac{N}{2}$  é a camada de codificação e a camada  $N$  é a camada de saída), então os pesos da camada de decodificação podem ser definidos simplesmente como:  $W_{N-L+1} = W_L^T$  (com  $L = 1, 2, \dots, \frac{N}{2}$ ).

Infelizmente, é um pouco incômodo implementar pesos amarrados no TensorFlow com a utilização da função `dense()`; na verdade, é mais fácil definir apenas as camadas manualmente. O código fica significativamente mais detalhado:

```

activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3") # pesos amarrados
weights4 = tf.transpose(weights1, name="weights4") # pesos amarrados

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

optimizer = tf.train.AdamOptimizer(learning_rate)

```

```
training_op = optimizer.minimize(loss)
```

```
init = tf.global_variables_initializer()
```

Este código é bastante simples, mas há algumas coisas importantes a serem observadas:

- Primeiro, `weights3` e `weights4` não são variáveis, são a transposição de `weights2` e `weights1` (eles estão “amarrados” a eles), respectivamente;
- Segundo, como não são variáveis, não adianta regularizá-las: só regularizamos `weights1` e `weights2`;
- Terceiro, vieses nunca são amarrados, e nunca regularizados.

## Treinando um Autoencoder por Vez

Em vez de treinar todo o autoencoder empilhado de uma só vez, como fizemos, é muito mais rápido treinar um autoencoder raso de cada vez e depois empilhá-los em um único autoencoder empilhado (daí o nome), como mostrado na Figura 15-4. Isto é especialmente útil para autoencoders muito profundos.

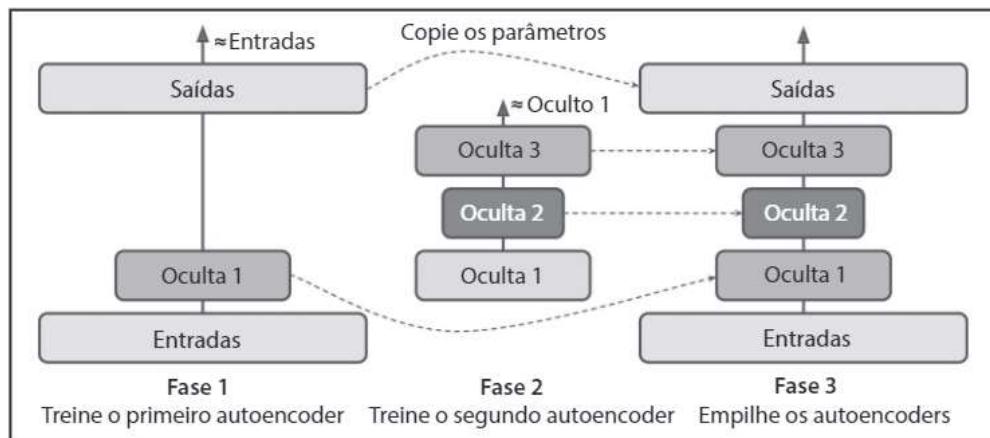


Figura 15-4. Treinando um autoencoder por vez

Durante a primeira fase do treinamento, o primeiro autoencoder aprende a reconstruir as entradas, já na fase seguinte, o segundo autoencoder aprende a reconstruir a saída da camada oculta do primeiro autoencoder. Finalmente, como mostrado na Figura 15-4, você constrói um grande sanduíche utilizando todos esses autoencoders (ou seja, você primeiro empilha as camadas ocultas de cada autoencoder e, em seguida, as camadas de saída na ordem inversa), o que lhe fornecerá o autoencoder final empilhado. Criando um autoencoder empilhado muito profundo, você poderia facilmente treinar mais autoencoders dessa maneira.

A abordagem mais simples para implementar este algoritmo de treinamento multifásico seria a utilização de um grafo diferente do TensorFlow para cada fase. Depois de treinar um autoencoder, basta executar o conjunto de treinamento e capturar a saída da camada oculta, que serve como o conjunto de treinamento para o próximo autoencoder. Após todos os autoencoders terem sido treinados dessa maneira, você copiará os pesos e os vieses de cada um e os utilizará para criar o autoencoder empilhado. A implementação desta abordagem é bem simples, por isso não a detalhamos aqui, mas, para ver um exemplo, confira o código nos notebooks do Jupyter (<https://github.com/ageron/handson-ml>).

Outra abordagem, como mostrado na Figura 15-5, seria utilizar um único grafo contendo o autoencoder todo empilhado, além de algumas operações extras para executar cada fase de treinamento.

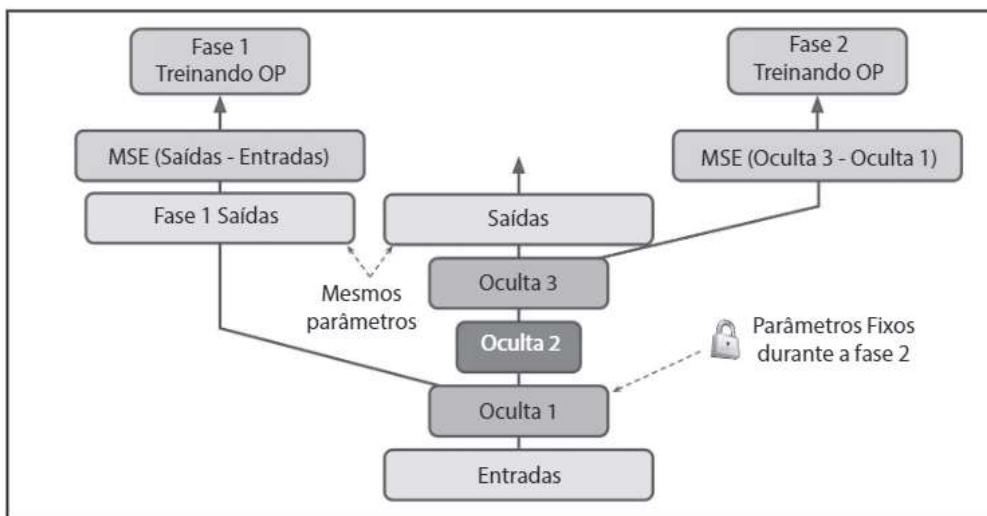


Figura 15-5. Um único grafo para treinar um autoencoder empilhado

Isto merece uma explicação:

- A coluna central no grafo é o autoencoder totalmente empilhado, parte que pode ser utilizada após o treinamento;
- A coluna da esquerda é o conjunto de operações necessário para executar a primeira fase do treinamento. Ela cria uma camada de saída que ultrapassa as camadas ocultas 2 e 3, e compartilha os mesmos pesos e vieses da camada de saída do autoencoder empilhado. Além disso, elas são as operações de treinamento que visam aproximar ao máximo a saída das entradas. Assim, esta fase treinará os pesos e vieses para a camada oculta 1 e a camada de saída (ou seja, o primeiro autoencoder);

- A coluna da direita no grafo é o conjunto de operações necessário para executar a segunda fase do treinamento. Ela adiciona a operação de treinamento que terá como objetivo de aproximar ao máximo a saída da camada oculta 3 da saída da camada oculta 1. Observe que devemos congelar a camada oculta 1 durante a fase 2, que treinará os pesos e vieses para as camadas ocultas 2 e 3 (ou seja, o segundo autoencoder).

O código do TensorFlow é assim:

```
[...] # Construa todo o autoencoder empilhado normalmente
      # Neste exemplo, os pesos não estão amarrados

optimizer = tf.train.AdamOptimizer(learning_rate)

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss = tf.reduce_mean(tf.square(phase1_outputs - X))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss = tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op = optimizer.minimize(phase2_loss, var_list=train_vars)
```

A primeira fase é bastante direta: apenas criamos uma camada de saída que pula as camadas ocultas 2 e 3, depois construímos as operações de treinamento para minimizar a distância entre as saídas e as entradas (mais alguma regularização).

A segunda fase apenas adiciona as operações necessárias para minimizar a distância entre a saída da camada oculta 3 e a camada oculta 1 (também com alguma regularização). Mais importante, deixando de fora `weights1` e `biases1`, fornecemos a lista de variáveis treináveis para o método `minimize()`; efetivamente congelando a camada oculta 1 durante a fase 2.

Durante a fase de execução, você só precisa executar o treinamento da fase 1 para um número de épocas, depois o treinamento de fase 2 para mais algumas épocas.



Como a camada oculta 1 é congelada durante a fase 2, sua saída sempre será a mesma para qualquer instância de treinamento. Para evitar recalcular a saída da camada oculta 1 em cada época, você pode calculá-la para todo o conjunto de treinamento no final da fase 1 e, em seguida, alimentar diretamente a saída em cache da camada oculta 1 durante a fase 2, resultando em um bom aumento de desempenho.

## Visualizando as Reconstruções

Comparar as entradas e as saídas é uma forma de garantir que um autoencoder seja adequadamente treinado. Elas devem ser bem semelhantes e as diferenças devem ser detalhes sem importância. Plotaremos dois dígitos aleatórios e suas reconstruções:

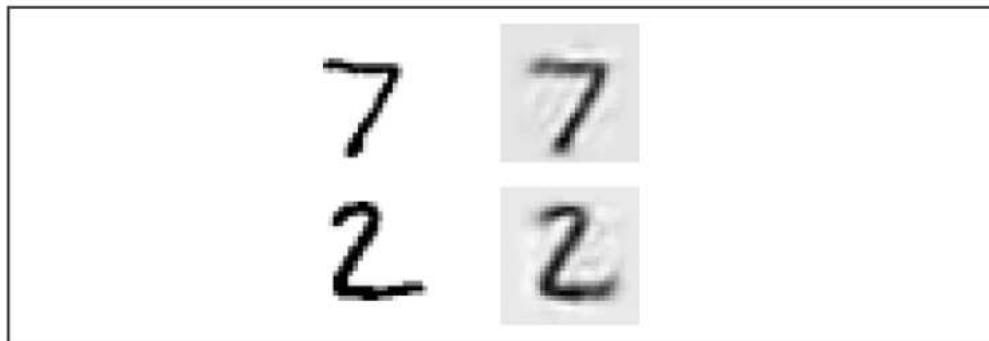
```
n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
    [...] # Treine o Autoencoder
    outputs_val = outputs.eval(feed_dict={X: X_test})

def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys", interpolation="nearest")
    plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])
```

A Figura 15-6 mostra as imagens resultantes.



*Figura 15-6. Dígitos Originais (esquerda) e suas reconstruções (direita)*

Parece perto o suficiente. Portanto, o autoencoder aprendeu adequadamente a reproduzir suas entradas, mas será que aprendeu características úteis? Vejamos.

## Visualizando as Características

Uma vez que seu autoencoder tenha aprendido algumas características, convém observá-las e existem várias técnicas para isso. Indiscutivelmente, mais simples delas é considerar cada neurônio em cada camada oculta e encontrar as instâncias de treinamento que mais o ativam, o que é especialmente útil para as camadas ocultas superiores, pois elas geral-

mente capturam características relativamente grandes que você identifica facilmente em um grupo de instâncias de treinamento que as contêm. Por exemplo, se um neurônio se ativar quando vir um gato em uma foto, será bastante óbvio que as imagens que mais o ativam contêm gatos. No entanto, para as camadas inferiores, essa técnica não funciona tão bem, pois as características são menores e mais abstratas. Por isso, muitas vezes é difícil entender exatamente o porquê de o neurônio estar se animando.

Veremos outra técnica. Para cada neurônio na primeira camada oculta é possível criar uma imagem cuja intensidade de um pixel corresponde ao peso da conexão com o mesmo neurônio. Por exemplo, o código a seguir mapeia as características aprendidas por cinco neurônios na primeira camada oculta:

```
with tf.Session() as sess:  
    [...] # treine o autoencoder  
    weights1_val = weights1.eval()  
  
    for i in range(5):  
        plt.subplot(1, 5, i + 1)  
        plot_image(weights1_val.T[i])
```

Você pode obter características de baixo nível, como as mostradas na Figura 15-7.

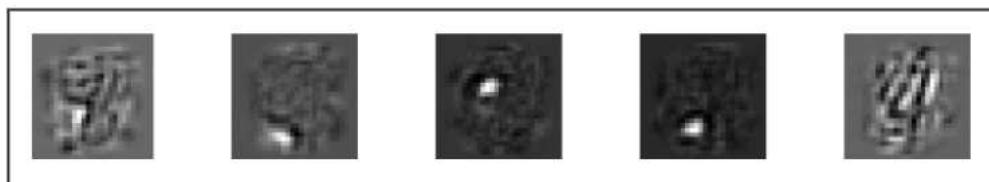


Figura 15-7. Características aprendidas por cinco neurônios da primeira camada oculta

As primeiras quatro características parecem corresponder a pequenos patches, enquanto a quinta característica parece procurar por traços verticais (observe que essas características vêm de autoencoders de remoção de ruídos empilhados que discutiremos mais adiante).

Outra técnica seria alimentar o autoencoder com uma imagem aleatória de entrada, medir a ativação do neurônio em que você está interessado e, em seguida, realizar a retropropagação para ajustar a imagem de tal forma que o neurônio seja ativado ainda mais. A imagem gradualmente se transformará na imagem mais excitante (para o neurônio) se você iterar várias vezes (realizando a subida do gradiente), uma técnica útil para visualização dos tipos de entradas buscadas por um neurônio.

Finalmente, se você estiver utilizando um autoencoder para executar pré-treinamento não supervisionado, por exemplo, para uma tarefa de classificação, medir o desempenho do classificador é uma forma simples de verificar se as características aprendidas pelo autoencoder são úteis.

## Pré-treinamento Não Supervisionado Utilizando Autoencoders Empilhados

Como discutimos no Capítulo 11, se você estiver lidando com uma tarefa supervisionada complexa, mas não tiver muitos dados rotulados de treinamento, uma solução seria encontrar uma rede neural que execute uma tarefa semelhante e, em seguida, reutilizar suas camadas inferiores. Isto possibilita treinar um modelo de alto desempenho utilizando apenas pequenos dados de treinamento porque sua rede neural não precisa aprender todas as características de baixo nível, apenas reutilizará os detectores das características aprendidos pela rede existente.

Da mesma forma, se você tiver um grande conjunto de dados, mas com a maioria sem rótulos, você pode primeiro treinar um autoencoder empilhado e, então, reutilizar as camadas inferiores para criar uma rede neural para sua tarefa atual e treiná-la com a atualização dos dados rotulados. Por exemplo, a Figura 15-8 mostra como utilizar um autoencoder empilhado para executar pré-treinamento não supervisionado para uma rede neural de classificação. Conforme discutido anteriormente, o próprio autoencoder empilhado é treinado em um autoencoder de cada vez. Se você realmente não tiver muitos dados de treinamento rotulados ao treinar o classificador, convém congelar as camadas pré-treinadas (pelo menos as inferiores).

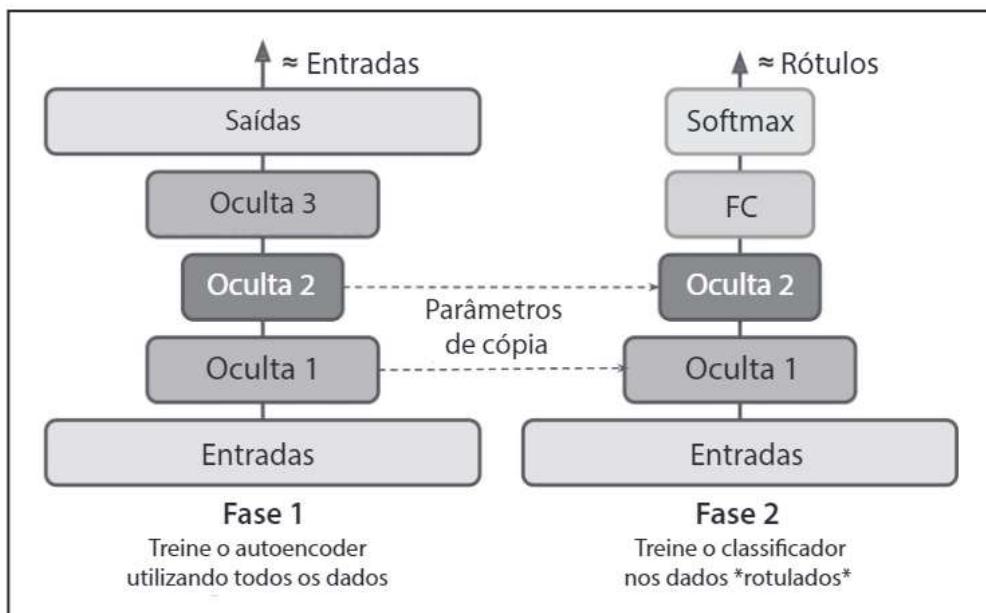


Figura 15-8. Pré-treinamento não supervisionado utilizando autoencoders



Essa situação é bastante comum porque a construção de um grande conjunto de dados não rotulados geralmente é barata (por exemplo, um simples script pode baixar milhões de imagens da internet), mas rotulá-las só poderia ser feito de forma confiável por humanos (por exemplo, classificar imagens como bonitas ou não). As instâncias de rotulagem consomem muito tempo e são caras, por isso é muito comum ter apenas algumas milhares de instâncias rotuladas.

Como discutimos anteriormente, a descoberta feita em 2006 por Geoffrey Hinton *et al.* de que as redes neurais profundas podem ser pré-treinadas de maneira não supervisionada é um dos desencadeadores do atual tsunami do Aprendizado Profundo. Para isso, eles utilizaram máquinas restritas de Boltzmann (consulte o apêndice E), mas, em 2007, Yoshua Bengio *et al.* (<https://goo.gl/R5L7HJ>)<sup>2</sup> mostraram que os autoencoders também funcionavam nesse caso.

Não há nada de especial sobre a implementação do TensorFlow: para criar uma nova rede neural, apenas treine um autoencoder utilizando todos os dados de treinamento e reutilize suas camadas do codificador (consulte o Capítulo 11 para obter mais detalhes sobre como reutilizar camadas pré-treinadas ou consulte os exemplos de código nos notebooks do Jupyter).

Até agora, limitamos o tamanho da camada de codificação tornando-a incompleta para forçar o autoencoder a aprender características interessantes. Na verdade, existem muitos outros tipos de restrições que podem ser utilizados, incluindo aquelas que permitem que a camada de codificação seja do tamanho das entradas, ou até maiores, resultando em um *autoencoder supercompleto*. Veremos algumas dessas abordagens agora.

## Autoencoders de Remoção de Ruídos

Outra maneira de forçar o autoencoder a aprender características úteis é adicionar ruído às suas entradas, treinando-o para recuperar as entradas originais sem ruído, impedindo que o autoencoder copie suas entradas para suas saídas e acabe forçado a encontrar padrões nos dados.

A ideia de utilizar autoencoders para remover o ruído existe desde os anos 1980 (por exemplo, é mencionada na tese de mestrado de Yann LeCun em 1987). Em um artigo de 2008 (<https://goo.gl/K9pqcx>),<sup>3</sup> Pascal Vincent *et al.* demonstraram que autoencoders também podiam ser utilizados para a extração de características. Em um artigo de

2 “Greedy Layer-Wise Training of Deep Networks,” Y. Bengio *et al.* (2007).

3 “Extracting and Composing Robust Features with Denoising Autoencoders”, P. Vincent *et al.* (2008)\*

2010 (<https://goo.gl/HgCDIA>),<sup>4</sup> Vincent *et al.* introduziram os *autoencoders de remoção de barulho empilhados*.

O ruído pode ser puro ruído Gaussiano adicionado às entradas ou pode ser desligado aleatoriamente, assim como no dropout (introduzido no Capítulo 11). A Figura 15-9 mostra as duas opções.

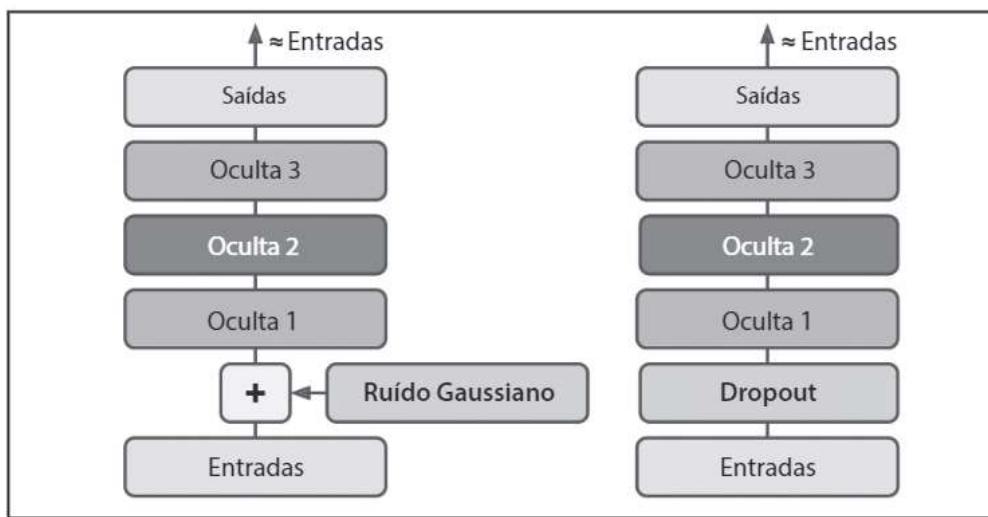


Figura 15-9. Autoencoders de Remoção de Ruído, com ruído Gaussiano (esquerda) ou dropout (direita)

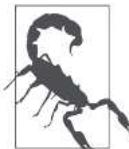
## Implementando o TensorFlow

A implementação de autoencoders de remoção de ruídos no TensorFlow não é muito difícil e começaremos com o ruído gaussiano. É como treinar um autoencoder regular, mas você adiciona ruído às entradas e a perda de reconstrução é calculada com base nas entradas originais:

```
noise_level = 1.0
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + noise_level * tf.random_normal(tf.shape(X))

hidden1 = tf.layers.dense(X_noisy, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

<sup>4</sup> “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”, P. Vincent *et al.* (2010).



Como a forma de  $X$  é apenas parcialmente definida durante a fase de construção, não podemos saber antecipadamente a forma do ruído que devemos adicionar a  $X$ . Nós não podemos recorrer a `X.get_shape()` porque isso apenas retornaria a forma parcialmente definida de  $X$  (`[None, n_inputs]`), e `random_normal()` espera uma forma totalmente definida para que possa gerar uma exceção. Em vez disso, chamamos `tf.shape(X)`, que cria uma operação que retornará a forma de  $X$  em tempo de execução, que será totalmente definida nesse ponto.

Não é difícil implementar a versão dropout, que é a mais comum:

```
dropout_rate = 0.3

training = tf.placeholder_with_default(False, shape=(), name='training')

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                         name="hidden1")
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

Durante o treinamento, devemos configurar `training` para `True` (como explicado no Capítulo 11) utilizando o `feed_dict`:

```
sess.run(training_op, feed_dict={X: X_batch, training: True})
```

Durante o teste, não é necessário configurar `training` para `False`, já que definimos isto como padrão na chamada para a função `placeholder_with_default()`.

## Autoencoders Esparsos

Outro tipo de restrição que muitas vezes leva a uma boa extração de características é a *esparsidade*: ao adicionar um termo apropriado à função de custo, o *autoencoder* é pressionado a reduzir o número de neurônios ativos na camada de codificação. Por exemplo, ele pode ser forçado a ter, em média, apenas 5% de neurônios significativamente ativos na camada de codificação, forçando o autoencoder a representar cada entrada como uma combinação de um pequeno número de ativações. Como resultado, cada neurônio na camada de codificação acaba representando uma característica útil (se você só pudesse falar algumas palavras por mês, provavelmente tentaria fazê-las dignas de serem ouvidas).

Devemos primeiro medir a dispersão real da camada de codificação em cada iteração de treinamento para favorecer modelos esparsos, o que pode ser feito por meio do cálculo da ativação média de cada neurônio na camada de codificação durante todo o lote de treinamento cujo tamanho não deve ser muito pequeno ou a média não será precisa.

Uma vez que temos a ativação média por neurônio, queremos penalizar aqueles que estão muito ativos com a adição de uma *perda de esparsidade* à função de custo. Por exemplo, se medirmos que um neurônio tem uma ativação média de 0,3, mas a esparsidade alvo é 0,1, ele deve ser penalizado para ativar menos. Uma abordagem poderia ser adicionar o erro ao quadrado  $(0,3 - 0,1)^2$  à função de custo, mas, na prática, como é possível ver na Figura 15-10, uma abordagem melhor seria utilizar a divergência de *Kullback-Leibler* (brevemente discutida no Capítulo 4) que tem gradientes muito mais fortes do que o erro médio quadrático.

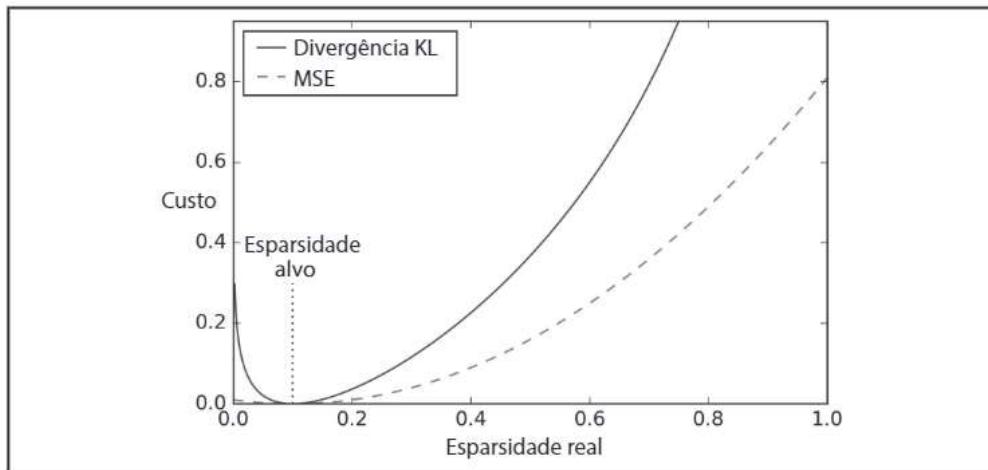


Figura 15-10. Perda de esparsidade

Dadas duas distribuições de probabilidade discretas  $P$  e  $Q$  e utilizando a Equação 15-1 a divergência KL entre essas distribuições, descrita  $D_{KL}(P \parallel Q)$ , pode ser calculada.

*Equação 15-1. Divergência Kullback–Leibler*

$$D_{KL}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

No nosso caso, queremos medir a divergência entre a probabilidade alvo  $p$  de que um neurônio ativará na camada de codificação e a probabilidade real  $q$  (isto é, a ativação média sobre o lote de treinamento). Então a divergência KL simplifica para a Equação 15-2.

*Equação 15-2. Divergência KL entre a esparsidade alvo p e a esparsidade real q*

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Depois de calcularmos a perda de esparsidade para cada neurônio na camada de codificação, resumimos essas perdas e adicionamos o resultado à função de custo. A fim de controlar a importância relativa da perda de esparsidade e da perda de reconstrução, podemos multiplicar a perda da esparsidade por um hiperparâmetro de peso da esparsidade. Se esse peso for muito alto, o modelo permanecerá próximo à esparsidade do alvo, mas poderá não reconstruir as entradas corretamente, tornando o modelo inútil. Por outro lado, se for muito baixo, o modelo ignorará principalmente o objetivo da esparsidade e não aprenderá nenhuma característica interessante.

## Implementando no TensorFlow

Agora, temos tudo que precisamos para implementar um autoencoder esparso utilizando o TensorFlow:

```
def kl_divergence(p, q):
    return p * tf.log(p / q) + (1 - p) * tf.log((1 - p) / (1 - q))

learning_rate = 0.01
sparsity_target = 0.1
sparsity_weight = 0.2

[...] # Construa um autoencoder normal (neste exemplo, a camada de código está escondida)

hidden1_mean = tf.reduce_mean(hidden1, axis=0) # batch mean
sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target, hidden1_mean))
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
loss = reconstruction_loss + sparsity_weight * sparsity_loss
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)
```

Um detalhe importante é o fato de que as ativações da camada de codificação devem estar entre 0 e 1 (mas não iguais a 0 ou 1) ou, então, a divergência KL retornará NaN (*Not a Number*). Uma solução simples é utilizar a função de ativação logística para a camada de codificação:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.sigmoid)
```

Um truque simples pode acelerar a convergência: em vez de usar o MSE, podemos escolher uma perda de reconstrução que terá gradientes maiores. A entropia cruzada geralmente é uma boa escolha e, para utilizá-la, devemos normalizar as entradas para fazê-las assumir valores de 0 a 1 e utilizar a função de ativação logística na camada de saída para que as elas também assumam valores de 0 a 1. A função do TensorFlow

`sigmoid_cross_entropy_with_logits()` cuida de aplicar eficientemente a função de ativação logística (sigmoide) às saídas e calcular a entropia cruzada:

```
[...]
logits = tf.layers.dense(hidden1, n_outputs)
outputs = tf.nn.sigmoid(logits)

xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits)
reconstruction_loss = tf.reduce_mean(xentropy)
```

Observe que a operação `outputs` não é necessária durante o treinamento (só a usamos quando queremos examinar as reconstruções).

## Autoencoders Variacionais

Outra importante categoria de autoencoders foi introduzida em 2014 (<https://goo.gl/NZq7r2>) por Diederik Kingma e Max Welling<sup>5</sup> e rapidamente se tornou um dos tipos mais populares: *autoencoders variacionais*.

Eles são bem diferentes de todos os autoencoders que discutimos até agora, em particular:

- São *autoencoders probabilísticos*, o que significa que suas saídas são parcialmente determinadas pelo acaso, mesmo após o treinamento (ao contrário dos autoencoders de remoção de ruídos que utilizam a aleatoriedade apenas durante o treinamento);
- Mais importante, eles são *autoencoders geradores*, o que significa que podem gerar novas instâncias que parecem ter sido amostradas a partir do conjunto de treinamento.

Ambas as propriedades os tornam bastante semelhantes às RBMs (consulte o Apêndice E), mas é fácil treiná-los e o processo de amostragem é muito mais rápido (com os RBMs você precisa esperar que a rede se estabilize em um “equilíbrio térmico” antes de poder amostrar uma nova instância).

Daremos uma olhada em como eles funcionam. A Figura 15-11 (esquerda) mostra um autoencoder variacional. Você pode reconhecer, é claro, a estrutura básica de todos os autoencoders, com um codificador seguido por um decodificador (neste exemplo, ambos têm duas camadas ocultas), mas há uma alteração nesse caso: em vez de produzir diretamente uma codificação para uma dada entrada, o codificador produz uma codificação média  $\mu$  e um desvio padrão  $\sigma$ . A codificação atual é, então, amostrada aleatoriamente a

---

<sup>5</sup> “Auto-Encoding Variation”, Bayes, D. Kingma and M. Welling (2014).

partir de uma distribuição Gaussiana, com média  $\mu$  e um desvio padrão  $\sigma$ . Depois disso, o decodificador apenas decodifica normalmente a codificação da amostra. A parte direita do diagrama mostra uma instância de treinamento passando por esse autoencoder. Primeiro, o codificador produz  $\mu$  e  $\sigma$ , então uma codificação é amostrada aleatoriamente (observe que ela não está exatamente localizada em  $\mu$ ) e, finalmente, essa codificação é decodificada e a saída final se assemelha à instância de treinamento.

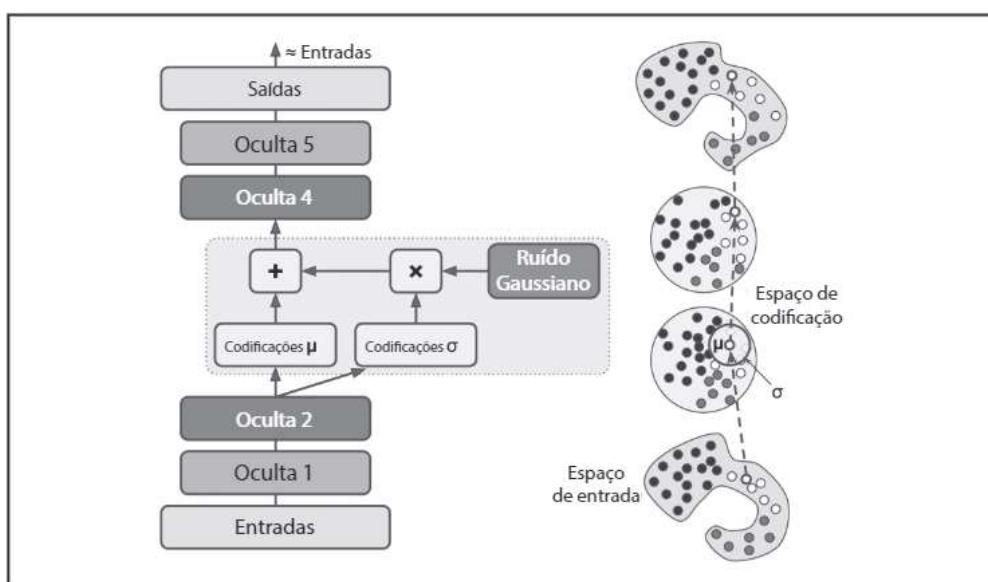


Figura 15-11. Autoencoder Variacional (esquerda) e uma instância passando por ele (direita)

Como pode ser visto no diagrama, embora as entradas possam ter uma distribuição muito complicada, um autoencoder variacional tende a produzir codificações que parecem ter sido amostradas a partir de uma distribuição Gaussiana simples:<sup>6</sup> a função de custo (discutida a seguir) empurra as codificações para migrar gradualmente dentro do espaço de codificação durante o treinamento (também chamado de *espaço latente*), para ocupar uma região aproximadamente (hiper) esférica que se parece com uma nuvem de pontos gaussianos. Uma grande consequência é que, após o treinamento de um autoencoder variacional, é possível gerar facilmente uma nova instância: basta amostrar uma codificação aleatória da distribuição gaussiana, decodificá-la e, *voilà!*

Então, observaremos a função de custo, que é composta de duas partes. A primeira é a perda de reconstrução usual que empurra o autoencoder para reproduzir suas entradas (podemos utilizar entropia cruzada para isso, conforme discutido anteriormente). A segunda é a *perda*

<sup>6</sup> Os *autoencoders* variacionais são, na verdade, mais gerais; as codificações não estão limitadas a distribuições gaussianas.

*latente* que faz com que o autoencoder tenha codificações que parecem ter sido amostradas a partir de uma distribuição Gaussiana simples, para a qual utilizamos a divergência KL entre a distribuição de destino (a distribuição Gaussiana) e a distribuição real das codificações. A matemática é um pouco mais complexa do que antes, principalmente por causa do ruído Gaussiano que limita a quantidade de informação que pode ser transmitida à camada de codificação (empurrando assim o autoencoder para aprender características úteis). Felizmente, as equações simplificam o código a seguir para a perda latente:<sup>7</sup>

```
eps = 1e-10 # suavizar termo para evitar computar log(0) que é NaN
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

Treinar o codificador para exibir  $\gamma = \log(\sigma^2)$  em vez de  $\sigma$  é uma variante comum. Onde quer que precisemos de  $\sigma$ , podemos apenas calcular  $\sigma = \exp\left(\frac{\gamma}{2}\right)$ , facilitando a captura de sigmas de diferentes escalas pelo codificador, e assim ajudando a acelerar a convergência. A perda latente é um pouco mais simples:

```
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
```

O código a seguir constrói o autoencoder variacional mostrado na Figura 15-11 (esquerda) utilizando a variante  $\log(\sigma^2)$ :

```
from functools import partial

n_inputs = 28 * 28
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20 # codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs
learning_rate = 0.001

initializer = tf.contrib.layers.variance_scaling_initializer()
my_dense_layer = partial(
    tf.layers.dense,
    activation=tf.nn.elu,
    kernel_initializer=initializer)

X = tf.placeholder(tf.float32, [None, n_inputs])
hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2)
hidden3_mean = my_dense_layer(hidden2, n_hidden3, activation=None)
hidden3_gamma = my_dense_layer(hidden2, n_hidden3, activation=None)
noise = tf.random_normal(tf.shape(hidden3_gamma), dtype=tf.float32)
hidden3 = hidden3_mean + tf.exp(0.5 * hidden3_gamma) * noise
```

---

<sup>7</sup> Para mais detalhes matemáticos, confira o artigo original sobre autoencoders variacionais, ou o ótimo tutorial de Carl Doersch (<https://goo.gl/ViiAzQ>) (2016).

```

hidden4 = my_dense_layer(hidden3, n_hidden4)
hidden5 = my_dense_layer(hidden4, n_hidden5)
logits = my_dense_layer(hidden5, n_outputs, activation=None)
outputs = tf.sigmoid(logits)
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits)
reconstruction_loss = tf.reduce_sum(xentropy)
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
loss = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

## Gerando Dígitos

Agora utilizaremos esse autoencoder variacional para gerar imagens que parecem com dígitos manuscritos. Precisamos apenas treinar o modelo, amostrar códigos aleatórios de uma distribuição Gaussiana e decodificá-los.

```

import numpy as np

n_digits = 60
n_epochs = 50
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

    codings_rnd = np.random.normal(size=[n_digits, n_hidden3])
    outputs_val = outputs.eval(feed_dict={hidden3: codings_rnd})

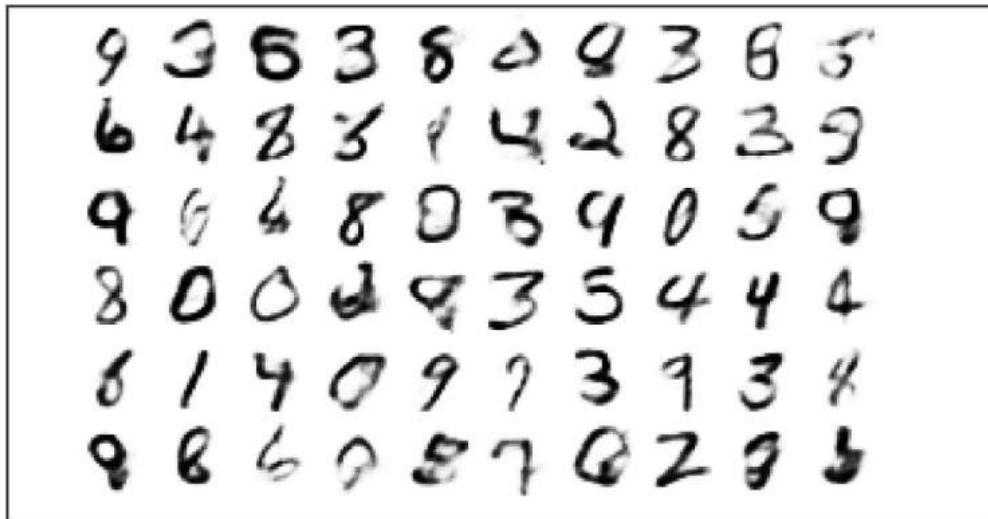
```

É isso aí. Agora podemos ver como são os dígitos “manuscritos” produzidos pelo autoencoder (veja a Figura 15-12):

```

for iteration in range(n_digits):
    plt.subplot(n_digits, 10, iteration + 1)
    plot_image(outputs_val[iteration])

```



*Figura 15-12. Imagens de dígitos manuscritos gerados pelo autoencoder variacional*

Enquanto alguns são bastante “criativos”, a maioria desses dígitos parece bastante convincente. Mas não seja muito duro com o autoencoder, pois ele começou a aprender menos de uma hora atrás. Dê um pouco mais de tempo de treinamento a ele e esses dígitos ficarão cada vez melhores.

## Outros Autoencoders

O incrível sucesso do aprendizado supervisionado ofuscou o aprendizado não supervisionado em reconhecimento de imagem, reconhecimento de voz, tradução de texto e muito mais, mas, na verdade, ele ainda está crescendo. Novas arquiteturas para autoencoders e outros algoritmos de aprendizado não supervisionados são inventadas regularmente, tanto que não conseguiremos abordar todas neste livro. Segue uma breve (não exaustiva) visão geral de mais alguns tipos de autoencoders que você talvez queira verificar:

*Contractive autoencoder (CAE) (<https://goo.gl/U5t9Ux>)<sup>8</sup>*

O autoencoder é restrinido durante o treinamento para que as derivadas das codificações com relação às entradas sejam pequenas. Em outras palavras, duas entradas semelhantes devem ter codificações semelhantes.

---

<sup>8</sup> “Contractive Auto-Encoders: Explicit Invariance During ”feature Extraction, S. Rifai *et al.* (2011).

*Stacked convolutional autoencoders (https://goo.gl/PTwsol)<sup>9</sup>*

Autoencoders que aprendem a extrair recursos visuais reconstruindo imagens processadas por meio de camadas convolucionais.

*Generative stochastic network (GSN) (https://goo.gl/HjON1m)<sup>10</sup>*

Uma generalização de autoencoders de remoção de ruídos, com a capacidade adicional de gerar dados.

*Winner-take-all (WTA) autoencoder (https://goo.gl/I1LvzL)<sup>11</sup>*

Após calcular as ativações de todos os neurônios na camada de codificação durante o treinamento, são preservadas apenas as ativações de  $k\%$  superiores para cada neurônio sobre o lote de treinamento, e o restante é definido como zero, naturalmente, levando a codificações esparsas. Além disso, pode ser utilizada uma abordagem WTA semelhante para produzir autoencoders convolucionais esparsos.

*Generative Adversarial Network (GAN) (https://goo.gl/qd4Rhn)<sup>12</sup>*

Uma rede chamada “discriminadora” é treinada para distinguir dados reais de dados falsos produzidos por uma segunda rede chamada “geradora”. A geradora aprende a enganar a discriminadora, enquanto a discriminadora aprende a evitar os truques da geradora. Esta competição leva a dados falsificados cada vez mais realistas e codificações muito robustas. Treinamento adversário é uma ideia muito poderosa que vem ganhando impulso. Yann Lecun chegou a chamá-lo de “a coisa mais legal desde o pão fatiado”.

## Exercícios

1. Quais são as principais tarefas realizadas pelos autoencoders?
2. Suponha que você queira treinar um classificador e tenha muitos dados de treinamento não rotulados e apenas algumas milhares de instâncias rotuladas. Como os autoencoders podem ajudar? Como você procederia?
3. Se um autoencoder reconstrói perfeitamente as entradas, ele é necessariamente um bom autoencoder? Como você pode avaliar o desempenho de um autoencoder?

9 “Stacked Convolutional Auto-Encoders for Hierarchical ”feature Extraction, J. Masci *et al.* (2011).

10 “GSNs: Generative Stochastic Networks”, G. Alain *et al.* (2015).

11 “Winner-Tak-All Autoencoders”, A. Makhzani and B. Frey (2015).

12 “Generative Adevsarial Networks”, I. Goodfellow *et al.* (2014).

4. O que são autoencoders incompletos e supercompletos? Qual é o principal risco de um autoencoder excessivamente incompleto? E quanto ao principal risco de um autoencoder supercompleto?
5. Como você amarra pesos em um autoencoder empilhado? Qual é o sentido de se fazer isso?
6. Qual é a técnica comum para visualizar as características aprendidas pela camada inferior de um autoencoder empilhado? E quanto às camadas superiores?
7. O que é um modelo gerador? Você pode nomear um tipo de autoencoder gerador?
8. Utilizaremos um autoencoder de remoção de ruído para pré-treinar um classificador de imagens:
  - Você pode utilizar o MNIST (mais simples) ou outro grande conjunto de imagens, como o CIFAR10 (<https://goo.gl/VbsmxG>) se quiser um desafio maior. Se você escolher o CIFAR10, precisará escrever o código para carregar lotes de imagens para treinamento. Se você quiser pular essa parte, o modelo zoo do TensorFlow contém ferramentas para fazer exatamente isso (<https://goo.gl/3iENgb>);
  - Dívida o conjunto de dados em um conjunto de treinamento e um conjunto de testes. Treine um autoencoder profundo de remoção de ruído no conjunto de treinamento completo;
  - Verifique se as imagens estão razoavelmente bem reconstruídas e visualize as características de baixo nível. Visualize as imagens que mais ativam cada neurônio na camada de codificação;
  - Construa uma rede neural profunda de classificação reutilizando as camadas inferiores do autoencoder. Treine-a utilizando apenas 10% do conjunto de treinamento. Você consegue fazer com que ela tenha desempenho tão bom quanto o mesmo classificador treinado no conjunto de treinamento completo?
9. *Semantic hashing* foi uma técnica introduzida em 2008 por Ruslan Salakhutdinov e Geoffrey Hinton (<https://goo.gl/LXzFX6>),<sup>13</sup> utilizada para a recuperação eficiente de informações: um documento (por exemplo, uma imagem) é passado por um sistema, geralmente, uma rede neural que exibe um vetor binário de baixa dimensionalidade (por exemplo, 30 bits). É provável que dois documentos semelhantes tenham hashes idênticos ou muito semelhantes. Ao indexar cada documento utilizando seu hash, é possível recuperar quase instantaneamente muitos documentos semelhantes a um específico, mesmo que haja bilhões deles: basta calcular o hash do documento e procurar todos aqueles com o mesmo hash (ou

---

<sup>13</sup> “Semantic Hashing”, R. Salakhutdinov and G. Hinton (2008).

hashes diferindo por apenas um ou dois bits). Implementaremos o hashing semântico com a utilização de um autoencoder empilhado ligeiramente ajustado:

- Crie um autoencoder empilhado contendo duas camadas ocultas abaixo da camada de codificação e treine-o no conjunto de dados de imagens que você utilizou no exercício anterior. A camada de codificação deve conter 30 neurônios e utilizar a função de ativação logística para gerar valores entre 0 e 1. Após o treinamento, para produzir o hash de uma imagem, execute-o através do autoencoder, obtenha a saída da camada de codificação e arredonde todos os valores para o inteiro mais próximo (0 ou 1);
  - Um truque proposto por Salakhutdinov e Hinton é adicionar ruído gaussiano (com média zero) às entradas da camada de codificação somente durante o treinamento. O autoencoder aprenderá a alimentar grandes valores para a camada de codificação a fim de preservar uma alta relação sinal-ruído (de modo que o ruído se torne insignificante). Por sua vez, isso significa que a função logística da camada de codificação provavelmente saturará em 0 ou 1. Como resultado, arredondando os códigos para 0 ou 1 não os distorcerá muito e isso aumentará a confiabilidade dos hashes;
  - Calcule o hash de cada imagem e veja se imagens com hashes idênticos são parecidas. Como o MNIST e o CIFAR10 são rotulados, uma forma mais objetiva de medir o desempenho do autoencoder para o hashing semântico é garantir que as imagens com o mesmo hash tenham a mesma classe. Uma maneira de fazer isso é medir o coeficiente de Gini médio (apresentado no Capítulo 6) dos conjuntos de imagens com hashes idênticos (ou muito semelhantes);
  - Tente ajustar os hiperparâmetros com a utilização da validação cruzada;
  - Observe que, com um conjunto de dados rotulados, outra abordagem seria treinar uma rede neural convolucional (consulte o Capítulo 13) para classificação e, em seguida, utilizar a camada abaixo da camada de saída para produzir os *hashes*. Veja o artigo de 2015 de Jinma Gua e Jianmin Li (<https://goo.gl/i9FTln>)<sup>14</sup> e descubra se isso funciona melhor.
10. Treine um autoencoder variacional no conjunto de dados da imagem utilizado nos exercícios anteriores (MNIST ou CIFAR10) e faça com que ele gere imagens. Como alternativa, tente encontrar um conjunto de dados não rotulados do seu interesse e veja se é possível gerar novas amostras.

Soluções para estes exercícios estão disponíveis no Apêndice A.

---

14 “CNN Based Hashing for Image Retrieval,” J. Gua and J. Li (2015)

## Capítulo 16

# Aprendizado por Reforço

O Aprendizado por Reforço (AR) é atualmente um dos campos mais empolgantes do Aprendizado de Máquina, e também um dos mais antigos. Ele existe desde os anos 1950, e vem produzindo muitas aplicações interessantes ao longo dos anos,<sup>1</sup> em particular nos jogos (por exemplo, o *TD-Gammon*, um programa de *gamão*) e no controle de máquinas, mas raramente aparece em manchetes. No entanto, uma revolução ocorreu em 2013 quando pesquisadores de uma startup inglesa chamada DeepMind demonstraram um sistema que poderia aprender a jogar praticamente qualquer jogo de Atari do zero (<https://goo.gl/hceDs5>)<sup>2</sup> e, na maioria deles, eventualmente superar os humanos (<https://goo.gl/hgpvz7>)<sup>3</sup> somente com a utilização de pixels brutos como entradas, e sem qualquer conhecimento prévio das regras dos jogos.<sup>4</sup> Este foi o primeiro de uma série de feitos incríveis, culminando em maio de 2017 com a vitória do seu sistema AlphaGo contra Ke Jie, o campeão mundial do jogo *Go*. Nenhum programa chegou perto de bater um mestre deste jogo, muito menos o campeão mundial. Hoje, todo o setor do AR está fervendo com novas ideias, com uma ampla gama de aplicações. A DeepMind foi comprada pelo Google por mais de 500 milhões de dólares em 2014.

Então, como eles fizeram? Em retrospecto, parece bastante simples: eles aplicaram o poder do Aprendizado Profundo no campo do Aprendizado por Reforço, e isso funcionou além de seus sonhos mais loucos. Neste capítulo, primeiro explicaremos o que é Aprendizado por Reforço e no que ele é bom, e então apresentaremos duas das técnicas mais importantes no Aprendizado por Reforço profundo: *gradientes de políticas* e *deep*

1 Para mais detalhes, não deixe de conferir o livro de Richard Sutton e Andrew Barto sobre AR (<https://goo.gl/7utZaz>), *Reinforcement Learning: An Introduction* (MIT Press), ou o curso gratuito de AR online de David Silver (<https://goo.gl/AWcMFW>) na University College London.

2 “Playing Atari with Deep Reinforcement Learning”, V. Mnih *et al.* (2013).

3 “Human-level control through deep reinforcement learning”, V. Mnih *et al.* (2015).

4 Confira os vídeos do sistema da DeepMind aprendendo a jogar Space Invaders, Breakout e outros em <https://goo.gl/yTsH6X>.

*Q-networks* (DQN), incluindo uma discussão dos processos de decisão de Markov (MDP, do inglês). Utilizaremos estas técnicas para treinar um modelo para equilibrar um bastão em um carrinho em movimento e outro para jogar jogos de Atari. As mesmas técnicas podem ser utilizadas para uma ampla variedade de tarefas, desde robôs andando, a carros autônomos.

## Aprendendo a Otimizar Recompensas

No *Aprendizado por Reforço*, um *agente* de software faz *observações* e realiza *ações* dentro de um *ambiente* e, em troca, recebe *recompensas*. Seu objetivo é aprender a agir de forma a maximizar suas recompensas esperadas de longo prazo. Caso não se importe com um pouco de antropomorfismo, pode-se pensar em recompensas positivas como o prazer, e recompensas negativas como a dor (o termo “recompensa” é um pouco enganador neste caso). Em suma, o agente age no ambiente e aprende por tentativa e erro a maximizar seu prazer e minimizar sua dor.

Esta é uma configuração bastante genérica que pode ser aplicada a uma ampla variedade de tarefas. Seguem alguns exemplos (veja a Figura 16-1):

- a. O agente pode ser um programa que controla um robô andante. Neste caso, o ambiente é o mundo real e o agente observa-o através de um conjunto de *sensores*, como câmeras e sensores de toque, e suas ações consistem em enviar sinais para ativar motores. Ele pode ser programado para obter recompensas positivas sempre que se aproximar do destino e recompensas negativas sempre que perder tempo, seguir na direção errada ou cair;
- b. O agente pode ser um programa que controla a Ms. Pac-Man. Neste caso, o ambiente é uma simulação do jogo de Atari, as ações são as nove possíveis posições do controle (superior esquerdo, baixo, centro e assim por diante), as observações são capturas de tela e as recompensas são apenas pontos do jogo;
- c. Da mesma forma, o agente pode ser um programa jogando um jogo de tabuleiro, como o Go;
- d. O agente não precisa controlar fisicamente uma coisa em movimento (ou virtualmente). Por exemplo, pode ser um termostato inteligente recebendo recompensas sempre que estiver próximo da temperatura desejada e economizar energia, além das recompensas negativas quando os humanos precisarem ajustar a temperatura para que o agente aprenda a antecipar suas necessidades;
- e. O agente pode observar os preços do mercado de ações e decidir a cada segundo quanto comprar ou vender. Recompensas são obviamente os ganhos e perdas monetárias.

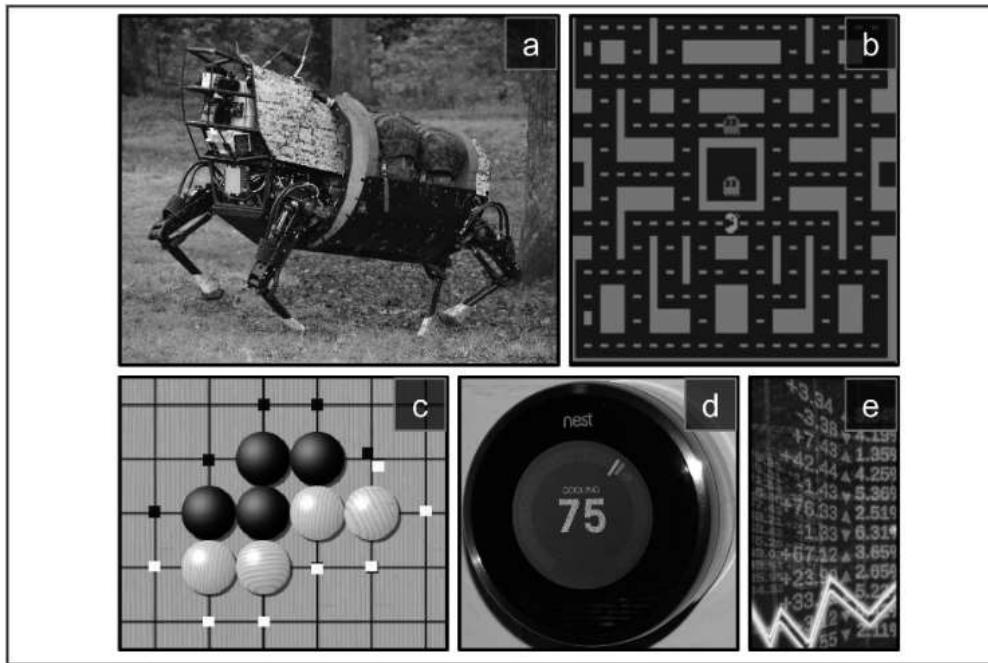


Figura 16-1. Exemplos de Aprendizado por Reforço: (a) robô andante, (b) Ms. Pac-Man, (c) jogador de Go, (d) termostato, (e) operador automático<sup>5</sup>

Note que pode não haver nenhuma recompensa positiva; por exemplo, o agente pode se movimentar em um labirinto recebendo uma recompensa negativa a cada intervalo de tempo, então é melhor encontrar a saída o mais rápido possível! Há muitos outros exemplos de tarefas nas quais o Aprendizado por Reforço é bem adequado, como carros autônomos, colocar anúncios em uma página da Web ou controlar onde um sistema de classificação de imagens deve focar sua atenção.

## Pesquisa de Políticas

O algoritmo utilizado pelo agente de software para determinar suas ações é chamado de *política*. Por exemplo, a política poderia ser uma rede neural absorvendo as observações como entradas e produzindo a ação a ser tomada (veja a Figura 16-2).

<sup>5</sup> Imagens (a), (c) e (d) são reproduzidas da Wikipedia. (a) e (d) estão no domínio público. (c) foi criada pelo usuário Stevertigo e lançada pela Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>). (b) é uma captura de tela do jogo Ms. Pac-Man, copyright Atari (o autor acredita que o uso seja justo para este capítulo). (e) foi reproduzido a partir do Pixabay, lançado pela Creative Commons CC0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

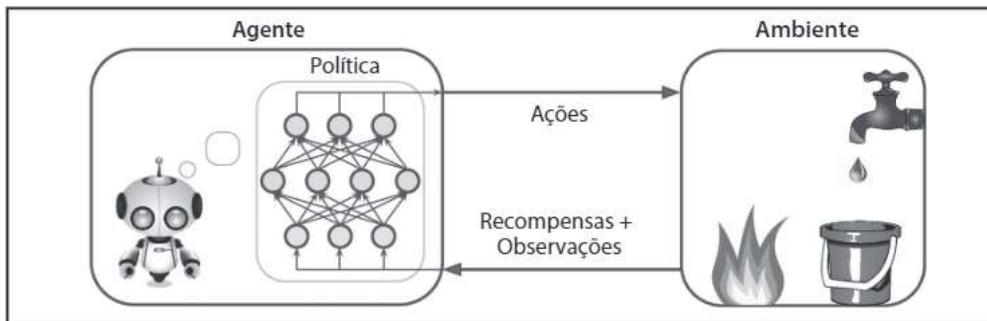


Figura 16-2. Aprendizado por reforço utilizando uma política de rede neural

A política pode ser qualquer algoritmo que você possa imaginar, e nem precisa ser determinista. Por exemplo, considere um aspirador de pó robótico cuja recompensa é a quantidade de poeira acumulada em 30 minutos. Sua política poderia ser avançar com alguma probabilidade  $p$  a cada segundo, ou girar aleatoriamente para a esquerda ou para a direita com probabilidade  $1 - p$ . O ângulo de rotação seria um ângulo aleatório entre  $-r$  e  $+r$ . Como essa política envolve alguma aleatoriedade, ela é chamada de *política estocástica*. O robô terá uma trajetória errática, o que garante que ele eventualmente chegará a qualquer lugar que possa alcançar e recolher toda a poeira. A questão é: quanto pó ele pegará em 30 minutos?

Como você treinaria um robô desses? Existem apenas dois *parâmetros de política* que você pode ajustar: a probabilidade  $p$  e o intervalo do ângulo  $r$ . Um possível algoritmo de aprendizado poderia ser testar muitos valores diferentes para esses parâmetros e escolher a combinação que tenha o melhor desempenho (veja a Figura 16-3). Este é um exemplo de *pesquisa de política*, neste caso utilizando uma abordagem de força bruta. No entanto, encontrar um bom conjunto de parâmetros dessa maneira quando o *espaço de política* for muito grande (o que geralmente é o caso), é como procurar uma agulha em um gigantesco palheiro.

Outra maneira de explorar o espaço da política seria utilizar *algoritmos genéticos*. Por exemplo, você poderia criar aleatoriamente uma primeira geração de 100 políticas e testá-las, depois “matar” as 80 piores políticas<sup>6</sup> e fazer com que as 20 sobreviventes produzam 4 descendentes cada. Uma descendente é apenas uma cópia de seu genitor,<sup>7</sup> além de alguma variação aleatória. As políticas sobreviventes e seus descendentes jun-

<sup>6</sup> Muitas vezes, é melhor dar aos participantes fracos uma pequena chance de sobrevivência, para preservar alguma “diversidade no pool genético”.

<sup>7</sup> Se houver apenas um genitor, esse processo é chamado de *reprodução assexuada*. Com dois (ou mais) genitores, é chamado de *reprodução sexuada*. O genoma de uma prole (neste caso, um conjunto de parâmetros de políticas) é composto aleatoriamente por partes dos genomas de seus pais.

tos constituem a segunda geração e é possível continuar iterando dessa maneira pelas gerações até encontrar uma boa política.

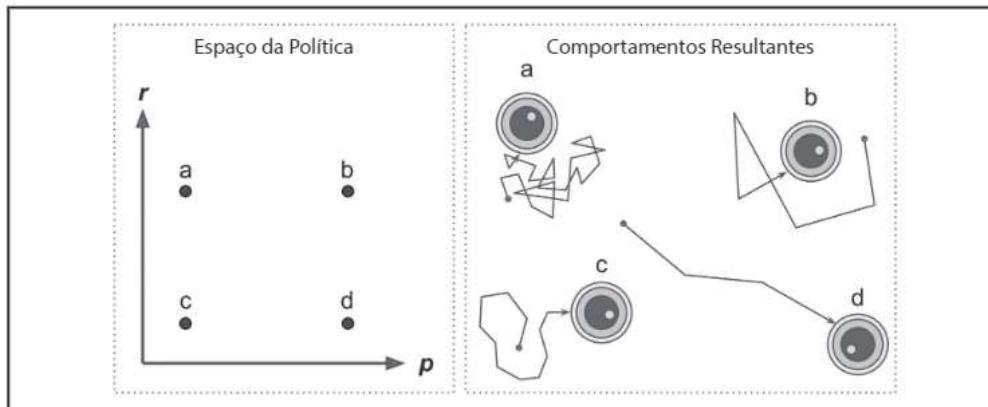


Figura 16-3. Quatro pontos no espaço da política e o comportamento correspondente do agente

Outra abordagem seria usar técnicas de otimização, avaliando os gradientes das recompensas com relação aos parâmetros da política  $\pi$ , e, então, ajustando esses parâmetros seguindo o gradiente na direção das recompensas mais altas (*gradiente ascendente*), abordagem chamada de *gradientes de políticas* (PG, do inglês) que discutiremos com mais detalhes mais adiante. Por exemplo, voltando ao robô aspirador de pó, você poderia aumentar um pouco  $p$  e avaliar se isso aumenta a quantidade de poeira captada pelo robô em 30 minutos; se isso acontecer, aumente  $p$  um pouco mais ou, então, o reduza. Com a utilização do TensorFlow, implementaremos um algoritmo PG popular, mas antes precisamos criar um ambiente para o agente morar, então é hora de apresentar o OpenAI gym.

## Introdução ao OpenAI Gym

Um dos desafios do Aprendizado por Reforço é que, para treinar um agente, você primeiro precisa ter um ambiente de trabalho. Você precisará de um simulador de jogos da Atari se quiser programar um agente que aprenderá a jogar um jogo de Atari. Se quiser programar um robô andante, então o ambiente é o mundo real e você pode treinar seu robô diretamente, mas isso tem seus limites: se o robô cair de um penhasco você não pode simplesmente clicar em “desfazer”. Você também não pode acelerar o tempo; adicionar mais poder de computação não fará com que o robô se mova mais rápido e, geralmente, é muito caro treinar 1 mil robôs em paralelo. Em suma, o treinamento é difícil e lento no mundo real, então você precisa de um *ambiente simulado* pelo menos para iniciar o treinamento.

O OpenAI gym (<https://gym.openai.com/>)<sup>8</sup> é um kit de ferramentas que fornece uma ampla variedade de ambientes simulados (jogos de Atari, jogos de tabuleiro, simulações físicas em 2D e 3D e assim por diante) para que você possa treinar agentes, compará-los, ou desenvolver novos algoritmos de RL.

Instalaremos o OpenAI gym. Se você criou um ambiente isolado utilizando virtualenv, primeiro precisa ativá-lo:

```
$ cd $ML_PATH           # Seu diretório de AM (por ex., $HOME/ml)
$ source env/bin/activate
```

Em seguida, instale o OpenAI Gym (se você não estiver utilizando um virtualenv, precisará de direitos de administrador ou adicionar a opção `--user`):

```
$ pip3 install --upgrade gym
```

Em seguida, abra um Python shell ou um notebook Jupyter e crie seu primeiro ambiente:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2017-08-27 11:08:05,742] Making new env: CartPole-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115,  0.02337094,  0.00720711])
>>> env.render()
```

A função `make()` cria um ambiente, neste caso um ambiente CartPole. Esta é uma simulação 2D na qual um carrinho pode ser acelerado para a esquerda ou para a direita a fim de equilibrar um bastão posicionado sobre ele (veja a Figura 16-4). Depois que o ambiente é criado, devemos iniciá-lo utilizando o método `reset()`, o que retorna a primeira observação. As observações dependem do tipo de ambiente, cada uma delas é um array 1D NumPy que contém quatro *floats* para o ambiente CartPole: esses floats representam a posição horizontal do carrinho (0.0 = centro), sua velocidade, o ângulo do bastão (0.0 = vertical) e sua velocidade angular. Finalmente, o método `render()` exibe o ambiente como mostrado na Figura 16-4.

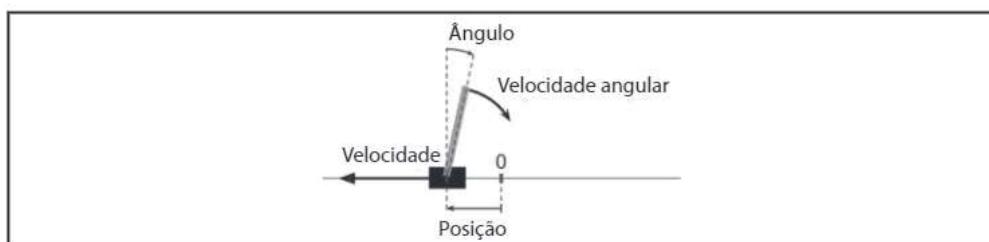


Figura 16-4. O ambiente CartPole

<sup>8</sup> A OpenAI é uma empresa de pesquisa de inteligência artificial sem fins lucrativos financiada em parte por Elon Musk. Seu objetivo declarado é promover e desenvolver IAs amigáveis que beneficiem a humanidade (em vez de exterminá-la).

Ajuste o parâmetro `mode` para `rgb_array` se quiser que `render()` retorne a imagem renderizada como um array NumPy (note que outros ambientes podem suportar diferentes modos):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # altura, largura, canais (3=RGB)
(400, 600, 3)
```



Infelizmente, o CartPole (e alguns outros ambientes) renderiza a imagem na tela mesmo se você definir o modo como “`rgb_array`”. A única maneira de evitar isso é utilizar um servidor X falso como `Xvfb` ou `Xdummy`. Por exemplo, você pode instalar o `Xvfb` e iniciar o Python com o seguinte comando: `xvfb-run -s "- screen 0 1400x900x24" python`. Ou utilizar o pacote `xvfbwrapper` (<https://goo.gl/wR1oJl>).

Perguntaremos ao ambiente quais ações são possíveis:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` significa que as ações possíveis são os inteiros 0 e 1 que representam a aceleração à esquerda (0) ou à direita (1). Outros tipos de ações ou outros ambientes podem ter ações mais discretas (por exemplo, contínuas). Como o bastão está inclinado para a direita, aceleraremos o carrinho para a direita:

```
>>> action = 1 # acelere bem
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

O método `step()` executa a dada ação e retorna quatro valores:

#### obs

Esta é a nova observação. O carrinho agora está se movendo para a direita (`obs[1]>0`). O bastão ainda está inclinado para a direita (`obs[2]>0`), mas sua velocidade angular agora é negativa (`obs[3]<0`), provavelmente será inclinado para a esquerda após o próximo passo.

#### reward

Neste ambiente, você consegue uma recompensa de 1,0 a cada passo, não importa o que você faça, então o objetivo é continuar correndo o maior tempo possível.

**done**

Este valor será `True` quando o *episode* terminar, o que acontecerá quando o bastão se inclinar demais. Depois disso, o ambiente deve ser redefinido antes de poder ser usado novamente.

**info**

Este dicionário pode fornecer informações adicionais de depuração em outros ambientes. Esses dados não devem ser utilizados para treinamento (isso seria trapaça).

Programaremos uma política simples que acelera para a esquerda quando o bastão está inclinado para a esquerda, e para a direita quando está inclinado para a direita. Nós executaremos essa política para ver as recompensas médias que ele recebe em mais de 500 episódios:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # no máximo mil passos, não queremos rodar para sempre
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

Este código é autoexplicativo. Vejamos o resultado:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(42.12599999999998, 9.1237121830974033, 24.0, 68.0)
```

Mesmo com 500 tentativas, essa política nunca conseguiu manter o bastão de pé por mais de 68 etapas consecutivas. Nada bom. Se você olhar para a simulação nos notebooks do Jupyter (<https://github.com/ageron/handson-ml>) verá que o carrinho oscila para a esquerda e para a direita mais e mais, até que o bastão se incline demais. Veremos se uma rede neural pode criar uma política melhor.

## Políticas de Rede Neural

Criaremos uma política de rede neural e, assim como a política que codificamos anteriormente, essa rede neural tomará uma observação como entrada e exibirá a ação a ser executada. Mais precisamente, estimamos uma probabilidade para cada ação e, em seguida, selecionamos ale-

toriamente uma ação de acordo com as probabilidades estimadas (ver Figura 16-5). Há apenas duas ações possíveis no caso do ambiente CartPole (esquerda ou direita), então só precisamos de um neurônio de saída que exibirá a probabilidade  $p$  de ação 0 (esquerda), e, claro, a probabilidade da ação 1 (direita) será  $1 - p$ . Por exemplo, se ele exibir 0,7, então escolheremos a ação 0 com 70% de probabilidade e a ação 1 com 30% de probabilidade.

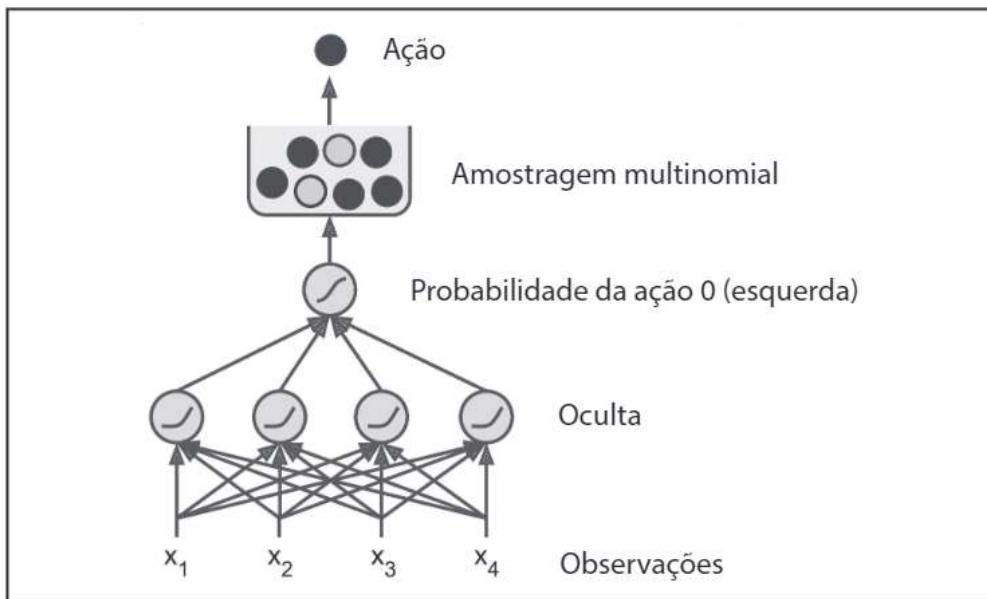


Figura 16-5. Política de Rede Neural

Você pode se perguntar por que estamos escolhendo uma ação aleatória com base na probabilidade dada pela rede neural, em vez de apenas escolher a ação com a pontuação mais alta. Essa abordagem permite que o agente encontre o equilíbrio certo entre *explorar* novas ações e a *exploração* das ações que funcionam bem. Pense nesta analogia: suponha que você vá a um restaurante pela primeira vez e todos os pratos pareçam igualmente atraentes, assim você escolhe aleatoriamente um. Se for bom, você pode aumentar a probabilidade de pedi-lo na próxima vez, mas você não deve aumentar essa probabilidade até 100% ou então nunca experimentará os outros pratos, alguns dos quais podem ser ainda melhores do que aquele que você escolheu.

Observe também que as ações e observações passadas podem ser ignoradas com segurança nesse ambiente específico, já que cada observação contém o estado completo do ambiente. Se houvesse algum estado oculto, talvez fosse necessário considerar ações e observações passadas também. Por exemplo, se o ambiente apenas revelou a posição do carro, mas não a sua velocidade, você teria que considerar não apenas a observação

atual, mas também a observação anterior a fim de estimar a velocidade atual. Outro exemplo é quando as observações forem ruidosas; nesse caso, você geralmente quer usar as últimas observações para estimar o estado atual mais provável. O problema do CartPole é, portanto, extremamente simples: as observações são livres de ruído e contêm o estado completo do ambiente.

Este é o código para construir esta política de rede neural usando o TensorFlow:

```
import tensorflow as tf

# 1. Especifique a arquitetura da rede neural
n_inputs = 4 # == env.observation_space.shape[0]
n_hidden = 4 # é uma tarefa simples, não precisamos de mais neurônios ocultos
n_outputs = 1 # só sai a probabilidade de aceleração faltante
initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Construa a rede neural
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,
                        kernel_initializer=initializer)
logits = tf.layers.dense(hidden, n_outputs,
                        kernel_initializer=initializer)
outputs = tf.nn.sigmoid(logits)

# 3. Selecione uma ação aleatória com base nas probabilidades estimadas
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

init = tf.global_variables_initializer()
```

Vamos entender esse código:

1. Após as importações, definimos a arquitetura da rede neural. O número de entradas é o tamanho do espaço de observação (que no caso do CartPole é quatro), temos apenas quatro unidades ocultas e não há necessidade de mais, e temos apenas uma probabilidade de saída (a probabilidade de ir para a esquerda);
2. Em seguida, construímos a rede neural. Neste exemplo, é um perceptron normal de várias camadas com uma única saída. Observe que a camada de saída utiliza a função de ativação logística (sigmoide) para gerar uma probabilidade de 0,0 a 1,0. Se houvesse mais de duas ações possíveis, haveria um neurônio de saída por ação e você utilizaria a função de ativação do softmax;
3. Por fim, chamamos a função `multinomial()` para escolher uma ação aleatória. Esta função, dada a probabilidade de log de cada inteiro, amostra um (ou mais) inteiros. Por exemplo, se você a chamar com o array `[np.log(0.5), np.log(0.2), np.log(0.3)]` e com `num_samples=5`, ela produzirá cinco inteiros, cada um dos quais terá 50% de probabilidade de ser 0, 20% de ser 1 e 30% de ser 2. No nosso caso, precisamos apenas de um inteiro representando a ação a ser executada. Como o tensor `outputs` contém apenas a probabilidade de ir para a esquerda, de-

vemos primeiro concatenar `1-outputs` para que ele tenha um tensor que contenha a probabilidade de ambas as ações, esquerda e direita. Observe que, se houvesse mais de duas ações possíveis, a rede neural teria que gerar uma probabilidade por ação para que você não precisasse da etapa de concatenação.

Ok, agora temos uma política de rede neural com observações e ações de saída, mas como a treinamos?

## Avaliação das Ações: o Problema de Atribuição de Crédito

Como de costume, se soubéssemos qual seria a melhor ação em cada etapa, poderíamos treinar a rede neural minimizando a entropia cruzada entre a probabilidade estimada e a probabilidade de destino. Seria apenas um aprendizado supervisionado comum, porém, no Aprendizado por Reforço, a única orientação que o agente recebe é por meio de recompensas, que são tipicamente escassas e atrasadas. Por exemplo, se o agente consegue equilibrar o bastão por 100 etapas, como ele pode saber quais das 100 ações foram boas e quais delas foram ruins? Ele só sabe que o bastão caiu após a última ação, mas certamente ele não é inteiramente responsável por esta última ação, o que é chamado de *problema de atribuição de crédito*: quando o agente recebe uma recompensa, é difícil saber quais ações devem ser creditadas (ou culpadas) por isso. Pense em um cachorro que é recompensado horas depois de se comportar bem; ele entenderá por que é recompensado?

Uma estratégia comum para resolver esse problema é avaliar uma ação com base na soma de todas as recompensas que vêm depois dela, geralmente aplicando uma *tasa de desconto*  $r$  em cada etapa. Por exemplo (veja a Figura 16-6), se um agente decide ir pela direita três vezes seguidas e recebe +10 de recompensa após o primeiro passo, 0 após o segundo passo, e finalmente -50 após o terceiro passo, então, assumindo que utilizamos uma taxa de desconto  $r = 0,8$ , a primeira ação terá uma pontuação total de  $10 + r \times 0 + r^2 \times (-50) = -22$ . Se a taxa de desconto for próxima de 0, as recompensas futuras não serão muito importantes em comparação com as recompensas imediatas. Por outro lado, se a taxa de desconto for próxima de 1, as recompensas no futuro contarão quase tanto quanto as recompensas imediatas. As taxas de desconto típicas são 0,95 ou 0,99. As recompensas de 13 passos no futuro contam mais ou menos metade das recompensas imediatas (já que  $0,95^{13} \approx 0,5$ ) com uma taxa de desconto de 0,95, enquanto com uma taxa de desconto de 0,99 por metade das recompensas imediatas ele terá uma recompensa de 69 etapas na contagem futura. No ambiente CartPole, as ações têm efeitos de curto prazo, portanto a escolha de uma taxa de desconto de 0,95 parece razoável.

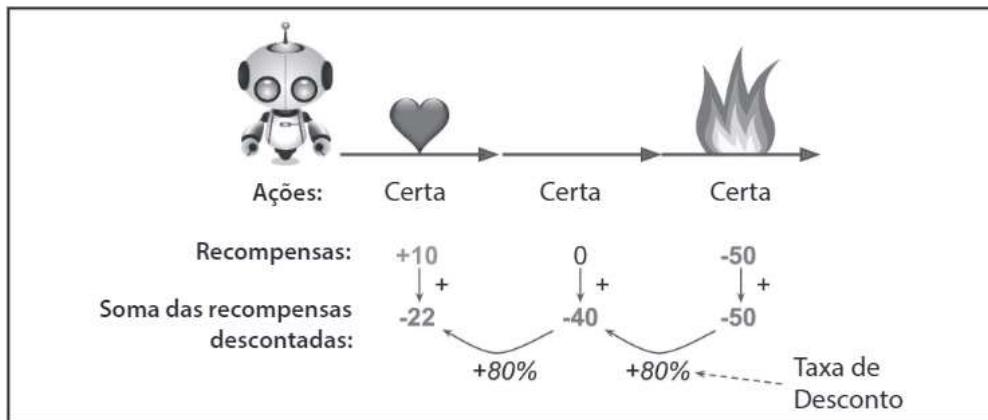


Figura 16-6. Recompensas descontadas

É claro que uma boa ação pode ser seguida por várias ações ruins que fazem com que o bastão caia rapidamente, resultando em uma boa ação obtendo uma pontuação baixa (da mesma forma, um bom ator pode, às vezes, estrelar um filme terrível). No entanto, em média, as boas ações terão uma pontuação melhor do que as ruins se jogarmos bastante o jogo. Assim, para obter pontuações de ação bastante confiáveis, devemos executar muitos episódios e normalizar todas as pontuações da ação (subtraindo a média e dividindo pelo desvio padrão). Depois disso, podemos supor razoavelmente que as ações com pontuação negativa foram ruins, enquanto as ações com pontuação positiva foram boas. Perfeito — agora que temos uma forma de avaliar cada ação utilizando gradientes de política, estamos prontos para treinar nosso primeiro agente. Veremos como.

## Gradientes de Política

Como discutido anteriormente, os algoritmos PG otimizam os parâmetros de uma política seguindo os gradientes em direção a recompensas mais altas. Uma classe popular de algoritmos PG, chamada *REINFORCE algorithms*, foi introduzida em 1992 (<https://goo.gl/tUe4Sh>)<sup>9</sup> por Ronald Williams. Uma variante comum é:

1. Primeiro, deixe a política da rede neural jogar o jogo várias vezes e, em cada etapa, calcule os gradientes que tornariam a ação escolhida ainda mais provável, mas não aplique-os ainda;
2. Depois de executar vários episódios, calcule a pontuação de cada ação (utilizando o método descrito no parágrafo anterior);

<sup>9</sup> "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning", R. Williams (1992).

3. Se a pontuação da ação for positiva, significa que a ação foi boa e você deseja aplicar os gradientes calculados anteriormente para tornar a ação ainda mais provável de ser escolhida no futuro. No entanto, se a pontuação for negativa, significa que a ação foi ruim e você deseja aplicar os gradientes opostos para tornar essa ação um pouco *menos* provável no futuro. A solução é simplesmente multiplicar cada vetor de gradiente pela pontuação da ação correspondente;
4. Por fim, calcule a média de todos os vetores de gradiente resultantes e a utilize para executar uma etapa de Gradiente Descendente.

Implementaremos esse algoritmo utilizando o TensorFlow e treinaremos a política de redes neurais que construímos anteriormente para que ela aprenda a equilibrar o bastão no carrinho. Começaremos concluindo a fase de construção que codificamos anteriormente para adicionar a probabilidade de destino, a função de custo e a operação de treinamento. Já que estamos agindo como se a ação escolhida fosse a melhor ação possível, se ela for a ação 0 (esquerda) a probabilidade de destino deve ser 1.0, e se for a ação 1 (direita) será 0,0:

```
y = 1. - tf.to_float(action)
```

Agora que temos uma probabilidade de destino, podemos definir a função de custo (entropia cruzada) e calcular os gradientes:

```
learning_rate = 0.01

cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y,
                                                       logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

Observe que estamos chamando o método `compute_gradients()` do otimizador em vez do `minimize()` porque queremos ajustar os gradientes antes de aplicá-los.<sup>10</sup> O método `compute_gradients()` retorna uma lista de pares vetor/variável do gradiente (um par por variável treinável). Para que obter os seus valores seja mais conveniente, colocaremos todos os gradientes em uma lista:

```
gradients = [grad for grad, variable in grads_and_vars]
```

Ok, agora vem a parte complicada. Durante a fase de execução, o algoritmo executará a política e, em cada etapa, vai avaliar esses tensores do gradiente e armazenar seus valores. Como explicado anteriormente, depois de vários episódios, ele ajustará esses gradientes (ou seja, os multiplicará pelas pontuações de ação e os normalizará) e calculará a média dos gradientes ajustados. Em seguida, ele precisará alimentar os gradientes resultantes de volta ao ot-

---

<sup>10</sup> Nós já fizemos algo semelhante no Capítulo 11 quando discutimos o Recorte do Gradiente: primeiro calculamos os gradientes, depois recortamos e, finalmente, aplicamos os gradientes recortados.

mizador para que possa executar uma etapa de otimização porque precisamos de um placeholder por vetor do gradiente. Além disso, devemos criar a operação que aplicará os gradientes atualizados chamando a função do otimizador `apply_gradients()` que pega uma lista de pares vetores/variáveis do gradiente. Em vez de fornecer os vetores originais do gradiente, forneceremos uma lista contendo os gradientes atualizados (ou seja, os fornecidos pelos placeholders do gradiente):

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))

training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

Vamos recuar e dar uma olhada na fase completa de construção:

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,
                        kernel_initializer=initializer)
logits = tf.layers.dense(hidden, n_outputs,
                        kernel_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Rumo à fase de execução! Dadas as recompensas cruas, precisaremos de algumas funções para calcular as recompensas totais com desconto na fase de execução, e normalizar os resultados em múltiplos episódios:

```
def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards = rewards[step] + cumulative_rewards * discount_rate
        discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards, discount_rate)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std
            for discounted_rewards in all_discounted_rewards]
```

Verificaremos se isso funciona:

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([ 1.26665318,  1.0727777 ])]
```

A chamada para `discount_rewards()` retorna exatamente o que esperamos (veja a Figura 16-6). Verifique que a função `discount_and_normalize_rewards()` realmente retorna as pontuações normalizadas para cada ação em ambos os episódios. Observe que o primeiro episódio foi muito pior do que o segundo, então suas pontuações normalizadas são todas negativas; todas as ações do primeiro episódio seriam consideradas ruins e, inversamente, todas as ações do segundo episódio seriam consideradas boas:

Temos agora tudo o que precisamos para treinar a política.

```
n_iterations = 250      # número de iterações de treinamento
n_max_steps = 1000      # etapas máximas por episódio
n_games_per_update = 10 # treine a política a cada 10 episódios
save_iterations = 10    # salve o modelo a cada 10 iterações de treinamento
discount_rate = 0.95

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []    # todas as sequências de recompensas para cada episódio
```

```

all_gradients = [] # gradientes saídos em cada etapa para cada episódio
for game in range(n_games_per_update):
    current_rewards = [] # todas as recompensas do episódio atual
    current_gradients = [] # todos os gradientes do episódio atual
    obs = env.reset()
    for step in range(n_max_steps):
        action_val, gradients_val = sess.run(
            [action, gradients],
            feed_dict={X: obs.reshape(1, n_inputs)}) # uma obs
        obs, reward, done, info = env.step(action_val[0][0])
        current_rewards.append(reward)
        current_gradients.append(gradients_val)
        if done:
            break
    all_rewards.append(current_rewards)
    all_gradients.append(current_gradients)

# Aqui, rodamos a política em 10 episódios e estamos
# prontos para uma atualização da política usando o algoritmo descrito anteriormente
all_rewards = discount_and_normalize_rewards(all_rewards, discount_rate)
feed_dict = {}
for var_index, grad_placeholder in enumerate(gradient_placeholders):
    # multiplique todos os gradientes pela pontuação de ação e compute a média
    mean_gradients = np.mean(
        [reward * all_gradients[game_index][step][var_index]
         for game_index, rewards in enumerate(all_rewards)
         for step, reward in enumerate(rewards)],
        axis=0)
    feed_dict[grad_placeholder] = mean_gradients
sess.run(training_op, feed_dict=feed_dict)
if iteration % save_iterations == 0:
    saver.save(sess, "./my_policy_net_pg.ckpt")

```

Cada iteração do treinamento começa executando a política de 10 episódios (com o máximo de 1 mil etapas por episódio, para evitar a execução para sempre). Também calculamos os gradientes em cada etapa fingindo que a ação escolhida foi a melhor. Depois que esses 10 episódios foram executados, calculamos as pontuações de ação usando a função `discount_and_normalize_rewards()`; passamos por cada variável treinável, em todos os episódios e passos, para multiplicar cada vetor gradiente por sua ação de pontuação correspondente, e calculamos a média dos gradientes resultantes. Finalmente, executamos a operação de treinamento alimentando esses gradientes médios (um por variável treinável). Também salvamos o modelo a cada 10 operações de treinamento.

E terminamos! Este código treinará a política de rede neural e aprenderá com sucesso a equilibrar o bastão no carrinho (você pode testá-lo nos notebooks Jupyter). Observe que há duas maneiras de o agente perder o jogo: o mastro pode inclinar demais ou o carrinho pode sair completamente da tela. Com 250 iterações de treinamento, a política aprende a equilibrar bem o bastão, mas ainda não é boa o suficiente para evitar que saia da tela. Algumas centenas de iterações de treinamento consertarão isso.



Os pesquisadores tentam encontrar algoritmos que funcionem bem mesmo quando o agente inicialmente não sabe nada sobre o ambiente. Entretanto, a menos que você esteja escrevendo um artigo, você deve injetar o máximo de conhecimento prévio possível no agente, pois isso acelerará o treinamento drasticamente. Por exemplo, você pode adicionar recompensas negativas proporcionais à distância do centro da tela e ao ângulo do bastão. Além disso, se você já tiver uma política razoavelmente boa (por exemplo, codificada), vai querer treinar a rede neural para imitá-la antes de utilizar gradientes de política para aprimorá-la.

Apesar de sua relativa simplicidade, esse algoritmo é bastante poderoso e é possível usá-lo para resolver problemas muito mais difíceis do que equilibrar um bastão em um carrinho. Na verdade, o AlphaGo foi baseado em um algoritmo PG similar (e o *Monte Carlo Tree Search*, que está além do escopo deste livro).

Agora veremos outra família popular de algoritmos. Enquanto os algoritmos PG tentam otimizar diretamente a política para aumentar as recompensas, os algoritmos que veremos agora são menos diretos: o agente aprende a estimar a soma esperada de recompensas futuras descontadas para cada estado ou a soma esperada de recompensas futuras descontadas para cada ação em cada estado, então utiliza esse conhecimento para decidir como agir. Para entender esses algoritmos, devemos primeiro introduzir os *processos de decisão de Markov* (MDP, do inglês).

## Processos de Decisão de Markov

No início do século XX o matemático Andrey Markov estudou processos estocásticos sem memória chamados *cadeias de Markov*. Esse processo tem um número fixo de estados e evolui aleatoriamente de um estado para outro em cada etapa. A probabilidade de evoluir de um estado  $s$  para um estado  $s'$  é fixa e depende apenas do par  $(s, s')$ , não de estados passados (o sistema não tem memória).

A Figura 16-7 mostra um exemplo de uma cadeia de Markov com quatro estados. Suponha que o processo comece no estado  $s_0$ , e haja uma chance de 70% de que ele permaneça nesse estado na próxima etapa. Eventualmente, ele é obrigado a deixar esse estado e nunca mais voltar, pois nenhum outro estado aponta para  $s_0$ . Se for para o estado  $s_1$ , então provavelmente irá para o estado  $s_2$  (90% de probabilidade) e, então, imediatamente de volta para o estado  $s_1$  (com 100% de probabilidade). Pode alternar um número de

vezes entre estes dois estados, mas, eventualmente, cairá no estado  $s_3$  e permanecerá lá para sempre (este é um *estado terminal*). As cadeias de Markov podem ter dinâmicas muito diferentes e são muito utilizadas em termodinâmica, química, estatística e muito mais.

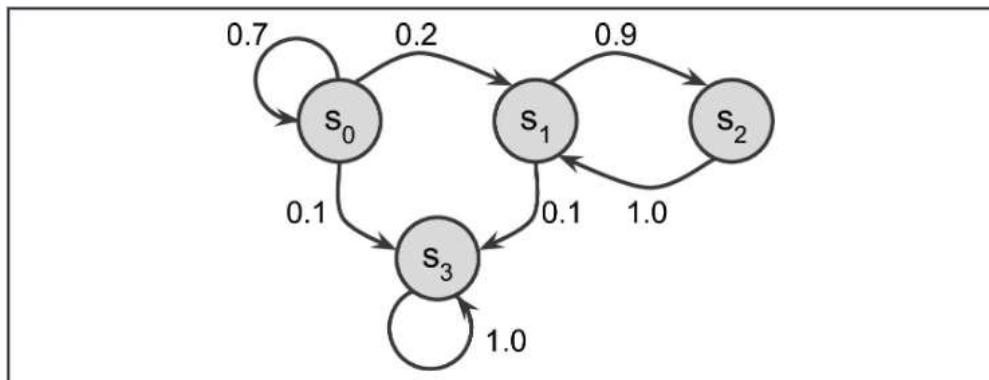


Figura 16-7. Exemplo de uma cadeia de Markov

Os processos de decisão de Markov foram descritos pela primeira vez na década de 1950 por Richard Bellman (<https://goo.gl/wZTVIN>)<sup>11</sup> e se assemelham a cadeias de Markov, mas com uma variação: a cada passo, um agente pode escolher uma das várias ações possíveis e as probabilidades de transição dependem da ação escolhida. Além disso, algumas transições de estado retornam alguma recompensa (positiva ou negativa) e o objetivo do agente é encontrar uma política que maximize-as ao longo do tempo.

Por exemplo, o MDP representado na Figura 16-8 possui três estados e até três possíveis ações discretas em cada etapa. Se começar no estado  $s_0$ , o agente pode escolher entre as ações  $a_0$ ,  $a_1$ , ou  $a_2$ . Se escolher a ação  $a_1$ , ele permanecerá no estado  $s_0$  com certeza, e sem recompensa alguma. Assim, se quiser, pode decidir ficar lá para sempre, mas, se escolher a ação  $a_0$ , ele tem 70% de probabilidade de ganhar uma recompensa de +10, e permanecer no estado  $s_0$ . Ele pode, então, tentar novamente por várias vezes para ganhar o máximo de recompensa possível, mas, em algum ponto, acabará no estado  $s_1$ . No estado  $s_1$  ele tem somente duas ações possíveis:  $a_0$  ou  $a_2$ , podendo escolher ficar parado selecionando repetidamente a ação  $a_0$ , ou se mover para o estado  $s_2$  e obter uma recompensa negativa de -50 (ai!). No estado  $s_2$ , ele não tem outra opção senão tomar a ação  $a_1$ , o que provavelmente o levará de volta ao estado  $s_0$ , ganhando uma recompensa de +40 no caminho. Você pegou o espírito. Ao olhar para esse MDP, você consegue adivinhar qual estratégia obterá mais recompensa ao longo do tempo? No estado  $s_0$ , é claro que a ação  $a_0$  é a melhor opção e no estado  $s_2$  o agente não tem escolha a não ser tomar a ação  $a_1$ , mas no estado  $s_1$  não é óbvio se o agente deve ficar parado ( $a_0$ ) ou passar pelo fogo ( $a_2$ ).

<sup>11</sup> “A Markovian Decision Process”, R. Bellman (1957).

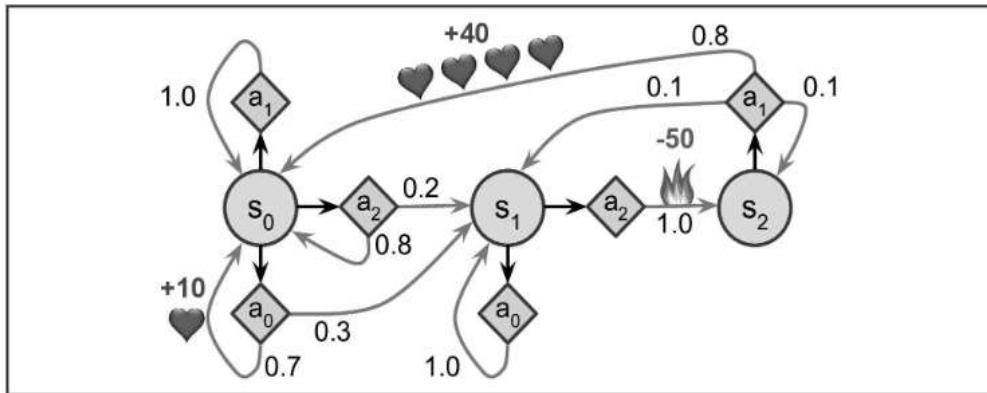


Figura 16-8. Exemplo de um processo de decisão de Markov

Bellman encontrou uma forma de estimar o *valor ideal do estado* de qualquer estado  $s$ , descrito  $V^*(s)$ , que é a soma de todas as recompensas futuras descontadas que o agente pode esperar, em média, após atingir um estado  $s$ , supondo que ele atue de forma otimizada. Ele mostrou que, se o agente age de maneira ideal, então a *Equação de Otimização de Bellman* se aplica (veja Equação 16-1). Esta equação recursiva diz que, se o agente age de forma ideal, o valor ideal do estado atual é igual à recompensa que ele obterá, em média, após tomar uma ação otimizada mais o valor ideal esperado de todos os próximos estados possíveis que essa ação possa levar.

*Equação 16-1. Equação de Otimização de Bellman*

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{para todo } s$$

- $T(s, a, s')$  é a probabilidade de transição do estado  $s$  para o estado  $s'$  dado que o agente escolheu a ação  $a$ ;
- $R(s, a, s')$  é a recompensa que o agente obtém quando passa do estado  $s$  para o estado  $s'$ , dado que tenha escolhido a ação  $a$ ;
- $\gamma$  é a taxa de desconto.

Essa equação leva diretamente a um algoritmo que pode estimar com precisão o valor do estado ideal de todos os estados possíveis: primeiro você inicializa todas as estimativas de valor de estado para zero e, utilizando o algoritmo de *Iteração de Valor*, as atualiza iterativamente (veja a Equação 16-2). Um resultado notável é que, com tempo suficiente, essas estimativas garantem convergir para os valores ótimos de estado, correspondendo à política otimizada.

*Equação 16-2. Algoritmo de Iteração de Valor*

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{para todo } s$$

- $V_k(s)$  é o valor estimado do estado  $s$  na  $k$ -ésima iteração do algoritmo.



Este algoritmo é um exemplo de *Programação Dinâmica* que divide um problema complexo (neste caso, estimando uma soma potencialmente infinita de recompensas futuras descontadas) em subproblemas tratáveis que podem ser abordados iterativamente (neste caso, encontrar a ação que maximiza a recompensa média mais o valor do próximo estado descontado).

Conhecer os valores ideais de estado pode ser útil, principalmente para avaliar uma política, mas não informa ao agente explicitamente o que fazer. Felizmente, Bellman encontrou um algoritmo muito semelhante para estimar *valores ótimos de estado-ação*, geralmente chamados *Q-Values*. O ótimo Q-Value do par estado-ação  $(s, a)$ , identificado como  $Q^*(s, a)$ , é a soma das recompensas futuras descontadas que o agente pode esperar, em média, depois de atingir o estado  $s$  e escolher a ação  $a$ , mas antes de ver o resultado desta ação, supondo que ela funcione de maneira ideal após essa ação.

Veja como funciona: mais uma vez, você começa inicializando todas as estimativas de Q-Value em zero e, em seguida, as atualiza com o algoritmo de *Iteração Q-Value* (veja a Equação 16-3).

#### *Equação 16-3. Algoritmo de Iteração Q-Value*

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot \max_a Q_k(s', a)] \quad \text{para todo } (s, a)$$

Definir a política ótima notada como  $\pi^*(s)$ , é trivial depois de ter os Q-Values ideais: quando o agente está no estado  $s$ , ele deve escolher a ação com o Q-Value mais alto para esse estado:  $\pi^*(s) = \underset{a}{\operatorname{argmax}} Q^*(s, a)$

Aplicaremos este algoritmo ao MDP representado na Figura 16-8. Primeiro, precisamos defini-lo:

```
nan = np.nan # representa ações impossíveis
T = np.array([
    # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
    [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
])
R = np.array([
    # shape=[s, a, s']
    [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
    [[0.0, 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
    [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Agora, executaremos o algoritmo de Iteração Q-Value:

```

Q = np.full((3, 3), -np.inf) # -inf para ações impossíveis
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Valor inicial = 0.0, para todas as ações possíveis

discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp]))
                for sp in range(3)
            ])
    ]
)

```

Os Q-Values resultantes se parecem com isso:

```

>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
       [ 1.11669335,      -inf,   1.17573546],
       [      -inf,  53.86946068,      -inf]])
>>> np.argmax(Q, axis=1) # ação ótima para cada estado
array([0, 2, 1])

```

Temos, portanto, a política ideal para este MDP quando utilizarmos uma taxa de desconto de 0,95: no estado  $s_0$ , escolha a ação  $a_0$ , no estado  $s_1$  escolha a ação  $a_2$  (passe pelo fogo!) e, no estado  $s_2$ , escolha a ação  $a_1$  (a única ação possível). Curiosamente, a política ótima muda se você reduzir a taxa de desconto para 0,9: no estado  $s_1$ , a melhor ação se torna  $a_0$  (fique aí; não passe pelo fogo). Faz sentido porque, se você valoriza o presente muito mais que o futuro, então a perspectiva de recompensas futuras não vale a pena.

## Aprendizado de Diferenças Temporais e Q-Learning

Problemas de Aprendizado por Reforço com ações discretas podem frequentemente ser modelados como processos de decisão de Markov, mas o agente inicialmente não tem ideia de quais são as probabilidades de transição ( $T(s, a, s')$ ), e também não sabe quais serão as recompensas ( $R(s, a, s')$ ). Ele deve experimentar cada estado e cada transição pelo menos uma vez para conhecer as recompensas, e deve experimentá-las várias vezes se quiser ter uma estimativa razoável das probabilidades de transição.

O algoritmo *Aprendizado por Diferença Temporal* (TD Learning) é muito semelhante ao algoritmo de Iteração de Valor, mas é ajustado para levar em conta o fato de o agente ter conhecimento apenas parcial do MDP. Em geral, assumimos que o agente inicialmente

conhece apenas os possíveis estados e ações, e nada mais. O agente utiliza uma *política de exploração* — por exemplo, uma política puramente aleatória — para explorar o MDP e, à medida que avança, o algoritmo TD Learning atualiza as estimativas dos valores de estado com base nas transições e recompensas realmente observadas (veja Equação 16-4).

*Equação 16-4. Algoritmo TD Learning*

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

- $\alpha$  é a taxa de aprendizado (por exemplo, 0,01).



O TD Learning tem muitas semelhanças com o Gradiente Descendente Estocástico, em especial o fato de manipular uma amostra de cada vez. Assim como o SGD, ele só pode realmente convergir se você reduzir gradualmente a taxa de aprendizado (caso contrário, ele continuará saltando em torno do ótimo).

Para cada estado  $s$ , além das recompensas que espera obter mais tarde (supondo que ele funcione de maneira ideal), este algoritmo simplesmente mantém o controle de uma média de execução das recompensas imediatas que o agente obtém ao sair desse estado.

Da mesma forma, o algoritmo Q-Learning é uma adaptação do algoritmo de iteração Q-Value para a situação em que as probabilidades de transição e as recompensas são inicialmente desconhecidas (veja a Equação 16-5).

*Equação 16-5. Algoritmo Q-Learning*

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha\left(r + \gamma \cdot \max_{a'} Q_k(s', a')\right)$$

Para cada par estado-ação  $(s, a)$ , além das recompensas que ele espera obter mais tarde, esse algoritmo controla a média de execução das recompensas  $r$  que o agente obtém ao deixar o estado  $s$  com a ação  $a$ . Uma vez que a política-alvo agiria de maneira otimizada, tomamos o máximo das estimativas de Q-value para o próximo estado.

O Q-Learning pode ser implementado assim:

```

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # começa no estado 0

Q = np.full((3, 3), -np.inf) # -inf para ações impossíveis
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Valor inicial = 0.0, para todas as ações possíveis

for iteration in range(n_iterations):
    a = np.random.choice(possible_actions[s]) # escolhe uma ação (aleatoriamente)
    sp = np.random.choice(range(3), p=T[s, a]) # escolhe o próximo estado usando T[s, a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = ((1 - learning_rate) * Q[s, a] +
               learning_rate * (reward + discount_rate * np.max(Q[sp])))
    s = sp # muda para o próximo estado

```

Com base em iterações suficientes, esse algoritmo convergirá para os Q-Values ideais, o que é chamado de algoritmo *off-policy* porque a política que está sendo treinada não é a que está sendo executada. É um pouco surpreendente que esse algoritmo seja capaz de aprender a política ótima observando um agente agir aleatoriamente (imagine aprender a jogar golfe quando seu professor é um macaco bêbado). Podemos melhorar?

## Políticas de Exploração

É claro que o Q-Learning só funciona se a política de exploração explorar o MDP minuciosamente. Embora uma política de visita puramente aleatória muitas vezes seja garantida em todos os estados e todas as transições, pode levar um tempo extremamente longo para fazer isso. Portanto, uma melhor opção seria utilizar a  *$\epsilon$ -greedy policy*: a cada passo, ela age aleatoriamente com probabilidade  $\epsilon$ , ou avidamente (escolhendo a ação com o maior Q-Value) com probabilidade  $1-\epsilon$ . A vantagem da política  $\epsilon$ -greedy (comparada a uma política completamente aleatória) é que ela gastará cada vez mais tempo explorando as partes interessantes do ambiente à medida que as estimativas de Q-Value ficarem melhores, enquanto ainda passa algum tempo visitando regiões desconhecidas do MDP. É bastante comum começar com um valor alto para  $\epsilon$  (por exemplo, 1,0) e depois reduzi-lo gradualmente (por exemplo, até 0,05).

Uma alternativa seria encorajar a política de exploração para tentar ações que ela não tentou antes, em vez de depender do acaso para a exploração, uma abordagem que pode ser implementada como um bônus adicionado às estimativas do Q-value, conforme mostrado na Equação 16-6.

*Equação 16-6. Q-Learning utilizando uma função de exploração*

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left( r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a')) \right)$$

- $N(s', a')$  conta o número de vezes que a ação  $a'$  foi escolhida no estado  $s'$ ;
- $f(q, n)$  é uma função de exploração, como  $f(q, n) = q + K/(1 + n)$ , em que  $K$  é um hiperparâmetro curioso que mede quanto o agente é atraído para o desconhecido.

## Q-Learning Aproximado e Deep Q-Learning

O principal problema com o Q-Learning é ele não se adaptar bem a grandes MDPs (ou mesmo médias) com muitos estados e ações. Considere a tentativa de usar o Q-Learning para treinar um agente para jogar Ms. Pac-Man. Existem mais de 250 pelotas que Ms. Pac-Man pode comer, cada uma das quais pode estar presente ou ausente (isto é, já consumida). Então, o número de estados possíveis é maior que  $2^{250} \approx 10^{75}$  (e isso considerando os possíveis estados apenas das pelotas), valor muito mais alto do que o número de átomos na nossa galáxia, então não há absolutamente nenhuma forma de rastrear uma estimativa para cada valor de Q.

A solução, chamada *Q-Learning Aproximado*, é encontrar uma função  $Q_\theta(s, a)$  utilizando um número gerenciável de parâmetros (dado pelo vetor de parâmetro  $\theta$ ) que se aproxime do valor Q de qualquer par ação-estado  $(s, a)$ . Durante anos, foi recomendado o uso de combinações lineares de recursos manuais extraídos do estado (por exemplo, distância dos fantasmas mais próximos, suas direções e assim por diante) para estimar Q-Values, mas a DeepMind mostrou que o uso de redes neurais profundas pode funcionar muito melhor, especialmente para problemas complexos, e não requer nenhuma engenharia de recursos. Uma DNN utilizada para estimar Q-Values é chamada *deep Q-network* (DQN) e utilizar uma DQN com *Q-Learning Aproximado* é chamado *Deep Q-Learning*.

Agora, como podemos treinar uma DQN? Bem, considere o Q-Value aproximado calculado pela DQN para um determinado par estado-ação  $(s, a)$ . Graças a Bellman, sabemos que queremos que esse Q-Value aproximado seja o mais próximo possível da recompensa  $r$  que nós realmente observamos depois de jogarmos a ação  $a$  no estado  $s$ , mais o valor descontado de jogar otimamente a partir de então. Para estimar este valor descontado futuro, podemos simplesmente executar a DQN no próximo estado  $s'$  e para todas possíveis ações  $a'$ , obtendo um Q-Value futuro aproximado para cada ação possível. Então escolhemos o maior (já que assumimos que estaremos jogando otimamente), o descontamos, e isso

nos dá uma estimativa do valor descontado futuro. Somando a recompensa  $r$  e o valor descontado futuro estimado, obtemos um Q-Value alvo  $y(s, a)$  para o par estado-ação  $(s, a)$ , como mostrado na Equação 16-7.

*Equação 16-7. Alvo Q-Value*

$$y(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

Com esse Q-Value desejado, podemos executar uma iteração de treinamento utilizando qualquer algoritmo de Gradiente Descendente. Especificamente, tentamos minimizar o erro quadrado entre o Q-Value estimado e o Q-Value desejado. E isso é tudo para o algoritmo básico do Deep Q-Learning!

No entanto, foram introduzidas duas modificações cruciais no algoritmo DQN da DeepMind:

- Em vez de treinar a DQN com base nas experiências mais recentes, o algoritmo DQN da DeepMind armazena experiências em uma grande *memória de repetição* e faz a amostragem de um lote de treinamento aleatório em cada iteração do treinamento, ajudando a reduzir as correlações entre as experiências em um lote de treinamento, o que é um tremendo auxílio;
- O algoritmo utiliza dois DQNs em vez de um: o primeiro, chamado *DQN online*, é aquele que reproduz e aprende a cada iteração de treinamento. O segundo, chamado de *DQN de destino*, é utilizado apenas para calcular os Q-Values de destino (Equação 16-7). Os pesos da rede online são copiados para a rede de destino em intervalos regulares. A DeepMind mostrou que essa mudança melhora drasticamente o desempenho do algoritmo. De fato, sem isso existe uma única rede que estabelece seus próprios alvos e tenta alcançá-los, um pouco como um cachorro correndo atrás da própria cauda, o que pode levar a loops de feedback, tornando a rede instável (pode divergir, oscilar, congelar e assim por diante). Ter duas redes ajuda a reduzir esses loops de feedback, o que estabiliza o processo de treinamento.

No restante deste capítulo, utilizaremos o algoritmo DQN da DeepMind para treinar um agente para jogar Ms. Pac-Man, assim como fizeram em 2013. O código pode ser facilmente ajustado para aprender a jogar muito bem a maioria dos jogos de Atari, desde que você treine por tempo suficiente (pode levar dias ou semanas, dependendo do seu hardware). Ele pode conseguir uma habilidade sobre-humana na maioria dos jogos de ação, mas não é tão bom em jogos com histórias de longa duração.

## Aprendendo a Jogar Ms. Pac-Man com a Utilização do Algoritmo DQN

Como utilizaremos um ambiente Atari, devemos primeiro instalar as dependências Atari do OpenAI gym. Enquanto estivermos nisso, também instalaremos dependências para outros ambientes do OpenAI gym com os quais você pode querer brincar. No macOS, supondo que você tenha instalado o Homebrew (<http://brew.sh/>), é preciso executar:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

No Ubuntu, digite o seguinte comando (substituindo `python3` por `python` se você estiver utilizando Python 2):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-dev\xvfb libav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev swig
```

Em seguida, instale os módulos extras do Python (se estiver utilizando um virtualenv, certifique-se de ativá-lo primeiro):

```
$ pip3 install --upgrade 'gym[all]'
```

Se tudo correr bem, você deve conseguir criar um ambiente Ms. Pac-Man:

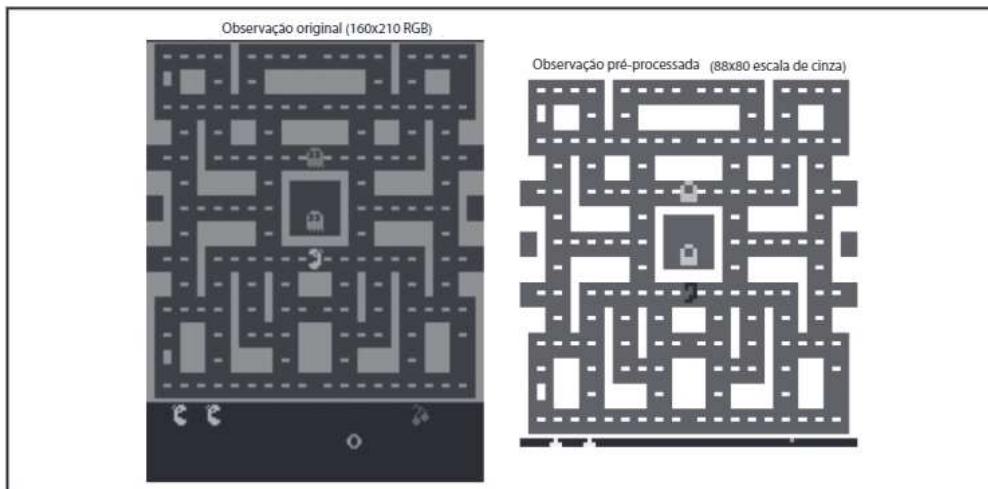
```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape # [altura, largura, canais]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

Como podemos ver, existem nove ações discretas disponíveis que correspondem às nove posições possíveis do controle (centro, cima, direita, esquerda, baixo, canto superior direito e assim por diante) e as observações são simplesmente capturas de tela na tela do Atari, (veja a Figura 16-9, à esquerda) representada como matrizes 3D NumPy. Essas imagens são um pouco grandes, então criaremos uma pequena função de pré-processamento que recortará a imagem e a reduzirá para 88x80 pixels, a converterá em tons de cinza e melhorará o contraste da Ms. Pac-Man, reduzindo a quantidade de cálculos exigidos pelo DQN e acelerando o treinamento.

```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # cortar e reduzir
    img = img.mean(axis=2) # para escalas de cinza
    img[img==mspacman_color] = 0 # melhorar contraste
    img = (img - 128) / 128 - 1 # normalizar de -1. para 1.
    return img.reshape(88, 80, 1)
```

O resultado do pré-processamento é mostrado na Figura 16-9 (direita).



*Figura 16-9. Observação da Ms. Pac-Man, original (esquerda) e após o pré-processamento (direita)*

Em seguida, criaremos a DQN, que poderia pegar apenas um par estado-ação ( $s, a$ ) como entrada e produzir uma estimativa do Q-Value correspondente  $Q(s, a)$ , mas, como as ações são discretas, é mais conveniente e eficiente utilizar uma rede neural que usa apenas um estado  $s$  como entrada e gera uma estimativa de Q-Value por ação. O DQN será composto de três camadas convolucionais, seguidas por duas camadas totalmente conectadas, incluindo a camada de saída (veja a Figura 16-10).



*Figura 16-10. Q-network profunda para jogar Ms. Pac-Man*

Como discutimos anteriormente, o algoritmo de treinamento DQN projetado pela DeepMind requer dois DQNs com a mesma arquitetura (mas parâmetros diferentes): o DQN online aprenderá a conduzir a Ms. Pac-Man e o DQN de destino será utilizado para construir Q-Values de destino para treinar o DQN online. Substituindo seus parâmetros em intervalos regulares, copiaremos o DQN online para o DQN de destino. Como precisamos de dois DQNs com a mesma arquitetura, criaremos uma função `q_network()` para construí-los:

```

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"] * 3
conv_activation = [tf.nn.relu] * 3
n_hidden_in = 64 * 11 * 10 # conv3 tem 64 mapas de 11x10
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n # 9 ações discretas estão disponíveis
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, name):
    prev_layer = X_state
    with tf.variable_scope(name) as scope:
        for n_maps, kernel_size, strides, padding, activation in zip(
            conv_n_maps, conv_kernel_sizes, conv_strides,
            conv_paddings, conv_activation):
            prev_layer = tf.layers.conv2d(
                prev_layer, filters=n_maps, kernel_size=kernel_size,
                strides=strides, padding=padding, activation=activation,
                kernel_initializer=initializer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
        hidden = tf.layers.dense(last_conv_layer_flat, n_hidden,
                               activation=hidden_activation,
                               kernel_initializer=initializer)
        outputs = tf.layers.dense(hidden, n_outputs,
                               kernel_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                      scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                             for var in trainable_vars}
    return outputs, trainable_vars_by_name

```

A primeira parte deste código define os hiperparâmetros da arquitetura DQN. Em seguida, considerando o estado do ambiente `X_state` como entrada e o nome do escopo da variável, definimos a função `q_network()` para criar os DQNs. Observe que utilizaremos apenas uma observação para representar o estado do ambiente, já que quase não há estado oculto algum (exceto para objetos que piscam e as direções dos fantasmas).



Jogos como Pong ou Breakout contêm uma bola em movimento cuja direção e velocidade não podem ser determinadas com uma única observação, portanto, exigiriam a combinação das últimas observações no estado do ambiente. Uma maneira de fazer isso seria criar uma imagem com um canal para cada uma das últimas observações. Como alternativa, poderíamos mesclar as últimas observações em uma imagem de canal único, por exemplo, calculando o máximo dessas observações (depois de escurecer as observações mais anti-gas, de modo que a direção do tempo esteja clara na imagem final).

O dicionário `trainable_vars_by_name` reúne todas as variáveis treináveis deste DQN e será útil em um minuto quando criarmos operações para copiar o DQN online para o DQN de destino. Retirando a parte do prefixo que corresponde apenas ao nome do escopo as chaves do dicionário são os nomes das variáveis. Fica mais ou menos assim:

```
>>> trainable_vars_by_name
{'/conv2d/bias:0': <tf.Variable... shape=(32,) dtype=float32_ref>,
 '/conv2d/kernel:0': <tf.Variable... shape=(8, 8, 1, 32) dtype=float32_ref>,
 '/conv2d_1/bias:0': <tf.Variable... shape=(64,) dtype=float32_ref>,
 '/conv2d_1/kernel:0': <tf.Variable... shape=(4, 4, 32, 64) dtype=float32_ref>,
 '/conv2d_2/bias:0': <tf.Variable... shape=(64,) dtype=float32_ref>,
 '/conv2d_2/kernel:0': <tf.Variable... shape=(3, 3, 64, 64) dtype=float32_ref>,
 '/dense/bias:0': <tf.Variable... shape=(512,) dtype=float32_ref>,
 '/dense/kernel:0': <tf.Variable... shape=(7040, 512) dtype=float32_ref>,
 '/dense_1/bias:0': <tf.Variable... shape=(9,) dtype=float32_ref>,
 '/dense_1/kernel:0': <tf.Variable... shape=(512, 9) dtype=float32_ref>}
```

Agora, criaremos o placeholder de entrada, os dois DQNs e a operação para copiar o DQN online para o DQN de destino:

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                             input_channels])
online_q_values, online_vars = q_network(X_state, name="q_networks/online")
target_q_values, target_vars = q_network(X_state, name="q_networks/target")

copy_ops = [target_var.assign(online_vars[var_name])
           for var_name, target_var in target_vars.items()]
copy_online_to_target = tf.group(*copy_ops)
```

Voltaremos por um segundo: agora temos dois DQNs capazes de tomar um estado do ambiente como entrada (neste exemplo, uma única observação pré-processada) e gerar um Q-Value estimado para cada ação possível nesse estado. Além disso, temos uma operação chamada `copy_online_to_target` para copiar os valores de todas as variáveis treináveis da DQN online para as variáveis da DQN de destino correspondentes. Utilizaremos a função `tf.group()` do TensorFlow para agrupar todas as operações de atribuição em uma única operação conveniente.

Agora, adicionaremos as operações de treinamento da DQN online. Primeiro, precisamos calcular seu Q-Value previsto para cada par estado-ação no lote de memória. Como a DQN gera um Q-Value para cada ação possível, precisamos manter apenas o que corresponde à ação que foi realmente executada. Para isto, converteremos a ação para um vetor one-hot (lembre-se de que este é um vetor cheio de 0s, exceto 1 no índice  $i$ ) e multiplicaremos pelos Q-Values: o que zerará todos, exceto aquele correspondente à ação memorizada. Em seguida, basta somar o primeiro eixo para obter apenas a previsão desejada do Q-Value para cada memória.

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(target_q_values * tf.one_hot(X_action, n_outputs),
                        axis=1, keep_dims=True)
```

Em seguida, criamos um placeholder  $y$  que utilizaremos para fornecer os Q-Values de destino e, em seguida, calcularemos a perda: utilizamos o erro quadrado quando ele for menor que 1,0 e o dobro do erro absoluto quando o erro quadrado for maior que 1,0. Em outras palavras, a perda é quadrática para pequenos erros e linear para os grandes, reduzindo o efeito dos grandes erros e ajudando a estabilizar o treinamento.

```
y = tf.placeholder(tf.float32, shape=[None, 1])
error = tf.abs(y - q_value)
clipped_error = tf.clip_by_value(error, 0.0, 1.0)
linear_error = 2 * (error - clipped_error)
loss = tf.reduce_mean(tf.square(clipped_error) + linear_error)
```

Por fim, para minimizar a perda, criamos um otimizador *Gradiente Acelerado de Nesterov*. Também para acompanhar a etapa de treinamento, criamos uma variável não treinável chamada `global_step`. A operação de treinamento cuidará de incrementá-lo. Além disso, criamos a operação usual do `init` e um `Saver`.

```
learning_rate = 0.001
momentum = 0.95

global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum, use_nesterov=True)
training_op = optimizer.minimize(loss, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Isto é tudo para a fase de construção e, antes de olharmos para a de execução, precisaremos de algumas ferramentas. Primeiro, começaremos implementando a memória de repetição. Utilizaremos uma lista `deque` já que é muito eficiente em empurrar itens para a fila e retirar os mais antigos quando o tamanho máximo da memória for atingido. Também escreveremos uma pequena função para amostrar aleatoriamente um lote de experiências da memória de repetição. Cada experiência será uma tupla quíntupla (estado, ação, recompensa, próximo estado, continuar), em que o item “continuar” será igual a 0,0 quando o jogo terminar, ou 1,0 caso contrário.

```

from collections import deque

replay_memory_size = 500000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = np.random.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # state, action, reward, next_state, continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],
            cols[4].reshape(-1, 1))

```

Em seguida, precisaremos do agente para explorar o jogo. Utilizaremos a política  $\epsilon$ -greedy, e gradualmente diminuiremos  $\epsilon$  de 1,0 para 0,1 em duas milhões de etapas de treinamento:

```

eps_min = 0.1
eps_max = 1.0
eps_decay_steps = 2000000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # random action
    else:
        return np.argmax(q_values) # optimal action

```

É isso aí! Temos tudo o que precisamos para começar a treinar. A fase de execução não contém nada muito complexo, mas é um pouco longa, então respire fundo. Pronto? Definiremos primeiro alguns parâmetros:

```

n_steps = 4000000 # número total de etapas de treinamento
training_start = 10000 # começa a treinar depois de 10 mil iterações
training_interval = 4 # toda uma etapa de treinamento a cada 4 iterações
save_steps = 1000 # salva o modelo a cada mil etapas de treinamento
copy_steps = 10000 # copia a DQN online para a DQN-alvo a cada 10 mil etapas de treinamento
discount_rate = 0.99
skip_start = 90 # Pula o começo do jogo (é tempo de espera)
batch_size = 50
iteration = 0 # iterações do jogo
checkpoint_path = "./my_dqn.ckpt"
done = True # env precisa ser resetado

```

Em seguida, abriremos a sessão e executaremos o loop de treinamento principal:

```

with tf.Session() as sess:
    if os.path.isfile(checkpoint_path + ".index"):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()

```

```

        copy_online_to_target.run()
while True:
    step = global_step.eval()
    if step >= n_steps:
        break
    iteration += 1
    if done: # game over, começa novamente
        obs = env.reset()
        for skip in range(skip_start): # pula o começo de cada jogo
            obs, reward, done, info = env.step(0)
        state = preprocess_observation(obs)

    # DQN online avalia o que fazer
    q_values = online_q_values.eval(feed_dict={X_state: [state]})
    action = epsilon_greedy(q_values, step)

    # DQN online joga
    obs, reward, done, info = env.step(action)
    next_state = preprocess_observation(obs)

    # Vamos memorizar o que acabou de acontecer
    replay_memory.append((state, action, reward, next_state, 1.0 - done))
    state = next_state

    if iteration < training_start or iteration % training_interval != 0:
        continue # apenas treine depois do período de aquecimento e em um intervalo regular

    # Memórias amostrais e usa a DQN-alvo para produzir o Q-Value alvo
    X_state_val, X_action_val, rewards, X_next_state_val, continues = (
        sample_memories(batch_size))
    next_q_values = target_q_values.eval(
        feed_dict={X_state: X_next_state_val})
    max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
    y_val = rewards + continues * discount_rate * max_next_q_values

    # Treina a DQN online
    training_op.run(feed_dict={X_state: X_state_val,
                               X_action: X_action_val, y: y_val})

    # Copia regularmente a DQN online para a DQN-alvo
    if step % copy_steps == 0:
        copy_online_to_target.run()

    # E salva regularmente
    if step % save_steps == 0:
        saver.save(sess, checkpoint_path)

```

Começamos restaurando o modelo se existir um arquivo de ponto de verificação, ou então apenas inicializamos as variáveis normalmente e copiamos a DQN online para a DQN de destino. Em seguida, o loop principal começa, a `iteration` conta o número total de etapas que passamos desde o início do programa e `step` conta o número total de etapas de treinamento desde o início (graças à variável `global_step`, se um ponto de verificação for restaurado, a etapa também será restaurada). Então o código reinicia o jogo e pula as primeiras etapas entediantes em que nada acontece. Em seguida, a DQN online avalia o que fazer, joga o jogo e sua experiência é gravada na memória de repetição. A seguir, em intervalos regulares (após um período de aquecimento), a DQN online passa por uma etapa de treinamento. Primeiro, amostramos um lote de memórias e pedimos à DQN alvo para estimar os Q-Values de todas as ações possíveis para cada “próximo estado” da memória,

depois aplicamos a Equação 16-7 para calcular `y_val`, que contém o Q-Value de destino para cada par estado-ação. A única parte complicada aqui é que devemos multiplicar o vetor `max_next_q_values` pelo vetor `continues` para zerar os valores futuros que correspondem às memórias onde o jogo terminou. Em seguida, realizamos uma operação de treinamento para melhorar a capacidade da DQN online de prever Q-Values. Finalmente, em intervalos regulares, copiamos a DQN online para a DQN de destino e salvamos o modelo.



Infelizmente, o treinamento é muito lento: se você utilizar seu notebook, levará dias até que a Ms. Pac-Man fique boa. Você pode plotar curvas de aprendizado, por exemplo, medindo as recompensas médias por jogo ou calculando o Q-Value máximo estimado pela DQN online em cada etapa do jogo e rastreando a média desses Q-Values máximos em cada jogo. Você notará que essas curvas são extremamente ruidosas e, em alguns pontos, pode não haver progresso aparente por um longo tempo até que, de repente, o agente aprenda a sobreviver por um período razoável. Como mencionado anteriormente, uma solução seria injetar o máximo de conhecimento prévio possível no modelo (por exemplo, por meio do pré-processamento, recompensas e assim por diante), e você também pode tentar inicializar o modelo primeiro treinando-o para imitar uma estratégia básica. Em qualquer caso, a RL ainda requer muita paciência e ajustes, mas o resultado final é empolgante.

## Exercícios

1. Como você definiria o Aprendizado por Reforço? Como ele é diferente do aprendizado regular supervisionado ou não supervisionado?
2. Você consegue pensar em três possíveis aplicações do RL que não foram mencionadas neste capítulo? Qual é o ambiente para cada uma delas? Qual é o agente? Quais são as ações possíveis? Quais são as recompensas?
3. Qual é a taxa de desconto? A política ótima pode mudar se você modificar a taxa de desconto?
4. Como você mede o desempenho de um agente do Aprendizado por Reforço?
5. Qual é o problema da atribuição de crédito? Quando ocorre? Como pode ser aliviado?
6. Qual é o objetivo de utilizar uma memória de repetição?
7. O que é um algoritmo de RL *off-policy*?
8. Utilize gradientes de políticas para enfrentar o “BipedalWalker-v2” da OpenAI gym.

9. Utilize o algoritmo DQN para treinar um agente para jogar *Pong*, o famoso jogo do Atari ([Pong-v0](#) no OpenAI gym). Cuidado: uma observação individual é insuficiente para dizer a direção e a velocidade da bola.
10. Se você tem cerca de US\$100 de sobra, pode comprar um Raspberry Pi 3 e alguns componentes robóticos baratos, instalar o TensorFlow no Pi e enlouquecer! Por exemplo, confira este post divertido (<https://goo.gl/Eu5u28>) de Lukas Biewald ou dê uma olhada no GoPiGo ou no BrickPi. Por que não tentar construir um carrinho que ande equilibrando um bastão na vida real treinando o robô com a utilização de gradientes de política? Ou construir uma aranha robótica que aprenda a andar; ofereça recompensas no momento que ela se aproximar de algum objetivo (você precisará de sensores para medir a distância até o objetivo). O único limite é a sua imaginação.

Soluções para estes exercícios estão disponíveis no Apêndice A.

## Obrigado!

Antes de fechamos o último capítulo, gostaria de agradecer a você pela leitura até o último parágrafo. Realmente espero que você tenha tido tanto prazer em ler este livro quanto eu tive de escrevê-lo, e que ele seja útil para seus projetos, grandes ou pequenos.

Se você encontrar erros, envie um feedback. Mais genericamente, adoraria saber o que você pensa, então, por favor, não hesite em entrar em contato comigo via O'Reilly, através do [ageron/handson-ml](#) projeto GitHub ou no Twitter em @aureliengeron.

Indo além, meu melhor conselho para você é praticar e praticar: resolva todos os exercícios se você ainda não o fez, brinque com os notebooks do Jupyter, junte-se ao Kaggle.com ou outra comunidade do AM, assista cursos de AM, leia artigos, participe de conferências, conheça especialistas. Você também pode estudar alguns tópicos que não abordamos neste livro, incluindo sistemas de recomendação, algoritmos de agrupamento, de detecção de anomalias e genéticos.

Minha maior esperança é que este livro inspire você a construir um maravilhoso aplicativo AM que beneficiará todos nós! O que será?

*Aurélien Géron, 26 de Novembro de 2016*



## Apêndice A

# Soluções dos Exercícios



Soluções para os exercícios de codificação estão disponíveis nos notebooks online do Jupyter em <https://github.com/ageron/handson-ml>.

## Capítulo 1: O Cenário do Aprendizado de Máquina

1. Aprendizado de Máquina refere-se à construção de sistemas que podem aprender com os dados. Dada alguma medida de desempenho, aprender significa melhorar em alguma tarefa.
2. O Aprendizado de Máquina é ótimo para problemas complexos para os quais não temos solução algorítmica, para substituir longas listas de regras ajustadas à mão, para construir sistemas que se adaptam a ambientes flutuantes e, finalmente, para ajudar humanos a aprender (por exemplo, mineração de dados).
3. Um conjunto de treinamento rotulado é um conjunto de treinamento que contém a solução desejada (também conhecida como um rótulo) para cada instância.
4. As duas tarefas supervisionadas mais comuns são a regressão e a classificação.
5. Tarefas comuns não supervisionadas incluem clustering, visualização, redução da dimensionalidade e aprendizado de regras de associação.
6. O Aprendizado por Reforço provavelmente funcionará melhor se quisermos que um robô aprenda a andar em vários terrenos desconhecidos, já que esse normalmente é o tipo de problema que o Aprendizado por Reforço aborda. Poderia ser possível expressar o problema como um problema de aprendizado supervisionado ou semissupervisionado, mas seria menos natural.
7. Se você não sabe como definir os grupos, utilize um algoritmo de clustering (aprendizado não supervisionado) para segmentar seus clientes em grupos de clientes

semelhantes. No entanto, será possível alimentar muitos exemplos de cada grupo com um algoritmo de classificação (aprendizado supervisionado) e classificar todos os seus clientes nesses grupos se souber quais grupos você gostaria de ter.

8. A detecção de spam é um problema típico do aprendizado supervisionado: o algoritmo é alimentado com muitos e-mails juntamente com seu rótulo (spam ou não spam).
9. Um sistema de aprendizado online pode aprender de forma incremental, ao contrário de um sistema de aprendizado em lote, fazendo com que seja capaz de se adaptar rapidamente às mudanças de dados e a sistemas autônomos de treinamento em grandes quantidades de dados.
10. Algoritmos out-of-core podem lidar com grandes quantidades de dados que não cabem na memória principal de um computador. Um algoritmo de aprendizado out-of-core divide os dados em minilotes e utiliza técnicas de aprendizado online para aprender com esses minilotes.
11. Um sistema de aprendizado baseado em instância decora os dados de treinamento; em seguida, utiliza uma medida de similaridade para encontrar as instâncias aprendidas mais semelhantes e as utiliza para fazer previsões quando recebe uma nova instância.
12. Dada uma nova instância, um modelo tem um ou mais parâmetros do modelo que determinam o que ele preverá (por exemplo, a inclinação de um modelo linear). Um algoritmo de aprendizado tenta encontrar valores ótimos para esses parâmetros de modo que o modelo generalize para novas instâncias. Um hiperparâmetro é um parâmetro do próprio algoritmo de aprendizado, não do modelo (por exemplo, a quantidade de regularização a ser aplicada).
13. Os algoritmos de aprendizado baseados em modelo buscam um valor ideal para os parâmetros do modelo de modo que o modelo generalize para novas instâncias. Geralmente, treinamos esses sistemas minimizando uma função de custo que mede o quanto mal o sistema faz previsões sobre os dados de treinamento, além de uma penalidade pela complexidade do modelo, se for regularizado. Inserimos os recursos da nova instância na função de previsão do modelo utilizando os valores de parâmetro encontrados pelo algoritmo de aprendizado para fazer previsões.
14. Alguns dos principais desafios no Aprendizado de Máquina são a falta de dados, a baixa qualidade de dados, dados não representativos, recursos não informativos, modelos excessivamente simples que servem de base aos dados de treinamento e modelos excessivamente complexos que superam os dados.
15. Se um modelo exibe um ótimo desempenho nos dados de treinamento, mas generaliza mal para novas instâncias, provavelmente está se sobreajustando aos dados de treinamento (ou tivemos muita sorte nos dados de treinamento). As possíveis soluções para o sobreajuste são: obter mais dados, simplificar o modelo (selecionar

um algoritmo mais simples, reduzir o número de parâmetros ou características utilizadas ou regularizar o modelo) ou reduzir o ruído nos dados de treinamento.

16. Um conjunto de testes é utilizado para estimar o erro de generalização que um modelo fará em novas instâncias antes de ser lançado em produção.
17. Um conjunto de validação é utilizado para comparar modelos. Permite selecionar o melhor e ajustar os hiperparâmetros.
18. Se você ajustar os hiperparâmetros com a utilização do conjunto de testes, corre o risco de sobreajustar o conjunto de testes e o erro de generalização medido será otimista (você pode lançar um modelo com desempenho pior do que o esperado).
19. A validação cruzada é uma técnica que permite comparar modelos (para a seleção de modelo e ajuste de hiperparâmetro) sem a necessidade de um conjunto de validação separado, economizando preciosos dados de treinamento.

## Capítulo 2: Projeto de Aprendizado de Máquina de Ponta a Ponta

Veja os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 3: Classificação

Veja os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 4: Treinando Modelos

1. Se você tiver um conjunto de treinamento com milhões de características, e se o conjunto de treinamento couber na memória, poderá utilizar o Gradiente Descendente Estocástico ou o Gradiente Descendente em Minilote ou, talvez, o Gradiente Descendente em Lote. Mas você não poderá utilizar o Método dos Mínimos Quadrados porque a complexidade dos cálculos cresce rapidamente (mais do que quadraticamente) devido ao número de características.
2. Se as características em seu conjunto de treinamento tiverem escalas muito diferentes, a função de custo terá a forma de uma tigela alongada, portanto, os algoritmos de Gradiente Descendente demorarão muito para convergir. Para resolver isso, você deve escalar os dados antes de treinar o modelo. Observe que o Método dos Mínimos Quadrados funcionará bem sem escalação. Além disso, modelos regularizados podem convergir para uma solução subótima se as carac-

terísticas não forem dimensionadas: na verdade, as características com valores menores tenderão a ser ignoradas quando comparadas com as características com valores maiores, pois a regularização penaliza grandes pesos.

3. Ao treinar um modelo de Regressão Logística, o Gradiente Descendente não poderá ficar preso em um mínimo local porque a função de custo é convexa.<sup>1</sup>
4. Se o problema de otimização for convexo (como Regressão Linear ou Regressão Logística), e assumindo que a taxa de aprendizado não é muito alta, todos os algoritmos do Gradiente Descendente se aproximam do ótimo global e acabarão produzindo modelos bastante similares. No entanto, a menos que você reduza gradualmente a taxa de aprendizado, o GD Estocástico e o GD de Minilote nunca convergirão de verdade; em vez disso, continuarão pulando de um lado para outro em torno do ótimo global. Isso significa que, mesmo se você deixá-los rodar por um longo período, esses algoritmos do Gradiente Descendente produzirão modelos ligeiramente diferentes.
5. Se o erro de validação crescer consistentemente após cada época, então uma possibilidade é que a taxa de aprendizado seja muito alta e o algoritmo seja divergente. Se o erro de treinamento também aumentar, então este é claramente o problema e você deve reduzir sua taxa de aprendizado. No entanto, se o erro não estiver aumentando, então seu modelo está se sobreajustando ao conjunto de treinamento e você deve parar o treinamento.
6. Devido à sua natureza aleatória, nem o Gradiente Descendente Estocástico nem o Gradiente Descendente de Minilote garantem o progresso em cada iteração do treinamento. Então, ao parar de treinar, imediatamente após o erro de validação aumentar, você pode estar parando cedo demais, antes que o ótimo seja alcançado. Uma melhor opção seria salvar o modelo em intervalos regulares e, não havendo melhora por um longo período de tempo (o que significa que provavelmente nunca baterá o recorde), você poderá reverter para o melhor modelo salvo.
7. O Gradiente Descendente Estocástico possui a iteração de treinamento mais rápida, pois considera apenas uma instância de treinamento de cada vez, então geralmente é o primeiro a alcançar a vizinhança do ótimo global (ou GD de Minilote com um minilote muito pequeno). No entanto, dado o tempo de treinamento necessário, apenas o Gradiente Descendente em Lote convergirá. Como mencionado, o GD Estocástico e o GD de Minilote rebaterão perto do ótimo, a menos que você reduza gradualmente sua taxa de aprendizado.
8. Se o erro de validação for muito maior do que o erro de treinamento, provavelmente será porque o seu modelo está se sobreajustando ao conjunto de treinamento. Uma maneira de tentar consertar isso seria reduzir o grau do polinômio:

---

<sup>1</sup> Se você desenhar uma linha reta entre dois pontos da curva, a linha nunca cruza a curva.

um modelo com menos graus de liberdade é menos propenso a se sobreajustar. Outra coisa que você pode tentar seria regularizar o modelo — por exemplo, adicionar uma penalidade  $\ell_2$  (Ridge) ou uma penalidade  $\ell_1$  (Lasso) à função de custo. Isso também reduzirá os graus de liberdade do modelo. Por fim, você pode tentar aumentar o tamanho do conjunto de treinamento.

9. Se tanto o erro de treinamento quanto o erro de validação forem quase iguais e razoavelmente altos, o modelo provavelmente está se subajustando ao conjunto de treinamento, o que significa que ele tem um alto viés e você deve tentar reduzir o hiperparâmetro de regularização  $\alpha$ .

#### 10. Vejamos:

- Normalmente, um modelo com alguma regularização tem um desempenho melhor do que um modelo sem qualquer regularização, portanto, você deve preferir a Regressão de Ridge em relação à Regressão Linear simples.<sup>2</sup>
  - A Regressão Lasso utiliza uma penalidade  $\ell_1$  que tende a empurrar os pesos para, exatamente, zero, levando a modelos esparsos nos quais todos os pesos valem zero, exceto para os mais importantes. Essa é uma forma de executar automaticamente a seleção das características, o que é bom, caso suspeite que apenas algumas realmente importam. Se não tiver certeza, prefira a Regressão de Ridge.
  - Já que o Lasso pode se comportar erraticamente em alguns casos (quando várias características são fortemente correlacionadas ou quando há mais características do que instâncias de treinamento), a Elastic Net geralmente é preferida. No entanto, ela adiciona um hiperparâmetro extra de ajuste. Se você quer apenas o Lasso sem o comportamento errático, pode simplesmente utilizar o Elastic Net com um `l1_ratio` perto de 1.
11. Você deve treinar dois classificadores de Regressão Logística se quiser classificar imagens como externas/internas e diurnas/noturnas, uma vez que estas não são classes exclusivas (ou seja, todas as quatro combinações são possíveis).
  12. Veja os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 5: Máquinas de Vetores de Suporte

1. A ideia fundamental por trás das Máquinas de Vetores de Suporte é adequar a “via” mais larga possível entre as classes. Em outras palavras, o objetivo é ter a maior margem possível entre a fronteira de decisão, que separa as duas classes, e as instâncias de treinamento. Ao realizar a classificação de margem suave, a SVM

---

<sup>2</sup> Além disso, o Método dos Mínimos Quadrados requer o cálculo do inverso de uma matriz, mas essa matriz nem sempre é inversível. Em contraste, a matriz de Regressão de Ridge é sempre inversível.

procura um compromisso entre a separação perfeita das duas classes e a existência da via mais ampla possível (ou seja, algumas instâncias podem acabar na via). Outra ideia chave é utilizar kernels ao treinar conjuntos de dados não lineares.

2. Um vetor de suporte é qualquer instância localizada na “via” após o treinamento de uma SVM (veja a resposta anterior), incluindo sua borda. A fronteira de decisão é inteiramente determinada pelos vetores de suporte. Qualquer instância que não seja um vetor de suporte (isto é, fora da via) não terá influência alguma; você pode removê-los, adicionar mais instâncias ou movê-los e, desde que permaneçam fora da via, eles não afetarão a fronteira de decisão. O cálculo das previsões envolve apenas os vetores de suporte, não todo o conjunto de treinamento.
3. As SVM tentam ajustar a maior “via” possível entre as classes (veja a primeira resposta), portanto, se o conjunto de treinamento não for escalonado, a SVM tenderá a negligenciar pequenas características (veja a Figura 5-2).
4. Um classificador SVM pode gerar a distância entre a instância de teste e o limite de decisão, e você pode utilizar isso como uma pontuação de confiança. No entanto, essa pontuação não poderá ser convertida diretamente em uma estimativa da probabilidade da classe. Se você configurar `probability=True` quando criar um SVM no Scikit-Learn, então, depois do treinamento, ele calibrará as probabilidades usando Regressão Logística nas pontuações da SVM (treinadas por uma validação cruzada de cinco dobradas nos dados de treinamento), o que adicionará os métodos `predict_proba()` e `predict_log_proba()` à SVM.
5. Esta questão se aplica apenas às SVM lineares, já que a kernelizada só pode utilizar a forma dual. A complexidade dos cálculos do problema SVM da forma primal é proporcional ao número de instâncias de treinamento  $m$ , enquanto a complexidade dos cálculos da forma dual é proporcional a um número entre  $m^2$  e  $m^3$ . Então, caso existam milhões de instâncias, você definitivamente deveria utilizar a forma primal, porque a forma dual será muito lenta.
6. Caso um classificador SVM treinado com um kernel RBF se subajuste ao conjunto de treinamento, poderá haver muita regularização. Para diminuí-la, você precisará aumentar `gamma` ou `C` (ou ambos).
7. Chamaremos os parâmetros QP para o problema de hard-margin  $\mathbf{H}'$ ,  $\mathbf{f}'$ ,  $\mathbf{A}'$  e  $\mathbf{b}'$  (veja “Programação Quadrática” no Capítulo 5). Os parâmetros QP para o problema da soft-margin tem  $m$  parâmetros adicionais ( $n_p = n + 1 + m$ ),  $m$  restrições adicionais ( $n_c = 2m$ ) e podem ser definidos como:
  - $\mathbf{H}$  é igual a  $\mathbf{H}'$ , mais  $m$  colunas de 0s à direita e  $m$  linhas de 0s no final:  $\mathbf{H} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} & \cdots \\ \mathbf{0} & \mathbf{0} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$
  - $\mathbf{f}$  é igual a  $\mathbf{f}'$  com  $m$  elementos adicionais, tudo igual ao valor do hiperparâmetro  $C$ .

- $\mathbf{b}$  é igual a  $\mathbf{b}'$  com  $m$  elementos adicionais, tudo igual a 0.
- $\mathbf{A}$  é igual a  $\mathbf{A}'$ , com uma matriz de identidade  $\mathbf{I}_m$  extra  $m \times m$  anexada a direita,  
–  $\mathbf{I}_m$  abaixo dela, e o restante preenchido com zeros:  $\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}$

Para as soluções dos exercícios 8, 9 e 10, consulte os notebooks do Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 6: Árvores de Decisão

1. A profundidade de uma árvore binária bem balanceada contendo  $m$  folhas é igual a  $\log_2(m)$ <sup>3</sup>, arredondado para cima. Uma Árvore de Decisão binária (que toma apenas decisões binárias, como é o caso de todas as árvores no Scikit-Learn) ficará mais ou menos equilibrada ao final do treinamento, com uma folha por instância de treinamento se for treinada sem restrições. Assim, se o conjunto de treinamento contiver um milhão de instâncias, a Árvore de Decisão terá uma profundidade de  $\log_2(10^6) \approx 20$  (na verdade, um pouco mais, uma vez que a árvore geralmente não estará perfeitamente equilibrada).
2. O coeficiente de Gini de um nó é geralmente menor que a de seus pais devido à função de custo do algoritmo de treinamento CART, que divide cada nó de maneira a minimizar a soma ponderada dos coeficientes de Gini de seus filhos. No entanto, é possível que um nó tenha um coeficiente de Gini mais alto do que seu pai, desde que esse aumento seja mais do que compensado pela diminuição da impureza do outro filho. Por exemplo, considere um nó contendo quatro instâncias da classe A e 1 da classe B, seu coeficiente de Gini é  $1 - \frac{1^2}{5} - \frac{4^2}{5} = 0.32$ . Agora, suponha que o conjunto de dados seja unidimensional e as instâncias estejam alinhadas na seguinte ordem: A, B, A, A, A. Verifique que o algoritmo dividirá este nó após a segunda instância produzindo um nó filho com instâncias A, B, e o outro nó filho com instâncias A, A, A. O primeiro coeficiente de Gini do nó filho é  $1 - \frac{1^2}{2} - \frac{1^2}{2} = 0.5$ , que é mais alto do que o dos seus pais. Isto é compensado pelo fato que o outro nó é puro, então o coeficiente ponderado geral Gini é  $\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0 = 0.2$ , que é inferior ao coeficiente de Gini do pai.
3. Pode ser uma boa ideia diminuir o `max_depth` se uma Árvore de Decisão estiver se sobreajustando ao conjunto de treinamento, pois isso restringirá o modelo, o regularizando.
4. As Árvores de Decisão não se importam se os dados de treinamento são escalados ou não; esse é um de seus pontos positivos, portanto, será um desperdício de

---

<sup>3</sup>  $\log_2$  é o log binário,  $\log_2(m) = \log(m) / \log(2)$ .

tempo apenas dimensionar os recursos de entrada se uma Árvore de Decisão se subajustar ao conjunto de treinamento.

5. A complexidade dos cálculos do treinamento de uma Árvore de Decisão é  $O(n \times m \log(m))$ . Então, se você multiplicar o tamanho do conjunto de treinamento por 10, o tempo de treinamento será multiplicado por  $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$ . Se  $m = 10^6$ , então  $K \approx 11,7$ , você pode esperar que o tempo de treinamento seja de aproximadamente 11,7 horas.
6. A pré-qualificação do conjunto de treinamento acelera o treinamento somente se o conjunto de dados for menor que alguns milhares de instâncias. Definir `presort=True` diminuirá consideravelmente o treinamento se contiver 100 mil instâncias.

Para as soluções dos exercícios 7 e 8, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 7: Ensemble Learning e Florestas Aleatórias

1. Você pode tentar combiná-los em um ensemble de votação se treinou cinco modelos diferentes e todos alcançaram 95% de precisão, o que geralmente lhe dará resultados ainda melhores. Ele funciona melhor se os modelos forem muito diferentes (por exemplo, um classificador SVM, um de Árvore de Decisão, um de Regressão Logística e assim por diante). Fica ainda melhor se eles forem treinados em diferentes instâncias de treinamento (esse é o objetivo de ensacar [bagging] e colar [pasting] ensembles), mas ainda funcionará caso não sejam treinados, desde que os modelos sejam muito diferentes.
2. Um classificador hard voting apenas conta os votos de cada classificador no ensemble e escolhe a classe que obtém o maior número de votos. Um classificador soft voting calcula a probabilidade da classe média estimada para cada classe e escolhe a com a maior probabilidade. Isso dá votos de alta confiança, mais peso, e geralmente tem melhor desempenho, mas funciona somente se cada classificador for capaz de estimar as probabilidades da classe (por exemplo, para os classificadores SVM no Scikit-Learn, você deve configurar `probability=True`).
3. É bem possível acelerar o treinamento de um conjunto de ensacamento [bagging ensemble] distribuindo-o por vários servidores já que cada previsor no conjunto é independente dos outros. Pelo motivo semelhante, o mesmo vale para pasting ensembles e Florestas Aleatórias. No entanto, cada previsor em um ensemble de reforço é criado com base no previsor anterior, portanto o treinamento é necessariamente sequencial e você não ganhará nada o distribuindo em vários servidores. Em relação aos stacking ensembles, todos os previsores em uma determinada camada são independentes uns dos outros para que possam ser treinados em paralelo em vários servidores. No entanto, os previsores em

uma camada só podem ser treinados depois que todos os previsores da camada anterior tiverem sido treinados.

4. Com a avaliação out-of-bag, cada previsor em um bagging ensemble é avaliado com a utilização de instâncias nas quais ele não foi treinado (eles foram mantidos de fora), possibilitando uma avaliação bastante imparcial do ensemble sem a necessidade de um ensemble adicional de validação. Assim, você tem mais instâncias disponíveis para treinamento, e seu ensemble pode ter um desempenho melhor.
5. Será considerado apenas um subconjunto aleatório das características para a divisão em cada nó quando você está cultivando uma árvore em uma Floresta Aleatória. Isso também vale para as Árvores-extra, mas elas vão além: em vez de procurar os melhores limiares possíveis, como acontece com as árvores de decisão comuns, elas utilizam limiares aleatórios para cada característica. Essa aleatoriedade extra age como uma forma de regularização: as Árvores-extras podem ter um melhor desempenho se uma Floresta Aleatória se sobreajusta aos dados de treinamento. Além disso, como as Árvores-extras não buscam os melhores limiares possíveis, elas são muito mais rápidas para treinar do que as Florestas Aleatórias. No entanto, quando fazem previsões, elas não são nem mais rápidas nem mais lentas que as Florestas Aleatórias.
6. Se o seu ensemble AdaBoost se subajustar aos dados de treinamento você pode tentar aumentar o número de estimadores ou reduzir os hiperparâmetros de regularização do estimador de base. Você também pode tentar aumentar um pouco a taxa de aprendizado.
7. Se o seu ensemble Gradiente Boosting se sobreajusta ao conjunto de treinamento, você deveria tentar diminuir a taxa de aprendizado ou utilizar a parada antecipada para encontrar o número correto de previsores (você provavelmente tem muitos).

Para as soluções dos exercícios 8 e 9, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 8: Redução da Dimensionalidade

1. Motivações e desvantagens:
  - As principais motivações para a redução da dimensionalidade são:
    - Acelerar um algoritmo de treinamento subsequente (em alguns casos, ele pode até mesmo remover ruídos e características redundantes, fazendo com que o algoritmo de treinamento tenha um melhor desempenho);
    - Visualizar os dados e obter insights sobre as características mais importantes;
    - Simplesmente economizar espaço (compressão).

- As principais desvantagens são:
  - Algumas informações são perdidas, possivelmente degradando o desempenho de algoritmos de treinamento subsequentes;
  - Pode ser intensivo computacionalmente;
  - Adiciona alguma complexidade aos seus pipelines de Aprendizado de Máquina;
  - Características transformadas geralmente são de difícil interpretação.
- 2. A maldição da dimensionalidade refere-se ao fato de que muitos problemas que não existem em um espaço de dimensão inferior surgem no espaço de alta dimensão. Uma manifestação comum no aprendizado de máquina é o fato de os vetores de alta dimensão amostrados aleatoriamente geralmente serem muito esparsos, aumentando o risco de sobreajuste e dificultando bastante a identificação de padrões nos dados sem ter muitos dados de treinamento.
- 3. Uma vez que a dimensionalidade foi reduzida do conjunto de dados com a utilização de um dos algoritmos que discutimos, quase sempre é impossível reverter perfeitamente a operação porque algumas informações são perdidas durante a redução de dimensionalidade. Além disso, enquanto alguns algoritmos (como o PCA) têm um procedimento simples de transformação reversa, que pode reconstruir um conjunto de dados relativamente similar ao original, outros algoritmos (como o T-SNE) não tem.
- 4. O PCA pode ser utilizado para reduzir significativamente a dimensionalidade da maioria dos conjuntos de dados mesmo que eles sejam altamente não lineares porque podem pelo menos se livrar de dimensões inúteis. No entanto, se não houver dimensões inúteis, por exemplo, o rolo suíço, então a redução da dimensionalidade com o PCA perderá muita informação. Você quer desenrolar o rolo suíço, não esmagá-lo.
- 5. Essa é uma pergunta complicada: depende do conjunto de dados. Vejamos dois exemplos extremos. Primeiro, suponha que o conjunto de dados seja composto por pontos quase perfeitamente alinhados. Nesse caso, preservando 95% da variância, o PCA pode reduzir o conjunto de dados a apenas uma dimensão. Agora, imagine que o conjunto de dados é composto de pontos perfeitamente aleatórios, espalhados por todas as mil dimensões. Nesse caso, serão necessárias aproximadamente 950 dimensões para preservar 95% da variância. Portanto, a resposta é: depende do conjunto de dados e pode ser qualquer número entre 1 e 950. Plotar a variância explicada como uma função do número de dimensões é uma forma de ter uma ideia aproximada da dimensionalidade intrínseca do conjunto de dados.

6. O PCA comum é o padrão, mas funciona somente se o conjunto de dados couber na memória. O PCA incremental é útil, mas é mais lento do que o PCA normal para conjuntos de dados grandes que não cabem na memória. Portanto, prefira o PCA comum se o conjunto se encaixar na memória. O PCA incremental também será útil para tarefas online quando precisarmos aplicar o PCA em tempo real toda vez que uma nova instância chegar. O PCA Randomizado é útil quando você deseja reduzir consideravelmente a dimensionalidade, e o conjunto de dados se encaixa na memória; neste caso, ele é muito mais rápido que o PCA comum. Finalmente, o Kernel PCA é útil para conjuntos de dados não lineares.
7. Intuitivamente, um algoritmo de redução de dimensionalidade desempenha bem se eliminar muitas dimensões do conjunto de dados sem perder muita informação. Uma maneira de medir isso é aplicar a transformação reversa e medir o erro de reconstrução. No entanto, nem todos os algoritmos de redução de dimensionalidade fornecem uma transformação inversa. Como alternativa, poderá simplesmente medir o desempenho desse segundo algoritmo se você estiver utilizando a redução de dimensionalidade como uma etapa de pré-processamento antes de outro algoritmo de Aprendizado de Máquina (por exemplo, um classificador Floresta Aleatória); se a redução de dimensionalidade não perder muita informação, então o algoritmo deve ter um desempenho tão bom quanto utilizar o conjunto de dados original.
8. Pode sim fazer sentido encadear dois algoritmos de redução de dimensionalidade diferentes. Um exemplo comum é utilizar o PCA para se livrar rapidamente de um grande número de dimensões inúteis e, em seguida, aplicar outro algoritmo de redução de dimensionalidade muito mais lento, como o LLE. Essa abordagem em duas etapas provavelmente produzirá o mesmo desempenho de se utilizar apenas o LLE, mas em uma fração do tempo.

Para as soluções dos exercícios 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 9: Em pleno Funcionamento com o TensorFlow

1. Principais benefícios e desvantagens de se criar um grafo de cálculos em vez de executá-los diretamente:
  - Principais benefícios:
    - O TensorFlow pode calcular automaticamente os gradientes para você (utilizando reverse-mode autodiff);

- O TensorFlow pode cuidar da execução das operações em paralelo em diferentes threads;
  - Isto facilita a execução do mesmo modelo em diferentes dispositivos;
  - Simplifica a introspecção — por exemplo, para visualizar o modelo no TensorBoard.
- Principais desvantagens:
    - Deixa a curva de aprendizado mais íngreme;
    - Dificulta a depuração passo a passo.
2. Sim, a declaração `a_val = a.eval(session=sess)` é equivalente a `a_val = sess.run(a)`.
  3. Não, a declaração `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` não é equivalente a `a_val, b_val = sess.run([a, b])`. Na verdade, a primeira declaração executa o grafo duas vezes (uma vez para calcular a, uma para calcular b), enquanto a segunda instrução executa o grafo apenas uma vez. Se qualquer uma dessas operações (ou as operações das quais dependem) tiverem efeitos colaterais (por exemplo, uma variável é modificada, um item é inserido em uma fila ou um leitor lê um arquivo), então os efeitos serão diferentes. Se elas não tiverem efeitos colaterais, as duas declarações retornarão o mesmo resultado, mas a segunda será mais rápida do que a primeira.
  4. Não, você não pode executar dois grafos na mesma sessão para isso seria necessário primeiro mesclar os grafos em um grafo único.
  5. No TensorFlow local as sessões gerenciam valores de variáveis, então, se você criar um grafo `g` contendo a variável `w` e, em seguida, iniciar dois threads e abrir uma sessão local em cada um, ambos utilizando o mesmo grafo `g`, cada sessão terá sua própria cópia da variável `w`. No entanto, os valores das variáveis são armazenados em contêineres gerenciados pelo cluster no TensorFlow distribuído, portanto, elas compartilharão o mesmo valor de variável para `w` se ambas as sessões se conectarem ao mesmo cluster e utilizarem o mesmo contêiner.
  6. Uma variável é inicializada quando você chama seu inicializador, e é destruída quando a sessão termina. No TensorFlow distribuído, fechar uma sessão não destrói a variável, pois elas vivem em contêineres no cluster. Para destruir uma variável, você precisa limpar seu contêiner.
  7. Variáveis e marcadores são extremamente diferentes, mas os iniciantes costumam confundi-los:
    - Uma variável é uma operação que contém um valor e que irá retorna-lo se você executar a variável, mas, antes de poder executá-la, você precisa inicializá-la.

Você pode alterar o valor da variável (por exemplo, utilizando uma operação de atribuição). Ela é estável: a variável mantém o mesmo valor em sucessivas execuções do grafo. Normalmente, é utilizada para manter parâmetros do modelo, mas também para outros fins (por exemplo, para contar a etapa de treinamento global);

- Eles apenas armazenam informações sobre o tipo e a forma do tensor que representam, mas não têm valor. Na verdade, o TensorFlow deverá ser alimentado com o valor do placeholder se tentar avaliar uma operação que dependa de um placeholder (utilizando o argumento `feed_dict`) ou obterá uma exceção. Normalmente, os placeholders são utilizados para fornecer dados de treinamento ou de testes ao TensorFlow durante a fase de execução e também são úteis para passar um valor para um nó de atribuição, para alterar o valor de uma variável (por exemplo, pesos de modelo).
8. Se você executar o grafo para avaliar uma operação que depende de um placeholder, mas não alimentar seu valor, receberá uma exceção. Nenhuma exceção será levantada se a operação não depender do placeholder.
  9. Quando você executa um grafo, poderá alimentar o valor de saída de qualquer operação, não apenas o valor dos placeholders. Na prática, no entanto, isso é bastante raro (pode ser útil, por exemplo, quando você está armazenando em cache a saída de camadas congeladas; consulte o Capítulo 11).
  10. Você pode especificar o valor inicial de uma variável ao construir o grafo, e ela será inicializada mais tarde, quando você executar o inicializador da variável durante a fase de execução. Se quiser alterar o valor dessa variável para qualquer coisa desejada durante a fase de execução, a opção mais simples é criar um nó de atribuição (durante a fase de construção do grafo) com a utilização da função `tf.assign()` levando a variável e um placeholder a passarem como parâmetros. Durante a fase de execução, você pode executar a operação de atribuição e fornecer o novo valor da variável usando placeholder.

```
import tensorflow as tf

x = tf.Variable(tf.random_uniform(shape=(), minval=0.0, maxval=1.0))
x_new_val = tf.placeholder(shape=(), dtype=tf.float32)
x_assign = tf.assign(x, x_new_val)

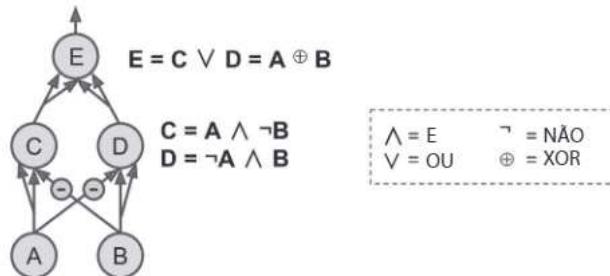
with tf.Session():
    x.initializer.run() # random number is sampled *now*
    print(x.eval()) # 0.646157 (some random number)

    x_assign.eval(feed_dict={x_new_val: 5.0})
    print(x.eval()) # 5.0
```

11. Autodiff no Modo Reverso (implementado pelo TensorFlow) precisa percorrer o grafo apenas duas vezes para calcular os gradientes da função de custo em relação a qualquer número de variáveis. Por outro lado, o Autodiff no forward-mode precisaria ser executado uma vez para cada variável (portanto, 10 vezes se quisermos os gradientes com relação a 10 variáveis diferentes). Quanto à diferenciação simbólica, ela criaria um grafo diferente para calcular os gradientes, de modo que não atravessasse o grafo original (exceto ao construir o novo grafo de gradientes). Um sistema de diferenciação simbólica altamente otimizado poderia executar o novo grafo de gradientes apenas uma vez para calcular os gradientes com relação a todas as variáveis, mas esse novo grafo pode ser terrivelmente complexo e ineficiente em comparação com o grafo original.
12. Veja os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 10: Introdução às Redes Neurais Artificiais

1. Esta é uma rede neural baseada nos neurônios artificiais originais que calculam  $A \oplus B$  ( $\oplus$  representa a OU exclusiva), utilizando o fato de  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ . Existem outras soluções — por exemplo, utilizando o fato de  $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$ , ou o fato de  $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$ , e assim por diante.



2. Um perceptron clássico convergirá somente se o conjunto de dados for linearmente separável e não puder estimar as probabilidades de classe. Por outro lado, um classificador de regressão logística convergirá para uma boa solução, mesmo que o conjunto de dados não seja linearmente separável e gerará probabilidades da classe. Se você alterar a função de ativação do perceptron para a função de ativação logística (ou a função de ativação softmax se houver múltiplos neurônios), e se você treiná-lo utilizando o Gradiente Descendente (ou algum outro algoritmo de otimização minimizando a função custo, tipicamente entropia cruzada), ele se tornará equivalente a um classificador de regressão logística.
3. A função de ativação logística foi um ingrediente-chave no treinamento das primeiras MLPs pois sua derivada é sempre diferente de zero, portanto, o Gradiente Descendente

sempre pode deslizar pela inclinação. Quando a função de ativação é uma função de grau, o Gradiente Descendente não pode se mover, pois não há nenhuma inclinação.

4. A função de degrau, a função logística, a tangente hiperbólica, a unidade linear retificada (veja a Figura 10-8). Veja o Capítulo 11 para outros exemplos, como o ELU e variantes do ReLU.
5. Considerando o MLP descrito na questão: suponha que você tenha um MLP composto de uma camada de entrada com 10 neurônios de passagem, seguido por uma camada oculta com 50 neurônios artificiais e, finalmente, uma camada de saída com 3 neurônios artificiais. Todos os neurônios artificiais utilizam a função de ativação ReLU.
  - A forma da matriz de entrada  $X$  é  $m \times 10$ , sendo que  $m$  representa o tamanho do lote de treinamento;
  - A forma do vetor de peso da camada oculta  $W_h$  é  $10 \times 50$  e o comprimento do seu vetor de viés  $b_h$  é 50;
  - A forma do vetor de peso da camada de saída  $W_o$  é  $50 \times 3$ , e o comprimento do seu vetor de viés  $b_o$  é 3;
  - A forma da matriz de saída da rede  $Y$  é  $m \times 3$ ;
  - $Y = \text{ReLU}(ReLU(X \cdot W_h + b_h) \cdot W_o + b_o)$ . Lembre-se de que a função ReLU apenas define todos os números negativos da matriz como zero. Observe também que, quando você está adicionando um vetor de polarização a uma matriz, ele é adicionado a cada uma de suas linhas, o que é chamado broadcasting.
6. Para classificar o e-mail em spam ou ham, você precisa apenas de um neurônio na camada de saída de uma rede neural — por exemplo, indicando a probabilidade de o e-mail ser spam. Normalmente, você utilizaria a função de ativação logística na camada de saída quando estimasse a probabilidade. Se, em vez disso, você quiser lidar com o MNIST, precisará de 10 neurônios na camada de saída e deve substituir a função logística pela função de ativação softmax que pode manipular múltiplas classes, gerando uma probabilidade por classe. Agora, se você quer que sua rede neural preveja os preços do setor imobiliário como no Capítulo 2, então precisará de um neurônio de saída sem utilizar nenhuma função de ativação na camada de saída.<sup>4</sup>
7. A retropropagação é uma técnica utilizada para treinar redes neurais artificiais que primeiro calcula os gradientes da função de custo em relação a cada parâmetro do modelo (todos os pesos e vieses) e, em seguida, executa uma etapa de Gradiente Des-

---

<sup>4</sup> Quando os valores a serem previstos tendem a variar em muitas ordens de grandeza, você pode optar por predizer o logaritmo do valor de destino em vez do valor de destino diretamente. Simplesmente calcular o exponencial da saída da rede neural fornecerá o valor estimado (desde que  $\exp(\log v) = v$ ).

cedente utilizando esses gradientes. Essa etapa de retropropagação é normalmente executada milhares ou milhões de vezes usando muitos lotes de treinamento até que os parâmetros do modelo converjam para valores que (esperamos) minimizam a função de custo. Para calcular os gradientes, a retropropagação utiliza autodiff no Modo Reverso (embora não tenha sido chamado assim quando a retropropagação foi inventada, e ela já foi reinventada várias vezes). O autodiff no Modo Reverso executa uma passagem direta calculando o valor de cada nó para o lote de treinamento atual através de um grafo de cálculo, e, em seguida, executa uma passagem inversa calculando todos os gradientes de uma vez (consulte o Apêndice D para obter mais detalhes). Então, qual é a diferença? Bem, utilizando várias etapas, a retropropagação refere-se a todo o processo de treinamento de uma rede neural artificial, cada uma das quais calcula gradientes e os utiliza para executar uma etapa do Gradiente Descendente. Em contraste, o autodiff no Modo Reverso é simplesmente uma técnica para calcular gradientes de forma eficiente e, por acaso, é utilizada pela retropropagação.

8. Eis uma lista de todos os hiperparâmetros que você pode ajustar em um MLP básico: o número de camadas ocultas, o número de neurônios em cada camada oculta e a função de ativação utilizada em cada camada oculta e na camada de saída.<sup>5</sup> Em geral, a função de ativação ReLU (ou uma de suas variantes; consulte o Capítulo 11) é um bom padrão para as camadas ocultas. Para a camada de saída, em geral, você desejará a função de ativação logística para classificação binária, a função de ativação softmax para classificação multiclasse ou nenhuma função de ativação para regressão.  
Se o MLP se sobreajusta aos dados de treinamento, você pode tentar reduzir o número de camadas ocultas e reduzir o número de neurônios por camada oculta.
9. Veja os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 11: Treinando Redes Neurais Profundas

1. Não, todos os pesos devem ser amostrados de forma independente; eles não devem todos ter o mesmo valor inicial. Um objetivo importante da amostragem aleatória de pesos é quebrar as simetrias: se todos os pesos tiverem o mesmo valor inicial, mesmo que esse valor não seja zero, a simetria não será quebrada (ou seja, todos os neurônios em uma determinada camada serão equivalentes) e a retropropagação será incapaz de quebrá-la. Concretamente, isso significa que todos os neurônios em qualquer camada terão sempre os mesmos pesos. É como ter apenas

---

<sup>5</sup> No Capítulo 11, discutimos muitas técnicas que introduzem hiperparâmetros adicionais: tipo de inicialização de peso, hiperparâmetros de função de ativação (por exemplo, quantidade de vazamento em leaky ReLUs), limite de Gradient Clipping, tipo de otimizador e seus hiperparâmetros (por exemplo, hiperparâmetros de momento MomentumOptimizer), tipo de regularização para cada camada e os hiperparâmetros de regularização (por exemplo, taxa de dropout ao usar o dropout) e assim por diante.

um neurônio por camada, e muito mais lento. É virtualmente impossível que uma configuração como essa convirja para uma boa solução.

2. Não há problema em inicializar os termos de polarização em zero. Algumas pessoas gostam de inicializa-los como pesos, e tudo bem também; não faz muita diferença.
3. Algumas vantagens da função ELU sobre a função ReLU são:
  - Pode assumir valores negativos, de modo que a saída média dos neurônios em qualquer camada específica é tipicamente mais próxima de 0 do que quando se utiliza a função de ativação ReLU (que nunca produz valores negativos), ajudando a aliviar o problema dos vanishing gradients;
  - Ela sempre tem um derivativo diferente de zero, o que evita o problema das unidades que estão morrendo, o que pode afetar as unidades ReLU;
  - É suave em todos os lugares, ao passo que a inclinação da ReLU pula abruptamente de 0 para 1  $z = 0$ . Uma mudança tão abrupta pode atrasar o Gradiente Descendente, porque ricocheteará em torno de  $z = 0$ .
4. A função de ativação ELU é um bom padrão. Se você precisar que a rede neural seja a mais rápida possível, utilize uma das variantes da leaky ReLU (por exemplo, uma leaky ReLU simples utilizando o valor padrão do hiperparâmetro). A simplicidade da função de ativação ReLU a torna a opção preferida de muitas pessoas, apesar do fato de que elas geralmente são superadas pela ELU e pela leaky ReLU. No entanto, a capacidade da função de ativação ReLU de gerar precisamente zero pode ser útil em alguns casos (por exemplo, consulte o Capítulo 15). Se você precisar gerar um número entre -1 e 1, a tangente hiperbólica (tanh) pode ser útil na camada de saída, mas atualmente não é muito utilizada em camadas ocultas. A função de ativação logística também é útil na camada de saída quando você precisa estimar uma probabilidade (por exemplo, para a classificação binária), mas ela também é raramente usada em camadas ocultas (há exceções — por exemplo, para a camada de codificação de autoencoders variacionais ; veja o Capítulo 15). Finalmente, a função de ativação softmax é útil para produzir probabilidades para classes mutuamente exclusivas na camada de saída, mas, fora isso, raramente é (se alguma vez) utilizada em camadas ocultas.
5. Se você definir o hiperparâmetro `momentum` muito próximo de 1 (por exemplo, 0,99999) ao utilizar um `MomentumOptimizer`, então o algoritmo provavelmente obterá muita velocidade em direção ao mínimo global, mas, então, disparará para além do mínimo devido o seu `momentum`, diminuirá a velocidade e voltará, acelerará novamente, passará novamente, e assim por diante. Ele pode oscilar muitas vezes dessa maneira antes de convergir, então, no geral, demorará muito mais para convergir do que se tiver um valor menor do `momentum`.

6. Uma forma de produzir um modelo esparso (isto é, com a maioria dos pesos igual a zero) é treinar o modelo normalmente e depois zerar pesos minúsculos. Você pode aplicar a regularização  $\ell_1$  durante o treinamento para mais esparsidade, o que empurra o otimizador em direção à dispersão. Uma terceira opção seria combinar a regularização  $\ell_1$  com a dual averaging, utilizando a classe `FTRL Optimizer` do TensorFlow.
7. Sim, o dropout diminui o treinamento, geralmente por um fator de dois. No entanto, uma vez que só é ativado durante o treinamento, não tem impacto na inferência.

Para as soluções dos exercícios 8, 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 12: Distribuindo o TensorFlow Por Dispositivos e Servidores

1. Quando um processo do TensorFlow é iniciado, ele pega toda a memória disponível em todos os dispositivos GPU que são visíveis a ele, então, se você receber um `CUDA_ERROR_OUT_OF_MEMORY` ao iniciar o programa do TensorFlow, isso provavelmente significa que outros processos estão em execução e já pegaram toda a memória em pelo menos um dispositivo GPU visível (provavelmente outro processo do TensorFlow). Uma solução trivial para corrigir esse problema é parar os outros processos e tentar novamente. No entanto, uma opção simples seria dedicar diferentes dispositivos a cada processo se você precisar que todos os processos sejam executados simultaneamente, definindo a variável de ambiente `CUDA_VISIBLE_DEVICES` adequadamente para cada dispositivo. Outra opção é configurar o TensorFlow para capturar apenas parte da memória da GPU, em vez dela toda, criando uma `ConfigProto` configurando sua `gpu_options.per_process_gpu_memory_fraction` para a proporção da memória total que ela deve pegar (por exemplo, 0,4) e utilizar este `ConfigProto` ao abrir uma sessão. Na última opção, configurar o `gpu_options.allow_growth` como `True` é dizer ao TensorFlow para pegar a memória somente quando precisar. No entanto, esta última opção geralmente não é recomendada porque qualquer memória que o TensorFlow agarra nunca é liberada, e é mais difícil garantir um comportamento repetitivo (pode haver condições de corrida dependendo de quais processos começam primeiro, quanta memória precisam durante o treinamento e assim por diante).
2. Ao fixar uma operação em um dispositivo, você está dizendo ao TensorFlow que é aqui que você gostaria que essa operação fosse posicionada. No entanto, algumas restrições podem impedir que o TensorFlow atenda ao seu pedido. Por exemplo, a

operação pode não ter nenhuma implementação (chamada kernel) para esse tipo específico de dispositivo. Neste caso, por padrão, o TensorFlow levantará uma exceção, mas você pode configurá-lo para voltar para a CPU (isso é chamado soft placement). Outro exemplo é uma operação que pode modificar uma variável; esta operação e a variável precisam ser feitas em conjunto. Portanto, a diferença entre fixar e colocar uma operação é que ao fixar você pede ao TensorFlow (“Coloque essa operação no GPU 1”), enquanto o posicionamento é o que ele faz (“Desculpe, voltando ao CPU”).

3. Se você está rodando em uma instalação do TensorFlow habilitada para GPU e você utiliza apenas o posicionamento padrão, então se todas as operações tiverem um kernel GPU (ou seja, uma implementação da GPU), sim, elas serão posicionadas na primeira GPU. No entanto, se uma ou mais operações não tiverem um kernel GPU, por padrão, o TensorFlow levantará uma exceção. Se você configurar o TensorFlow para retornar à CPU (soft placement), então todas as operações serão posicionadas na primeira GPU, exceto aquelas sem um kernel GPU e todas as operações posicionadas em conjunto com elas (veja a resposta no exercício anterior).
4. Sim, se você fixar uma variável em `/gpu:0`, ela pode ser utilizada por operações localizadas em `/gpu:1`. O TensorFlow automaticamente cuidará de adicionar as operações apropriadas para transferir o valor das variáveis pelos dispositivos. O mesmo vale para dispositivos localizados em diferentes servidores (desde que façam parte do mesmo cluster).
5. Sim, duas operações posicionadas no mesmo dispositivo podem ser executadas em paralelo: o TensorFlow cuida automaticamente das operações em execução em paralelo (em diferentes núcleos de CPU ou diferentes encadeamentos de GPU), desde que nenhuma delas dependa da saída da outra operação. Além disso, você pode iniciar várias sessões em encadeamentos paralelos (ou processos) e avaliar operações em cada encadeamento. Como as sessões são independentes, o TensorFlow poderá avaliar qualquer operação de uma sessão em paralelo com qualquer operação de outra sessão.
6. Dependências de controle são utilizadas quando você quer adiar a avaliação de uma operação X até que outras operações sejam executadas, mesmo que não sejam necessárias para calcular X. Isto é útil principalmente quando X ocupar muita memória e você somente precisaria disso mais tarde no grafo de cálculo, ou se o X utiliza muita E/S (por exemplo, requer um grande valor de variável localizado em um dispositivo ou servidor diferente) e você não quer que ele seja executado ao mesmo tempo de outras operações famintas por E/S, para evitar a saturação da largura de banda.
7. Você está com sorte! No TensorFlow distribuído, os valores das variáveis residem em contêineres gerenciados pelo cluster, portanto, mesmo se você fechar a sessão e

sair do programa cliente, os parâmetros do modelo ainda estarão ativos e estarão bem no cluster. Basta abrir uma nova sessão no cluster e salvar o modelo (certifique-se de não chamar os inicializadores de variável nem de restaurar um modelo anterior, pois isso destruiria seu novo e precioso modelo!).

Para as soluções dos exercícios 8, 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 13: Redes Neurais Convolucionais (CNN)

1. Estas são as principais vantagens de uma CNN sobre uma DNN totalmente conectada para classificação de imagem:
  - Como as camadas consecutivas são parcialmente conectadas, e porque reutiliza seus pesos, uma CNN tem muito menos parâmetros do que uma DNN totalmente conectada, o que torna seu treinamento muito mais rápido, reduz o risco de sobreajuste e exige muito menos dados de treinamento;
  - Quando uma CNN aprendeu um kernel que detecta uma determinada característica, ela pode detectá-la em qualquer lugar da imagem. Em contraste, quando uma DNN aprende uma característica em um local, ela pode detectá-la apenas neste local específico. Como as imagens geralmente têm características muito repetitivas, as CNNs são capazes de generalizar muito melhor do que as DNNs em tarefas de processamento de imagens, utilizando menos exemplos de treinamento como a classificação.
  - Finalmente, uma DNN não tem conhecimento prévio de como os pixels são organizados; não sabe que os pixels próximos estão próximos. A arquitetura da CNN incorpora esse conhecimento anterior. As camadas inferiores geralmente identificam características em pequenas áreas das imagens, enquanto as camadas superiores combinam as características de nível inferior em características maiores, o que funciona bem com a maioria das imagens naturais, dando às CNNs uma vantagem decisiva em comparação às DNNs.
2. Calcularemos quantos parâmetros a CNN possui. Como sua primeira camada convolucional tem kernels  $3 \times 3$  e a entrada tem três canais (vermelho, verde e azul), cada mapa de características tem pesos  $3 \times 3 \times 3$ , além de um termo de polarização. São 28 parâmetros por mapa de características. Como essa primeira camada convolucional possui 100 mapas de características, ela tem um total de 2.800 parâmetros. A segunda camada convolucional tem kernels  $3 \times 3$  e sua entrada é o conjunto de 100 mapas de características da camada anterior, então cada mapa tem  $3 \times 3 \times 100 = 900$  pesos, mais um termo de polarização. Como possui 200 mapas de características, essa camada tem  $900 \times 200 = 180.200$  parâmetros. Por fim, a terceira e última

camada convolucional também possui kernels  $3 \times 3$  e sua entrada é o conjunto de 200 mapas das camadas anteriores, portanto cada mapa de características tem pesos  $3 \times 3 \times 200 = 1.800$ , além de um termo de polarização. Como possui 400 mapas, essa camada tem um total de  $1.801 \times 400 = 720.400$  parâmetros. Ao todo, a CNN tem  $2.800 + 180.200 + 720.400 = 903.400$  parâmetros.

Agora, calcularemos a quantidade de memória RAM que essa rede neural exigirá (pelo menos) ao fazer uma previsão para uma única instância. Primeiro, calcularemos o tamanho do mapa de características para cada camada e, como estamos usando um incremento de 2 e o padding SAME, os tamanhos horizontal e vertical dos mapas são divididos por 2 em cada camada (arredondados, se necessário), assim como os canais de entrada são  $200 \times 300$  pixels, o primeiro mapa de características da camada são  $100 \times 150$ , os mapas da segunda camada são  $50 \times 75$  e os mapas da terceira camada são  $25 \times 38$ . Como 32 bits são 4 bytes e a primeira camada convolucional possui 100 mapas de características, essa primeira camada ocupa  $4 \times 100 \times 150 \times 100 = 6$  milhões de bytes (cerca de 5,7 MB, considerando que 1 MB = 1.024 KB e 1 KB = 1.024 bytes). A segunda camada ocupa  $4 \times 50 \times 75 \times 200 = 3$  milhões de bytes (cerca de 2,9 MB). Finalmente, a terceira camada ocupa  $4 \times 25 \times 38 \times 400 = 1.520.000$  bytes (cerca de 1,4 MB). No entanto, uma vez que uma camada tenha sido calculada, a memória ocupada pela anterior pode ser liberada, então, se tudo estiver bem otimizado, apenas  $6 + 3 = 9$  milhões de bytes (cerca de 8,6 MB) de RAM serão necessários (quando a segunda camada acaba de ser calculada, mas a memória ocupada pela primeira camada ainda não foi liberada). Mas, espere, também é preciso adicionar a memória ocupada pelos parâmetros da CNN. Calculamos anteriormente que ele tem 903.400 parâmetros, cada um utilizando até 4 bytes, então isso adiciona 3.613.600 bytes (cerca de 3,4 MB). A RAM total requerida é de (no mínimo) 12.613.600 bytes (cerca de 12 MB).

Por fim, calcularemos a quantidade mínima de RAM necessária ao treinamento da CNN em um minilote de 50 imagens. Durante o treinamento, o TensorFlow utiliza a retropropagação, que requer manter todos os valores calculados durante o forward pass até que o reverse pass comece. Portanto, deve-se calcular o total de RAM exigido por todas as camadas para uma única instância e multiplicar por 50!. Nesse ponto, começaremos a contar em megabytes em vez de bytes. Calculamos antes que as três camadas requerem respectivamente 5,7, 2,9 e 1,4 MB para cada instância, resultando no total de 10,0 MB por instância. Portanto, para 50 instâncias, a RAM total é de 500 MB. Acrescente a isso a RAM exigida pelas imagens de entrada, que é  $50 \times 4 \times 200 \times 300 \times 3 = 36$  milhões de bytes (cerca de 34,3 MB), mais a RAM necessária para os parâmetros do modelo, que é de cerca de 3,4 MB (calculado anteriormente), mais alguma RAM para os gradientes (vamos negligenciá-los, uma vez que eles podem ser liberados gradualmente à medida que a retropropagação desce pelas camadas durante o reverse

pass). Nós temos até aqui um total de aproximadamente  $500,0 + 34,3 + 3,4 = 537,7$  MB, que é realmente o mínimo necessário em uma previsão otimista.

3. Cinco coisas que você poderia tentar para resolver o problema se sua GPU ficar sem memória durante o treinamento de uma CNN (além de comprar uma GPU com mais RAM):
  - Reduzir o tamanho do minilote;
  - Reduzir a dimensionalidade utilizando um grande incremento em uma ou mais camadas;
  - Remover uma ou mais camadas;
  - Utilizar flutuadores de 16-bits em vez de 32-bits;
  - Distribuir a CNN através de múltiplos dispositivos.
4. Uma camada max pooling não tem parâmetros, enquanto uma camada convolucional tem alguns (veja as perguntas anteriores).
5. Uma *camada normalização de resposta local* faz com que os neurônios que mais ativam neurônios inibidos no mesmo local, mas em mapas de características vizinhos, incentivem diferentes mapas a se especializarem e a separá-los, forçando-os a explorar uma gama maior de características. É normalmente utilizado nas camadas inferiores para ter um conjunto maior de características de baixo nível sobre as quais as camadas superiores podem construir.
6. As principais inovações na AlexNet em comparação com a LeNet-5 são (1) ela é muito maior e mais profunda e (2) empilha camadas convolucionais diretamente umas sobre as outras, em vez de empilhar uma camada pooling sobre cada camada convolucional. A principal inovação no GoogLeNet é a introdução dos módulos inception, que possibilitaram a existência de uma rede muito mais profunda que as arquiteturas CNN anteriores, com menos parâmetros. Por fim, a principal inovação da ResNet é a introdução de conexões ignoradas, que permitem ultrapassar 100 camadas. Indiscutivelmente, a sua simplicidade e consistência também são bastante inovadoras.

Para as soluções dos exercícios 7, 8, 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>

## Capítulo 14: Redes Neurais Recorrentes (RNN)

1. Algumas aplicações da RNN são:
  - Para uma RNN sequência-para-sequência: previsão do tempo (ou de qualquer outra série temporal), tradução automática (utilizando uma arquitetura *codi-*

(*codificador-decodificador*), legendagem de vídeo, texto falado, geração de música (ou outra geração de sequência), identificando os acordes de uma música;

- Para uma RNN sequência-para-vetor: classificar amostras de música por gênero musical, analisar o sentimento de uma resenha de livro, prever em que palavra um paciente afásico está pensando com base em leituras de implantes cerebrais, prever a probabilidade de um usuário querer assistir um filme baseado em sua experiência (esta é uma das muitas implementações possíveis do *collaborative filtering*);
  - Para uma RNN de vetor-para-sequência: legendagem de imagens, criar uma lista de reprodução de música com base na incorporação do artista atual, gerando uma melodia com base em um conjunto de parâmetros, localizando pedestres em uma imagem (por exemplo, um frame de vídeo de câmera do carro).
2. Em geral, se você traduzir uma frase uma palavra por vez, o resultado será terrível. Por exemplo, a frase francesa “Je vous en prie” significa “por nada”, mas, se você traduz uma palavra de cada vez, você tem “eu você em oração”. Hum? É muito melhor ler a sentença inteira primeiro e depois traduzi-la. Uma RNN simples de sequência-para-sequência iniciaria a tradução de uma sentença imediatamente após a leitura da primeira palavra, enquanto um codificador-decodificador RNN leria primeiro a frase inteira e depois a traduziria. Dito isto, pode-se imaginar uma RNN simples de sequência-para-sequência que produziria silêncio sempre que não tiver certeza sobre o que dizer em seguida (assim como os tradutores humanos fazem quando precisam traduzir uma transmissão ao vivo).
  3. Para classificar vídeos com base no conteúdo visual, uma arquitetura possível poderia ser pegar (digamos) um quadro por segundo, em seguida, executar cada quadro por meio de uma rede neural convolucional, alimentar a saída da CNN para uma RNN sequência-para-vetor e, finalmente, executar sua saída através de uma camada softmax, dando a você todas as probabilidades de classe. Para o treinamento, você utilizaria a entropia cruzada como função de custo. Se quiser utilizar o áudio para classificação também, seria possível converter cada segundo de áudio para um espetrógrafo, alimentar este espetrógrafo para uma CNN e alimentar a saída dessa CNN para a RNN (juntamente com a saída correspondente da outra CNN).
  4. Construir uma RNN utilizando `dynamic_rnn()` em vez de `static_rnn()` tem muitas vantagens:
    - Ele é baseado em uma operação `while_loop()` que é capaz de trocar a memória da GPU para a memória da CPU durante a repropagação, evitando erros de falta de memória;
    - É indiscutivelmente mais fácil de usar, já que pode levar diretamente um único tensor como entrada e saída (cobrindo todos os intervalos de tempo) em vez

de uma lista de tensores (um por intervalo de tempo). Não há necessidade de empilhar, desempilhar ou transpor.

- Ele gera um gráfico menor, mais fácil de visualizar no TensorBoard.
5. A opção mais simples é definir o parâmetro `sequence_length` para lidar com sequências de entrada de comprimento variável quando recorrer às funções `static_rnn()` ou `dynamic_rnn()`. Outra opção é preencher as entradas menores (por exemplo, com zeros) para torná-las do mesmo tamanho da maior entrada (o que pode ser mais rápido que a primeira opção se as sequências de entrada tiverem comprimentos muito semelhantes). Se você souber antecipadamente o tamanho de cada sequência de saída para lidar com sequências de saída de tamanho variável, utilize o parâmetro `sequence_length` (por exemplo, considere uma RNN sequência-para-sequência que rotule cada frame em um vídeo com uma pontuação de violência: a sequência de saída será exatamente do mesmo comprimento que a sequência de entrada). Se você não souber antecipadamente o comprimento da sequência de saída, poderá utilizar o truque de preenchimento: sempre imprima a mesma sequência de tamanho, mas ignore todas as saídas que vêm após o token de fim-de-sequência (ignorando-as ao calcular função de custo).
  6. Para distribuir o treinamento e a execução de uma RNN profunda em várias GPUs, uma técnica comum é colocar cada camada em uma GPU diferente (consulte o Capítulo 12)

Para as soluções dos exercícios 7, 8 e 9, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 15: Autoencoders

1. Veja algumas das principais tarefas utilizadas pelos autoencoders:
  - Extração de características;
  - Pré-treinamento não supervisionado;
  - Redução da Dimensionalidade;
  - Modelos geradores;
  - Detecção de anomalias (um autoencoder normalmente é ruim para a reconstrução de outliers).
2. Se você quiser treinar um classificador e tiver muitos dados de treinamento não rotulados, mas apenas alguns minilotes de instâncias rotuladas, então você poderia primeiro treinar um autoencoder profundo no conjunto de dados completo (rotulado + não rotulado) e, então, reutilizar sua metade inferior para o classificador (isto é, reutilizar as camadas até a camada de codificação, inclusive) e treinar o

classificador utilizando os dados rotulados. Se tiver poucos dados rotulados, provavelmente desejará congelar as camadas reutilizadas ao treinar o classificador.

3. O fato de um autoencoder reconstruir perfeitamente suas entradas não significa necessariamente que seja um bom autoencoder; talvez seja simplesmente um autoencoder supercompleto que aprendeu a copiar suas entradas para a camada de codificação e depois para as saídas. De fato, mesmo se a camada de codificações contivesse um único neurônio, seria possível que um autoencoder muito profundo aprendesse a mapear cada instância de treinamento para uma codificação diferente (por exemplo, a primeira instância poderia ser mapeada para 0,001, a segunda para 0,002, a terceira para 0,003, e assim por diante), e ele poderia aprender “de cor” a reconstruir a instância de treinamento certa para cada codificação. Reconstruiria perfeitamente suas entradas sem realmente aprender qualquer padrão útil nos dados. Na prática, é improvável que tal mapeamento aconteça, mas ilustra o fato de que reconstruções perfeitas não são uma garantia de que o autoencoder tenha aprendido algo útil. No entanto, se ele produzir reconstruções muito ruins, é quase garantido que seja um autoencoder ruim. Uma opção para avaliar o desempenho de um autoencoder é medir a perda de reconstrução (por exemplo, calcular o MSE, o quadrado médio das saídas menos as entradas). Novamente, uma alta perda de reconstrução é um bom sinal de que o autoencoder é ruim, mas uma baixa perda de reconstrução não é uma garantia de que seja bom. Você também deve avaliar o autoencoder de acordo com o que será usado. Por exemplo, avalie também o desempenho do classificador se estiver utilizando o pré-treinamento não supervisionado de um deles.
4. Um autoencoder incompleto é aquele cuja camada de codificação é menor que as camadas de entrada e saída. Se for maior, é um autoencoder supercompleto. O principal risco de um autoencoder excessivamente incompleto é que ele pode falhar na reconstrução das entradas, por outro lado, o principal risco de um autoencoder supercompleto é que ele pode simplesmente copiar as entradas para as saídas, sem aprender nenhum recurso útil.
5. Para amarrar os pesos de uma camada de codificador e sua correspondente camada decodificador, você simplesmente torna os pesos do decodificador iguais à transposição dos pesos do codificador. O que reduz o número de parâmetros no modelo pela metade, geralmente fazendo com que o treinamento convirja mais rápido com menos dados de treinamento e reduzindo o risco de sobreajuste no conjunto de treinamento.
6. Uma técnica comum para visualizar as características aprendidas pela camada inferior de um autoencoder empilhado é simplesmente plotar os pesos de cada neurônio, remodelando cada vetor de peso para o tamanho de uma imagem de entrada

(por exemplo, para o MNIST, remodelando um vetor de peso no formato [784] para [28, 28]). Uma técnica é exibir as instâncias de treinamento que mais ativam cada neurônio para visualizar as características aprendidas pelas camadas superiores.

7. Um modelo gerado é um modelo capaz de gerar saídas aleatoriamente que se assemelham às instâncias de treinamento. Por exemplo, uma vez treinado com sucesso no conjunto de dados MNIST, um modelo gerado pode ser utilizado para gerar aleatoriamente imagens realistas de dígitos. A distribuição de saída é tipicamente semelhante aos dados de treinamento. Por exemplo, como MNIST contém muitas imagens de cada dígito, o modelo gerado produziria aproximadamente o mesmo número de imagens de cada dígito. Alguns modelos gerados podem ser parametrizados, por exemplo, para gerar apenas alguns tipos de saídas. Um exemplo de um autoencoder gerado é o autoencoder variacional.

Para as soluções dos exercícios 8, 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>.

## Capítulo 16: Aprendizado por Reforço

1. O Aprendizado por Reforço é uma área do Aprendizado de Máquina que visa criar agentes capazes de realizar ações em um ambiente de forma a maximizar as recompensas ao longo do tempo. Existem muitas diferenças entre AR e o aprendizado regular supervisionado e não supervisionado. Veja algumas:
  - No aprendizado supervisionado e não supervisionado, o objetivo geralmente é encontrar padrões nos dados e utilizá-los para fazer previsões. No Aprendizado por Reforço, o objetivo é encontrar uma boa política;
  - Ao contrário do aprendizado supervisionado, o agente não recebe explicitamente a resposta “certa”, mas deve aprender por tentativa e erro;
  - Ao contrário do aprendizado não supervisionado, existe uma forma de supervisão por meio de recompensas. Não informamos ao agente como realizar a tarefa, mas informamos quando está progredindo ou falhando;
  - Um agente do Aprendizado por Reforço precisa encontrar o equilíbrio certo entre explorar o ambiente, procurar novas maneiras de obter recompensas e explorar fontes de recompensas que ele já conhece. Em contraste, sistemas de aprendizado supervisionados e não supervisionados geralmente não precisam se preocupar com a exploração; apenas se alimentam dos dados de treinamento que recebem;
  - No aprendizado supervisionado e não supervisionado, as instâncias de treinamento são geralmente independentes (na verdade, são geralmente embaralhadas). No Aprendizado por Reforço, as observações consecutivas geralmente *não* são inde-

pendentes. Um agente pode permanecer na mesma região do ambiente por um tempo antes de se mover, então observações consecutivas serão muito correlacionadas. Uma memória de repetição é usada em alguns casos para garantir que o algoritmo de treinamento receba observações razoavelmente independentes.

2. Estas são algumas possíveis aplicações do Aprendizado por Reforço, além das mencionadas no Capítulo 16:

#### *Personalização de Música*

O ambiente é uma rádio Web de um usuário. O agente é o software que decide qual música tocar em seguida. Suas ações possíveis são tocar qualquer música no catálogo (ele deve tentar escolher uma música que o usuário gostará) ou tocar um anúncio (ele deve tentar escolher um anúncio no qual o usuário esteja interessado). O agente recebe uma pequena recompensa toda vez que o usuário ouve uma música, uma recompensa maior toda vez que o usuário ouve um anúncio, uma recompensa negativa quando o usuário pula uma música ou um anúncio, e uma recompensa muito negativa se o usuário sair.

#### *Marketing*

O ambiente é o departamento de marketing da sua empresa. O agente é o software que define para quais clientes uma campanha de e-mail deve ser direcionada, considerando seu perfil e histórico de compras (para cada cliente, há duas ações possíveis: enviar ou não enviar). Ele obtém uma recompensa negativa pelo custo da campanha de mala direta e uma recompensa positiva pela receita estimada gerada por essa campanha.

#### *Entrega de Produtos*

Deixe o agente controlar uma frota de caminhões de entrega, decidindo o que eles devem pegar nos depósitos, onde devem ir, o que devem entregar, e assim por diante. Eles receberiam recompensas positivas por cada produto entregue no prazo e recompensas negativas por entregas atrasadas.

3. Ao estimar o valor de uma ação, os algoritmos do Aprendizado por Reforço somam todas as recompensas que essa ação levou, dando mais peso às imediatas e menos peso às posteriores (considerando que uma ação tem mais influência no futuro próximo do que no futuro distante). Uma taxa de desconto é normalmente aplicada a cada intervalo de tempo para modelar isso. Por exemplo, com uma taxa de desconto de 0,9, uma recompensa de 100 recebida após dois intervalos de tempo é contada como apenas  $0,9^2 \times 100 = 81$  ao estimar o valor da ação. Pense na taxa de desconto como uma medida de quanto o futuro é avaliado em relação ao presente: se é muito próximo de 1, então o futuro é valorizado quase tanto quanto o presente, se for perto de 0, então

apenas as recompensas imediatas importam. É claro que isso impacta tremendamente a política ótima: se você valoriza o futuro, pode estar disposto a suportar muita dor imediata pela perspectiva de eventuais recompensas, ao passo que, se você não valoriza o futuro, simplesmente pegará qualquer recompensa imediata que encontrar, nunca investindo no futuro.

4. Você pode simplesmente resumir as recompensas obtidas para medir o desempenho de um agente do Aprendizado por Reforço. Em um ambiente simulado, você pode executar muitas simulações e observar a média das recompensas totais obtidas (e possivelmente observar o mínimo, o máximo, o desvio padrão e assim por diante).
5. O problema da atribuição de crédito se deve ao fato de um agente não ter nenhuma forma direta de saber quais de suas ações anteriores contribuíram para a recompensa que ganhou. O que normalmente ocorre quando há um grande atraso entre uma ação e as recompensas resultantes (por exemplo, durante um jogo Pong, do Atari, podem haver algumas dezenas de intervalos de tempo entre o momento em que o agente atinge a bola e o que ele ganha o ponto). Uma maneira de aliviar isso é fornecer recompensas de curto prazo ao agente quando possível. Geralmente, isso requer conhecimento prévio sobre a tarefa. Por exemplo, se quisermos construir um agente que aprenda a jogar xadrez, poderemos dar uma recompensa toda vez que ele capturar uma das peças do oponente em vez de recompensá-lo apenas quando vencer o jogo.
6. Um agente pode frequentemente permanecer na mesma região do seu ambiente por um tempo, então todas as suas experiências serão muito semelhantes durante esse período de tempo, podendo introduzir algum viés no algoritmo de aprendizado. Ele pode ajustar sua política para esta região do ambiente, mas não terá um bom desempenho assim que se deslocar para fora dela. Para resolver esse problema, você pode utilizar uma memória de repetição; em vez de usar apenas as experiências mais imediatas para aprender, o agente aprenderá baseado em um buffer de suas experiências passadas, recentes e não tão recentes (talvez seja por isso que sonhamos à noite: repetir nossas experiências do dia e aprender melhor com elas?).
7. Um algoritmo RL fora da política aprende o valor da política ótima (ou seja, a soma de recompensas com desconto que pode ser esperada para cada estado se o agente agir de maneira ideal), enquanto o agente segue uma política diferente. O Q-Learning é um bom exemplo de tal algoritmo. Em contraste, um algoritmo da política aprende o valor da política que o agente realmente executa, incluindo *exploration* e *exploitation*.

Para as soluções dos exercícios 8, 9 e 10, consulte os notebooks Jupyter disponíveis em <https://github.com/ageron/handson-ml>

## Apêndice B

# Lista de Verificação do Projeto de Aprendizado de Máquina

Esta lista de verificação pode guiá-lo em seus projetos de Aprendizado de Máquina.

São oito passos principais:

1. Foque o problema e olhe para o quadro geral;
2. Obtenha os dados;
3. Explore os dados para obter insights;
4. Prepare os dados para melhor expor os padrões de dados subjacentes aos algoritmos do Aprendizado de Máquina;
5. Explore vários diferentes modelos e liste os melhores;
6. Ajuste seus modelos e combine-os em uma ótima solução;
7. Apresente sua solução;
8. Lance, monitore e faça a manutenção de seu sistema.

Obviamente, você deve se sentir livre para adaptar esta lista às suas necessidades.

## Foque o Problema e Olhe para o Quadro Geral

1. Defina o objetivo em termos de negócios;
2. Como sua solução será usada?
3. Quais são as soluções atuais/soluções alternativas (se houver)?
4. Como você deve enquadrar esse problema (supervisionado/não supervisionado, online/offline, etc.)?
5. Como o desempenho deve ser medido?

6. A medida de desempenho está alinhada com o objetivo de negócios?
7. Qual seria o desempenho mínimo necessário para atingir o objetivo dos negócios?
8. Quais são os problemas comparáveis? Você pode reutilizar experiências ou ferramentas?
9. A perícia humana está disponível?
10. Como você resolveria o problema manualmente?
11. Liste as suposições que você (ou outros) fizer(am) até agora;
12. Verifique as suposições, se possível.

## Obtenha os Dados

Obs: para que você possa obter facilmente novos dados, automatize o máximo possível.

1. Liste os dados que você precisa e quanto você precisa;
2. Encontre e documente onde você pode obtê-los;
3. Verifique quanto espaço ocupará;
4. Verifique as obrigações legais e obtenha autorização, se necessário;
5. Obtenha autorizações de acesso;
6. Crie um espaço de trabalho (com espaço de armazenamento suficiente);
7. Obtenha os dados;
8. Converta os dados para um formato que você possa manipular facilmente (sem alterá-los em si);
9. Garanta que informações confidenciais sejam excluídas ou protegidas (por exemplo, anonimato);
10. Verifique o tamanho e o tipo de dados (séries cronológicas, amostra, geográfica, etc.);
11. Experimente um conjunto de teste, coloque-o de lado e nunca olhe para ele (sem bisbilhotar dados!).

## Explore os Dados

Obs: tente obter insights de um especialista do setor para essas etapas.

1. Crie uma cópia dos dados para exploração (amostre até um tamanho gerenciável, se necessário);

2. Crie um notebook Jupyter para manter um registro da sua exploração de dados;
3. Estude cada atributo e suas características:
  - Nome;
  - Tipo (categórico, int/float, limitado/ilimitado, texto, estruturado, etc.);
  - Porcentagem dos valores ausentes;
  - Ruídos e tipos de ruído (estocásticos, anomalias, erros de arredondamento, etc.);
  - Possivelmente útil para a tarefa?
  - Tipo de distribuição (Gaussian, uniforme, logarítmica, etc.).
4. Para tarefas de Aprendizado supervisionadas, identifique o atributo alvo;
5. Visualize os dados;
6. Estude as correlações entre atributos;
7. Estude como você resolveria o problema manualmente;
8. Identifique as transformações promissoras que você pode querer aplicar;
9. Identifique dados extras que seriam úteis (volte a “Obtenha os Dados” na página anterior);
10. Documente o que aprendeu.

## Prepare os Dados

Observações:

- Trabalhe em cópias dos dados (mantenha o conjunto de dados original intacto);
- Escreva funções para todas as transformações que você venha a aplicar nos dados, por cinco razões:
  - Para que você possa preparar facilmente os dados na próxima vez que obtiver novos conjuntos de dados;
  - Para que você possa aplicar essas transformações em projetos futuro;
  - Para limpar e preparar o conjunto de teste;
  - Para limpar e preparar novas instâncias de dados se sua solução for ao vivo;
  - Para facilitar o tratamento de suas opções de preparação como hiperparâmetros.

1. Limpeza de dados:
  - Conserte ou remova os outliers (opcional);
  - Preencha os valores ausentes (por exemplo, com zero, média, mediana... ) ou descarte suas linhas (ou colunas).
2. Seleção de características (opcional):
  - Descarte os atributos que fornecem informações inúteis para a tarefa.
3. Engenharia das características, quando apropriado:
  - Discretizar caraterísticas contínuas;
  - Decompor caraterísticas (por exemplo, categórico, data/hora, etc.);
  - Adicionar transformações promissoras de características (por exemplo,  $\log(x)$ ,  $\sqrt{x}$ ,  $x^2$ , etc.);
  - Agregar características em novas características promissoras.
4. Escala de características: padronizá-las ou normalizá-las.

## Modelos Promissores em Lista Resumida

Observações:

- Se os dados forem enormes, talvez você queira experimentar conjuntos de treinamento menores para poder treinar vários modelos diferentes em um tempo razoável (esteja ciente de que isso penaliza modelos complexos, como grandes redes neurais ou Florestas Aleatórias);
  - Mais uma vez, tente automatizar ao máximo essas etapas.
1. Utilizando parâmetros padrão, treine muitos modelos rápidos de diferentes categorias (por exemplo, lineares, Naive-Bayes, SVM, Florestas Aleatórias, redes neurais, etc.);
  2. Meça e compare seu desempenho;
    - Utilize validação cruzada *N-fold* e calcule a média e o desvio padrão da medida de desempenho nas N dobras para cada modelo.
  3. Analise as variáveis mais significantes para cada algoritmo;
  4. Analise os tipos de erros que os modelos cometem;
    - Quais dados um humano teria utilizado para preveni-los?
  5. Tenha uma rápida rodada de seleção de características e engenharia;

6. Itere rapidamente uma ou duas vezes as cinco etapas anteriores;
7. Faça uma lista resumida dos três ou cinco principais modelos mais promissores, preferindo modelos que cometam diferentes tipos de erros.

## Ajuste Fino do Sistema

Observações:

- Utilize o máximo de dados possível para esta etapa, especialmente à medida que você se aproxima do final do ajuste fino;
  - Como sempre, automatize o que puder.
1. Faça um ajuste fino nos hiperparâmetros utilizando a validação cruzada;
    - Trate como hiperparâmetros suas escolhas de transformação de dados, especialmente quando você não tiver certeza sobre elas (por exemplo, devo substituir valores ausentes por zero ou pelo valor mediano? Ou simplesmente descartar as linhas?);
    - A menos que haja poucos valores de hiperparâmetros para explorar, prefira a pesquisa aleatória em relação à grid search. Se o treinamento for muito longo, prefira uma abordagem de otimização de Bayes (por exemplo, utilizando priorizações Gaussianas de processo, como descrito por Jasper Snoek, Hugo Larochelle e Ryan Adams (<https://goo.gl/PEFfGr>).<sup>1</sup>
  2. Tente os métodos do ensemble. Combinar frequentemente seus melhores modelos terá melhor desempenho do que executá-los individualmente;
  3. Uma vez confiante em seu modelo final, meça seu desempenho no conjunto de testes para estimar o erro de generalização.



Não ajuste seu modelo depois de medir o erro de generalização: você apenas começaria a sobreajustar o conjunto de testes.

## Apresente Sua Solução

1. Documente o que você fez
2. Crie uma boa apresentação;
  - Certifique-se de destacar primeiro o quadro geral.

<sup>1</sup> "Practical Bayesian Optimization of Machine Learning Algorithms", J. Snoek, H. Larochelle, R. Adams (2012).

3. Explique por que sua solução atinge o objetivo comercial;
4. Não se esqueça de apresentar pontos interessantes que você anotou ao longo do caminho;
  - Descreva o que funcionou e o que não funcionou;
  - Liste suas suposições e as limitações do seu sistema.
5. Assegure-se de que suas principais descobertas sejam comunicadas por meio de belas visualizações ou declarações fáceis de lembrar (por exemplo, “a renda média é o principal previsor dos preços imobiliários”).

## Lance!

1. Tenha sua solução pronta para a produção (conecte-se a entradas de dados da produção, escreva testes de unidade, etc.);
2. Escreva código de monitoramento para verificar o desempenho ao vivo em intervalos regulares do seu sistema e acione alertas quando ele cair;
  - Cuidado com a degradação lenta também: os modelos tendem a “apodrecer” à medida que os dados evoluem;
  - A medição do desempenho pode exigir um canal humano (por exemplo, por meio de um serviço de crowdsourcing);
  - Monitore também a qualidade de suas entradas (por exemplo, um sensor com defeito enviando valores aleatórios ou a saída de outra equipe se tornando obsoleta), o que é particularmente importante para sistemas de aprendizado online.
3. Volte a treinar seus modelos em novos dados regularmente (automatize o máximo possível).

## Apêndice C

# Problema SVM Dual

Para entender a *dualidade*, primeiro você precisa entender o método dos *multiplicadores Lagrange*. A ideia geral é transformar um objetivo restrito de otimização em um objetivo não restrito movendo as restrições para uma função objetiva. Veremos um exemplo simples. Suponha que você queira encontrar os valores de  $x$  e  $y$  que minimizam a função  $f(x,y) = x^2 + 2y$ , sujeita a uma *restrição de igualdade*:  $3x + 2y + 1 = 0$ . Utilizando o método dos multiplicadores de Lagrange, começamos definindo uma nova função chamada *Lagrangiana* (ou *função Lagrange*):  $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$ . Cada restrição (neste caso, apenas uma) é subtraída do objetivo original e multiplicada por uma nova variável chamada multiplicador de Lagrange.

Joseph-Louis Lagrange mostrou que, se  $(\hat{x}, \hat{y})$  for uma solução para o problema de otimização restrita, então deve existir um  $\alpha$  tal que  $(\hat{x}, \hat{y}, \hat{\alpha})$  é um *ponto estacionário* Lagrangiano (um ponto estacionário é um ponto onde todas as derivadas parciais são iguais a zero). Em outras palavras, podemos calcular as derivadas parciais de  $g(x, y, \alpha)$  com relação a  $x$ ,  $y$ , e  $\alpha$ ; podemos encontrar os pontos onde essas derivadas são iguais a zero; e as soluções para o problema de otimização restrita (se existirem) devem estar entre esses pontos estacionários.

Neste exemplo as derivadas parciais são:

$$\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$$

Quando todas essas derivadas parciais são iguais a 0, descobrimos que  $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$ , a partir do qual podemos facilmente encontrar  $\hat{x} = \frac{3}{2}$ ,  $\hat{y} = -\frac{11}{4}$  e  $\hat{\alpha} = 1$ . Esse é o único ponto estacionário e, como ele respeita a restrição, deve ser a solução para o problema da otimização restrita.

No entanto, esse método se aplica somente a restrições de igualdade e, felizmente, pode ser generalizado para as *restrições de desigualdade* também (por exemplo,  $3x + 2y + 1 \geq 0$ ) sob algumas condições de regularidade (que são respeitadas pelos objetivos da SVM). A *Lagrangiana generalizada* para o problema da hard margin é dada pela Equação C-1, na qual as variáveis  $\alpha^{(i)}$  são chamadas de multiplicadores *Karush–Kuhn–Tucker* (KKT), e devem ser maiores ou iguais a zero.

*Equação C-1. Lagrangiana Generalizada para o problema da hard margin*

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} (t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) - 1)$$

com  $\alpha^{(i)} \geq 0$  para  $i = 1, 2, \dots, m$

Assim como com o método dos multiplicadores de Lagrange, você pode calcular as derivadas parciais e localizar os pontos estacionários. Se houver uma solução, ela estará necessariamente entre os pontos estacionários  $(\hat{\mathbf{w}}, \hat{b}, \hat{\alpha})$  que respeitam as *condições KKT*:

- Respeite as limitações do problema:  $t^{(i)}((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}) \geq 1$  para  $i = 1, 2, \dots, m$ ;
- Verifique  $\hat{\alpha}^{(i)} \geq 0$  para  $i = 1, 2, \dots, m$ ;
- Tanto  $\hat{\alpha}^{(i)} = 0$  ou a  $i$ -ésima limitação deve ser uma *limitação ativa*, o que significa que deve manter a igualdade:  $t^{(i)}((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}) = 1$ . Essa condição é chamada de condição *folga complementar* e implica que tanto  $\hat{\alpha}^{(i)} = 0$  ou a  $i$ -ésima instância fica na fronteira (é um vetor de suporte).

Observe que as condições de KKT são necessárias para que um ponto estacionário seja uma solução do problema da otimização restrita. Sob algumas condições, elas também são condições suficientes. Felizmente, o problema de otimização da SVM cumpre essas condições, portanto, qualquer ponto estacionário que atenda às condições do KKT é garantido como uma solução para o problema da otimização restrita.

Com a Equação C-2 podemos calcular as derivadas parciais da Lagrangiana generalizada em relação ao  $\mathbf{w}$  e  $b$ .

*Equação C-2. Derivadas parciais da Lagrangiana generalizada*

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \alpha) = - \sum_{i=1}^m \alpha^{(i)} t^{(i)}$$

Quando essas derivadas parciais são iguais a 0, temos a Equação C-3.

*Equação C-3. Propriedades dos pontos estacionários*

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} = 0$$

Se conectarmos esses resultados na definição da Lagrangiana Generalizada, alguns termos desaparecem e encontramos a Equação C-4.

*Equação C-4. Dupla forma do problema SVM*

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \hat{\alpha}) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \hat{\alpha}^{(i)} \hat{\alpha}^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \hat{\alpha}^{(i)}$$

com  $\hat{\alpha}^{(i)} \geq 0$  para  $i = 1, 2, \dots, m$

O objetivo agora é encontrar o vetor  $\hat{\alpha}$  que minimiza esta função, com  $\hat{\alpha}^{(i)} \geq 0$  para todas as instâncias. Esse problema de otimização restrita é o problema duplo que procurávamos.

Uma vez encontrada a  $\hat{\alpha}$  ótima, você pode calcular  $\hat{\mathbf{w}}$  utilizando a primeira linha da Equação C-3. Para calcular  $b$ , você pode utilizar o fato de um vetor de suporte dever verificar  $t^{(i)}(\hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)} + \hat{b}) = 1$ , para que, se a k-ésima instância for um vetor de suporte (ou seja,  $\hat{\alpha}^{(k)} > 0$ ), você poderá usá-lo para calcular  $\hat{b} = t^{(k)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(k)}$ . No entanto, é preferível calcular a média de todos os vetores de suporte para obter um valor mais estável e preciso, como na Equação C-5.

*Equação C-5. Estimativa de termo de polarização utilizando a forma dual*

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m [t^{(i)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}]$$



## Apêndice D

# Autodiff

Este apêndice explica como funciona o recurso autodiff do TensorFlow, e como ele se compara a outras soluções. Suponha que você defina uma função  $f(x,y) = x^2y + y + 2$ , e precise de suas derivadas parciais,  $\frac{\partial f}{\partial x}$  e  $\frac{\partial f}{\partial y}$ , para desempenhar o Gradiente Descendente (ou algum outro algoritmo de otimização). Suas principais opções são a diferenciação manual, a simbólica, a numérica, autodiff no modo forward ou, finalmente, autodiff no modo reverso. O TensorFlow implementa esta última, mas passaremos por cada uma delas.

## Diferenciação Manual

A primeira abordagem é pegar um lápis, um pedaço de papel e usar seu conhecimento de cálculo para derivar as derivadas parciais manualmente. Não é muito difícil para a função  $f(x,y)$  que acabou de ser definida; você só precisa usar cinco regras:

- A derivada de uma constante é 0;
- A derivada de  $\lambda x$  é  $\lambda$  ( $\lambda$  é uma constante);
- A derivada de  $x^\lambda$  é  $\lambda x^{\lambda-1}$ , então a derivada de  $x^2$  é  $2x$ ;
- A derivada de uma soma das funções é a soma dessas derivadas da função;
- A derivada de  $\lambda$  vezes uma função é  $\lambda$  vezes sua derivada;

A partir destas regras, você pode derivar a Equação D-1:

*Equação D-1. Derivadas Parciais de  $f(x,y)$*

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Essa abordagem pode se tornar muito tediosa para funções mais complexas e você corre o risco de cometer erros. A boa notícia é que derivar as equações matemáticas para as derivadas parciais, como acabamos de fazer, pode ser automatizado através de um processo chamado *diferenciação simbólica*.

## Diferenciação Simbólica

A Figura D-1 mostra como a diferenciação simbólica funciona em uma função ainda mais simples,  $g(x, y) = 5 + xy$ . O grafo para essa função é representado à esquerda. Após a diferenciação simbólica, obtemos o grafo à direita, que representa a derivada parcial  $\frac{\partial f}{\partial x} = 0 + (0 \times x + y \times 1) = y$  (poderíamos da mesma forma obter a derivada parcial com relação a  $y$ ).

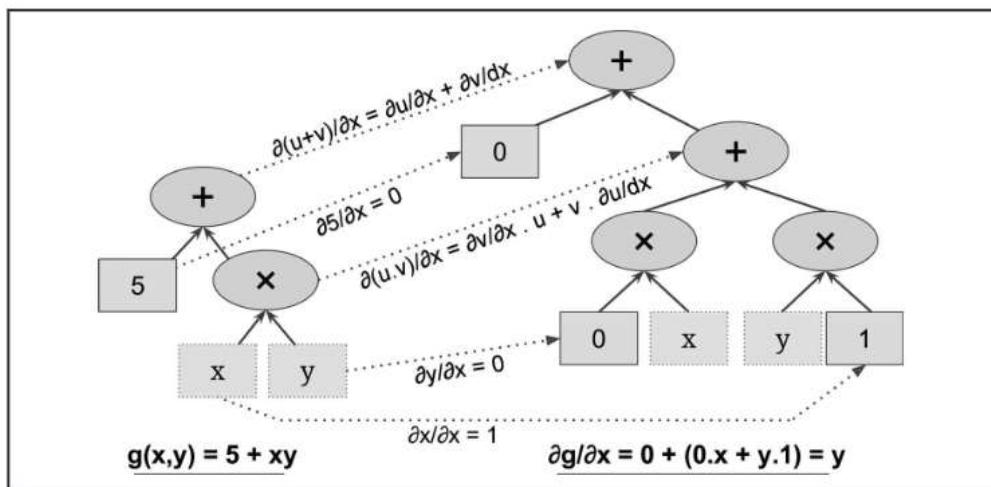


Figura D-1. Diferenciação Simbólica

O algoritmo começa obtendo a derivada parcial dos nós da folha. O nó constante (5) retorna a constante 0, pois a derivada de uma constante é sempre 0. A variável  $x$  retorna a constante 1 pois  $\frac{\partial f}{\partial x} = 1$ , e a variável  $y$  retorna a constante 0, pois  $\frac{\partial f}{\partial y} = 0$  (se estivéssemos procurando pela derivada parcial em relação a  $y$ , seria o inverso).

Agora, temos tudo o que precisamos para subir o grafo para o nó de multiplicação na função  $g$ . O cálculo nos diz que a derivada do produto das duas funções  $u$  e  $v$  é  $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + v \times \frac{\partial u}{\partial x}$ , portanto podemos construir uma grande parte do grafo à direita, representando  $0 \times x + y \times 1$ .

Finalmente, podemos ir até o nó de adição na função  $g$ . Como mencionado, a derivada de uma soma de funções é a soma das derivadas dessas funções. Então, precisamos apenas criar um nó de adição e conectá-lo às partes do grafo já calculadas. Obtemos a derivada parcial correta:  $\frac{\partial f}{\partial x} = 0 + 0 (\times x + y \times 1)$ .

No entanto, isso pode ser simplificado (e muito). Para se livrar de todas as operações desnecessárias, algumas etapas triviais de poda podem ser aplicadas a este grafo e obtemos um grafo muito menor com apenas um nó:  $\frac{\partial f}{\partial x} = y$ .

Nesse caso, a simplificação é bem fácil, mas, para uma função mais complexa, a diferenciação simbólica pode produzir um grande grafo, que pode ser difícil de simplificar e levar a um desempenho abaixo do ideal. Mais importante ainda, a diferenciação simbólica não pode lidar com funções definidas com código arbitrário, por exemplo a seguinte função discutida no Capítulo 9:

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

## Diferenciação Numérica

A solução mais simples é calcular numericamente uma aproximação das derivadas. Lembre-se de que a derivada  $h'(x_0)$  de uma função  $h(x)$  em um ponto  $x_0$  é a inclinação da função naquele ponto, ou, mais precisamente, a Equação D-2.

*Equação D-2. Derivada de uma função  $h(x)$  no ponto  $x_0$*

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\epsilon \rightarrow 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon} \end{aligned}$$

Então, se quisermos calcular a derivada parcial de  $f(x,y)$  em relação a  $x$ , sendo  $x = 3$  e  $y = 4$ , podemos simplesmente calcular  $f(3 + \epsilon, 4) - f(3, 4)$  e dividir o resultado por  $\epsilon$ , utilizando um valor muito pequeno para  $\epsilon$ , que é exatamente o que faz o código a seguir:

```

def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)

```

Infelizmente, o resultado é impreciso (e piora para funções mais complexas). Os resultados corretos na verdade são, respectivamente, 24 e 10:

```

>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966

```

Note que, para calcular ambas as derivadas parciais, temos que chamar `f()` pelo menos três vezes (nós a chamamos quatro vezes no código anterior, mas ela poderia ser otimizada). Se houvesse mil parâmetros, precisaríamos chamar `f()` pelo menos mil e uma vezes, tornando a diferenciação numérica muito ineficiente quando lidamos com grandes redes neurais.

No entanto, é tão simples implementar a diferenciação numérica que é uma ótima ferramenta para verificar se os outros métodos foram implementados corretamente. Por exemplo, se ela não concordar com sua função derivada manualmente, sua função provavelmente terá um erro.

## Autodiff no Modo Forward

Autodiff no modo forward não é nem uma diferenciação numérica nem uma diferenciação simbólica, mas, de certa forma, é o fruto do amor de ambas. Ela se baseia em *números duais*, que são números (estranhos, mas fascinantes) na forma  $a + b\epsilon$ , sendo  $a$  e  $b$  números reais e  $\epsilon$  um número infinitesimal tal que  $\epsilon^2 = 0$  (mas  $\epsilon \neq 0$ ). Você pode pensar no número dual  $42 + 24\epsilon$  como algo semelhante a  $42,0000\cdots000024$  com um número infinito de “0” (mas é claro que isso é simplificado apenas para dar uma ideia do que são os números duais). Um número dual é representado como um par de floats na memória. Por exemplo,  $42 + 24\epsilon$  é representado pelo par  $(42,0, 24,0)$ .

Conforme mostrado na Equação D-3, Números Duais podem ser adicionados, multiplicados e assim por diante.

*Equação D-3. Algumas operações com números duais*

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Mais importante, pode ser mostrado que  $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$ , então calcular  $h(a + \epsilon)$  dá a você  $h(a)$  e a derivada  $h'(a)$  em uma só tacada. A Figura D-2 mostra como o autodiff no modo forward calcula a derivada parcial de  $f(x,y)$  com relação a  $x$  em  $x = 3$  e  $y = 4$ . Tudo que precisamos é calcular  $f(3 + \epsilon, 4)$ , que gerará um número dual cujo primeiro componente é igual a  $f(3, 4)$  e cujo segundo componente é igual a  $\frac{\partial f}{\partial x}(3, 4)$ .

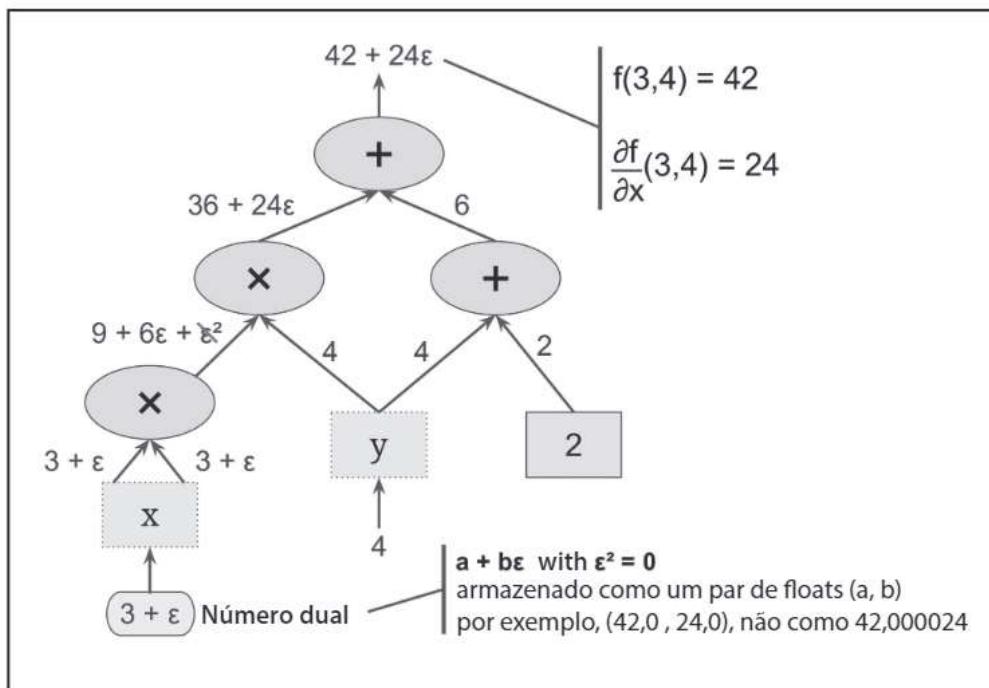


Figura D-2. Autodiff no Modo Forward

Para calcular  $\frac{\partial f}{\partial y}(3, 4)$  teríamos que passar pelo grafo novamente, mas, desta vez, com  $x = 3$  e  $y = 4 + \epsilon$ .

Portanto, o autodiff no modo forward é muito mais preciso do que a diferenciação numérica, mas sofre da mesma grande falha: se houvesse mil parâmetros, seriam necessárias mil passagens pelo grafo para calcular todas as derivadas parciais. É aqui que o autodiff no modo reverso brilha porque pode calcular todas elas em apenas duas passagens pelo grafo.

## Autodiff no Modo Reverso

O autodiff no modo reverso é a solução implementada pelo TensorFlow. Primeiro, ele passa pelo grafo na direção forward para calcular o valor de cada nó (ou seja, das entradas para a saída). Em seguida, faz uma segunda passagem, desta vez na direção inversa, para calcular todas as derivadas parciais (ou seja, da saída para as entradas). A figura D-3 representa a segunda passagem. Todos os valores do nó foram calculados durante a primeira, começando de  $x = 3$  e  $y = 4$ , valores que podem ser vistos no canto inferior direito de cada nó (por exemplo,  $x \times x = 9$ ). Para maior clareza, os nós são rotulados de  $n_1$  a  $n_7$  e o nó de saída é  $n_7$ :  $f(3,4) = n_7 = 42$ .

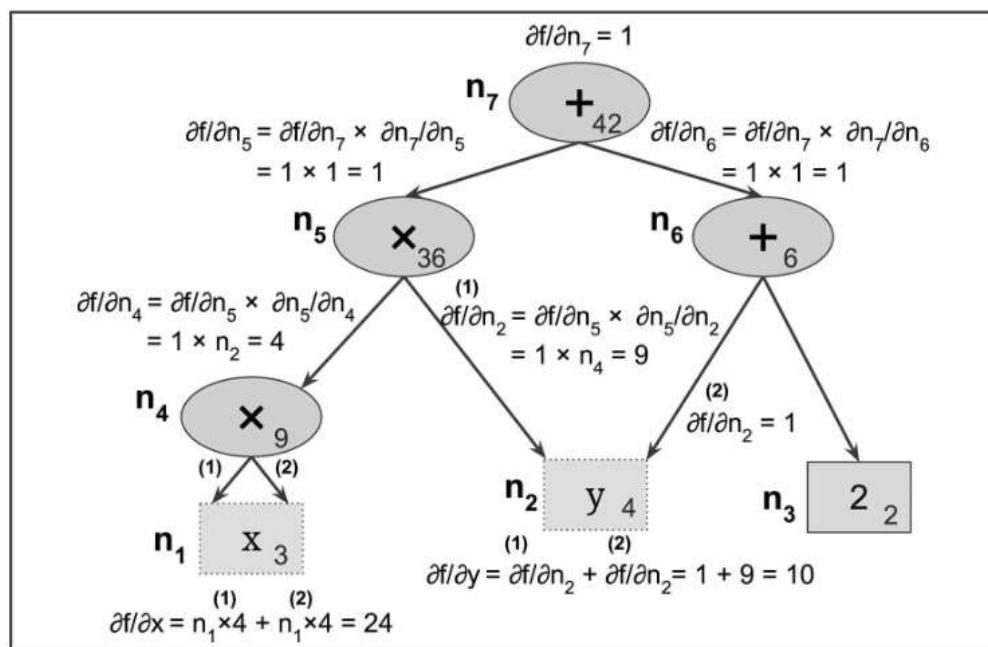


Figura D-3. Autodiff no Modo Reverso

A ideia é descer gradualmente no grafo em relação a cada nó consecutivo até chegarmos aos nós variáveis calculando a derivada parcial de  $f(x,y)$ . Para isso, o autodiff no modo reverso depende muito da *regra da cadeia*, mostrada na Equação D-4.

Equação D-4. Regra da Cadeia

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Desde que  $n_7$  é o nó de saída,  $f = n_7$  tão trivialmente  $\frac{\partial f}{\partial n_7} = 1$ .

Continuaremos no grafo até o  $n_5$ : qual a variação do  $f$  quando o  $n_5$  varia? A resposta é  $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$ . Já sabemos que  $\frac{\partial f}{\partial n_7} = 1$ , então, tudo que precisamos é  $\frac{\partial n_7}{\partial n_5}$ . Como o  $n_7$  simplesmente realiza a soma  $n_5 + n_6$ , descobrimos que  $\frac{\partial f}{\partial n_7} = 1$ , então  $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$ . Agora, podemos prosseguir para o nó  $n_4$ : qual a variação do  $f$  quando o  $n_4$  varia? A resposta é  $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$ . Como  $n_5 = n_4 \times n_2$ , descobrimos que  $\frac{\partial n_5}{\partial n_4} = n_2$ , então  $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$ .

O processo continua até chegarmos ao final do grafo. Nesse ponto, teremos calculado todas as derivadas parciais de  $f(x,y)$  no ponto  $x = 3$  e  $y = 4$ . Neste exemplo, descobrimos  $\frac{\partial f}{\partial y} = 24$  e  $\frac{\partial f}{\partial x} = 10$ . Parece correto!

O autodiff no modo reverso é uma técnica muito poderosa e precisa, especialmente quando há muitas entradas e poucas saídas, pois requer apenas um passo a frente e um passo reverso por saída para calcular todas as derivadas parciais de todas as saídas em relação a todas as entradas. Mais importante, ele pode lidar com funções definidas por código arbitrário e também pode manipular funções que não são totalmente diferenciáveis, contanto que você peça para calcular as derivadas parciais em pontos diferenciáveis.



Se você implementar um novo tipo de operação no TensorFlow e quiser torná-la compatível com o autodiff, será necessário fornecer uma função que crie um subgrafo para calcular suas derivadas parciais em relação às suas entradas. Por exemplo, suponha que você implemente uma função que calcula o quadrado de sua entrada  $f(x) = x^2$ . Nesse caso, você precisaria fornecer a função derivativa correspondente  $f'(x) = 2x$ . Note que esta função não calcula um resultado numérico, mas constrói um subgrafo que (mais tarde) calculará o resultado, o que é muito útil porque significa que você pode calcular gradientes de gradientes (para calcular derivadas de segunda ordem ou até mesmo derivadas de ordem superior).



## Apêndice E

# Outras Arquiteturas Populares de RNA

Neste apêndice, daremos uma visão geral de algumas arquiteturas de redes neurais historicamente importantes muito menos utilizadas hoje do que Perceptrons Multicamadas profundas (Capítulo 10), redes neurais convolucionais (Capítulo 13), redes neurais recorrentes (Capítulo 14), ou autoencoders (Capítulo 15). Elas são frequentemente mencionadas na literatura, e algumas ainda são utilizadas em muitas aplicações, então vale a pena conhecê-las. Além disso, discutiremos as *redes de crença profunda* (DBNs), que foram a tecnologia de ponta em Aprendizado Profundo até o início de 2010 e ainda são objeto de pesquisas muito ativas, então podem voltar com força em um futuro próximo.

## Redes Hopfield

*Redes Hopfield*, introduzidas por W.A. Little em 1974 e popularizadas por J. Hopfield em 1982, são redes de *memória associativa*: primeiro você ensina alguns padrões e, quando veem um novo, elas (esperamos) produzem o padrão aprendido mais próximo. O que as tornou úteis antes de serem superadas por outras abordagens, principalmente para o reconhecimento de caracteres. Primeiro, você treina a rede mostrando exemplos de imagens de caracteres (cada pixel binário é mapeado em um neurônio) e, em seguida, quando você apresenta uma nova imagem de caractere, ela exibe o caractere aprendido mais próximo após algumas iterações.

Eles são grafos totalmente conectados (veja a Figura E-1); isto é, cada neurônio está conectado a todos os outros. Note que, no diagrama, as imagens são  $6 \times 6$  pixels, então a rede neural à esquerda deve conter 36 neurônios (e 648 conexões), mas, para fins de maior clareza visual, é representada uma rede muito menor.

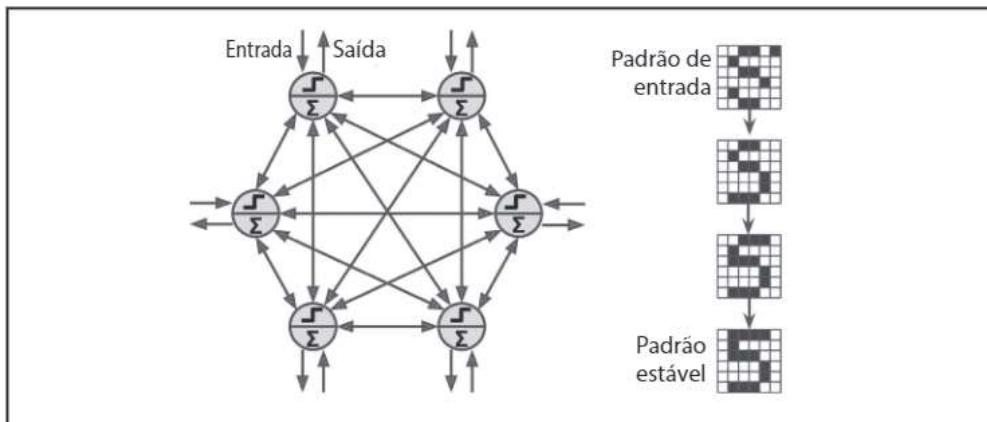


Figura E-1. Rede Hopfield

O algoritmo de treinamento funciona com a utilização da regra de Hebb: para cada imagem do treinamento, se os pixels correspondentes estiverem ligados ou desligados, o peso entre dois neurônios é aumentado, mas é diminuído se um pixel estiver ligado e o outro desligado.

Basta ativar os neurônios que correspondem aos pixels ativos para mostrar uma nova imagem para a rede que, então, calcula a saída de cada neurônio, e dando-lhe uma nova imagem. Você pode, então, pegar essa nova imagem, repetir todo o processo e, depois de um tempo, a rede atinge um estado estável. Geralmente, isso corresponde à imagem do treinamento que mais se assemelha à imagem de entrada.

Esta *função de energia*, como é chamada, está associada às redes Hopfield. Em cada iteração, a energia diminui, então a rede eventualmente se estabiliza em um estado de baixa energia. O algoritmo de treinamento ajusta os pesos para diminuir o nível de energia dos padrões de treinamento, de modo que a rede se estabilize em uma dessas configurações de baixa energia. Infelizmente, alguns padrões que não estavam no conjunto de treinamento também ficam com pouca energia, então a rede às vezes se estabiliza em uma configuração que não foi aprendida. Estes são chamados de *padrões espúrios*.

Outra grande falha das redes Hopfield é que elas não escalonam muito bem; sua capacidade de memória é aproximadamente igual a 14% do número de neurônios. Por exemplo, para classificar imagens  $28 \times 28$ , você precisaria de uma rede Hopfield com 784 neurônios totalmente conectados e 306.936 pesos. Essa rede só aprenderia cerca de 110 caracteres diferentes (14% de 784), o que é uma grande quantidade de parâmetros para uma memória tão pequena.

## Máquinas de Boltzmann

As *Máquinas de Boltzmann* foram inventadas em 1985 por Geoffrey Hinton e Terrence Sejnowski e, assim como as redes Hopfield, elas são RNAs totalmente conectadas, mas são baseadas em *neurônios estocásticos*: em vez de utilizar uma função de etapa determinística para decidir que valor produzir, esses neurônios geram 1 com alguma probabilidade, e 0 caso contrário. A função de probabilidade que essas RNAs utilizam baseia-se na distribuição de Boltzmann (usada em mecânica estatística), daí seu nome. A equação E-1 fornece a probabilidade de que um neurônio em particular produza 1.

*Equação E-1. Probabilidade de o i-ésimo neurônio exibir 1*

$$p(s_i^{(\text{next step})} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{ij} s_j + b_i}{T}\right)$$

- $s_j$  é o j-ésimo estado do neurônio (0 ou 1);
- $w_{ij}$  é o peso de conexão entre o i-ésimo e o j-ésimo neurônios. Note que  $w_{ii}=0$ ;
- $b_i$  é o i-ésimo termo de polarização do neurônio e podemos implementá-lo adicionando um neurônio de viés à rede;
- $N$  é o número de neurônios na rede;
- $T$  é um número chamado de *temperatura* da rede; quanto mais alta a temperatura, mais aleatória é a saída (ou seja, maior a probabilidade de se aproximar de 50%);
- $\sigma$  é a função logística.

Os neurônios nas máquinas de Boltzmann são separados em dois grupos: *unidades visíveis* e *unidades ocultas* (veja a Figura E-2). Todos os neurônios funcionam da mesma maneira estocástica, mas as unidades visíveis são as que recebem as entradas e de onde as saídas são lidas.

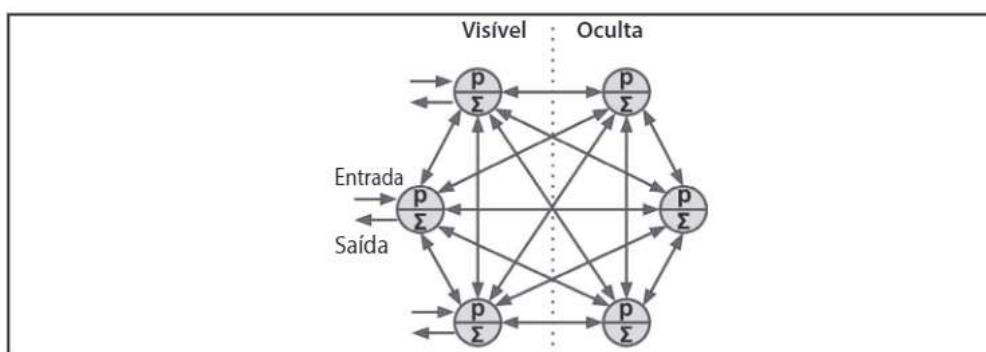


Figura E-2. Máquina de Boltzmann

Devido à sua natureza estocástica, uma máquina de Boltzmann nunca se estabilizará em uma configuração fixa, mas manterá a comutação entre muitas configurações. Se estiver funcionando por um tempo suficientemente longo, a probabilidade de observar uma configuração específica será apenas uma função dos pesos de conexão e termos de polarização, não da configuração original (da mesma forma que, após embaralhar um baralho de cartas por tempo suficiente, a configuração do *deck* não depende do seu estado inicial). Quando a rede atinge esse estado no qual a configuração original é “esquecida”, diz-se que ela está em *equilíbrio térmico* (embora sua configuração continue mudando o tempo todo). Podemos simular uma ampla gama de distribuições de probabilidade definindo os parâmetros de rede apropriadamente, permitindo que a rede atinja o equilíbrio térmico e, em seguida, observando seu estado, o que é chamado de *modelo gerador*.

Treinar uma máquina Boltzmann significa encontrar os parâmetros que farão com que a rede se aproxime da distribuição de probabilidade do conjunto de treinamento. Por exemplo, se houver três neurônios visíveis e o conjunto de treinamento contiver 75% do trio (0, 1, 1), 10% do trio (0, 0, 1) e 15% do trio (1, 1, 1), então, após treinar uma máquina Boltzmann, você poderia utilizá-la para gerar trios binários aleatórios, com aproximadamente a mesma distribuição de probabilidade. Por exemplo, cerca de 75% do tempo seria a saída do trio (0, 1, 1).

Esse modelo gerador pode ser utilizado de várias maneiras. Por exemplo, ele automaticamente “reparará” a imagem de maneira razoável se for treinado em imagens e você fornecer uma imagem incompleta ou ruidosa para a rede. Você também pode utilizar um modelo gerador para a classificação, basta adicionar alguns neurônios visíveis para programar a classe da imagem de treinamento (por exemplo, adicionar 10 neurônios visíveis e ligar apenas o quinto neurônio quando a imagem de treinamento representar um 5). Então, quando receber uma nova imagem, a rede ativará automaticamente os neurônios visíveis apropriados, indicando a classe da imagem (por exemplo, ativará o quinto neurônio visível se a imagem representar um 5).

Infelizmente, não há técnica eficiente para treinar máquinas de Boltzmann, mas algoritmos bastante eficientes para treinar *máquinas restritas de Boltzmann* (RBM) foram desenvolvidos.

## Máquinas Restritas de Boltzmann

Uma *Máquina Restrita de Boltzmann* (RBM) é simplesmente uma máquina de Boltzmann na qual não existem conexões entre unidades visíveis ou entre unidades ocultas, somente entre unidades visíveis e ocultas. Por exemplo, a Figura E-3 representa uma RBM com três unidades visíveis e quatro ocultas.

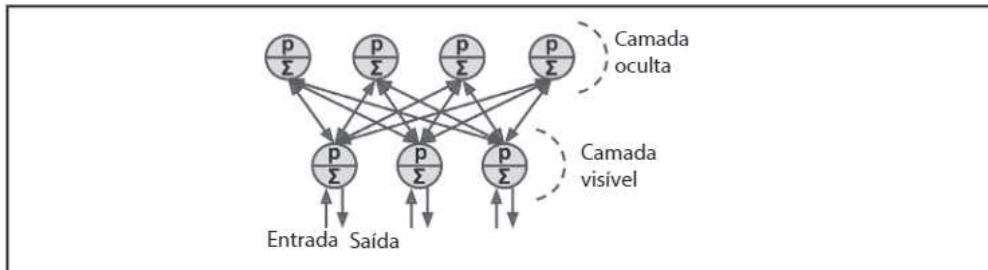


Figura E-3. Máquina Restrita de Boltzmann

Um algoritmo de treinamento muito eficiente denominado *Divergência de Contraste* foi introduzido em 2005 por Miguel Á. Carreira-Perpiñán e Geoffrey Hinton (<http://goo.gl/ZCP6Ir>)<sup>1</sup> e funciona assim: para cada instância de treinamento  $x$ , o algoritmo começa por fornecê-lo para a rede definindo o estado das unidades visíveis como  $x_1, x_2, \dots, x_n$ . Então, aplicando a equação estocástica descrita anteriormente, você calcula o estado das unidades ocultas (Equação E-1), resultando em um vetor oculto  $h$  (em que  $h_i$  é igual ao estado da  $i$ -ésima unidade). Em seguida, aplicando a mesma equação estocástica, calcule o estado das unidades visíveis, resultando no vetor  $x'$ . Então, mais uma vez, você calcula o estado das unidades ocultas, o que lhe dá um vetor  $h'$ . Agora, aplicando a regra da Equação E-2, você pode atualizar cada peso de conexão, sendo  $\eta$  a taxa de aprendizado.

*Equação E-2. Atualização de peso de divergência contrastante*

$$w_{i,j} \leftarrow w_{i,j} + \eta(x \cdot h^T - x' \cdot h'^T)$$

O grande benefício desse algoritmo é que ele não exige esperar que a rede atinja o equilíbrio térmico: ele simplesmente avança, retrocede e avança novamente, e é isso. Isto o torna incomparavelmente mais eficiente do que os algoritmos anteriores, e foi um ingrediente-chave para o primeiro sucesso do Aprendizado Profundo baseado em múltiplas RBMs empilhadas.

## Redes de Crença Profunda

Podemos empilhar várias camadas de RBMs; as unidades ocultas RBM de primeiro nível servem como unidades visíveis para a RBM da segunda camada e assim por diante. Tal pilha RBM é chamada de *rede de crença profunda* (DBN).

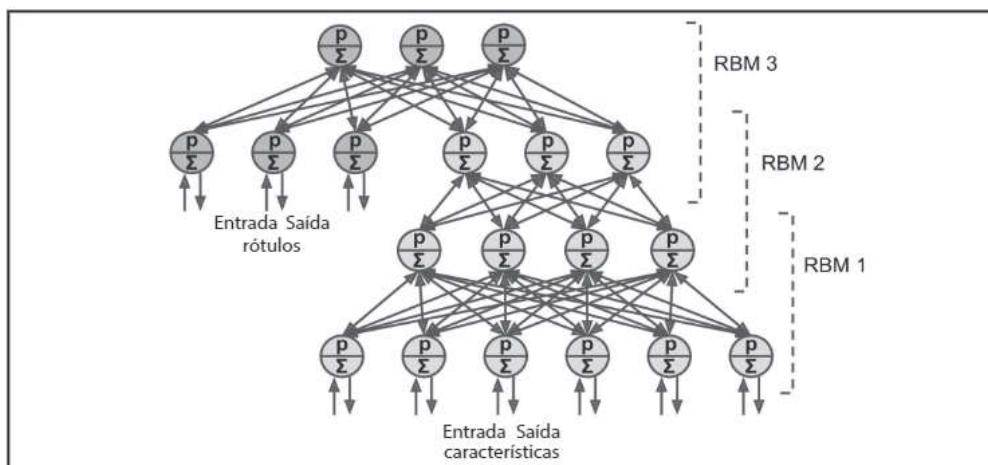
Yee-Whye Teh, um dos estudantes de Geoffrey Hinton, observou que, com a utilização da Divergência de Contraste, era possível treinar DBNs uma camada de cada vez, co-

<sup>1</sup> “On Contrastive Divergence Learning,” M. Á. Carreira-Perpiñán and G. Hinton (2005).

meçando com as camadas inferiores e, em seguida, movendo-se gradualmente até as camadas superiores, levando ao artigo inovador que deu início ao tsunami do Aprendizado Profundo em 2006 (<http://goo.gl/BcZQrH>).<sup>2</sup>

Assim como as RBMs, as DBNs aprendem a reproduzir a distribuição de probabilidade de suas entradas sem qualquer supervisão. No entanto, são muito melhores nisso, pela mesma razão que as redes neurais profundas são mais poderosas que as rasas: os dados do mundo real costumam ser organizados em padrões hierárquicos, e as DBNs se aproveitam disso. Suas camadas inferiores aprendem características de baixo nível nos dados de entrada enquanto camadas mais altas aprendem características de alto nível.

Assim como as RBMs, as DBNs são fundamentalmente não supervisionadas, mas, adicionando algumas unidades visíveis para representar os rótulos, você também pode treiná-las de maneira supervisionada. Além disso, uma grande característica das DBNs é que elas podem ser treinadas de forma semissupervisionada. A Figura E-4 representa uma DBN configurada para o aprendizado semissupervisionado.



*Figura E-4. Uma rede de crença profunda configurada para aprendizado semissupervisionado*

Primeiro, a RBM 1 é treinada sem supervisão e aprende características de baixo nível nos dados de treinamento. Em seguida, novamente sem supervisão, a RBM 2 é treinada como entradas com unidades ocultas da RBM 1: ela aprende características de um nível mais alto (observe que as unidades ocultas da RBM 2 incluem apenas as três unidades mais à direita, não as unidades de rótulo). Várias outras RBMs poderiam ser empilhadas dessa maneira, mas você entendeu. Até agora, o treinamento foi 100% sem supervisão.

<sup>2</sup> “A Fast Learning Algorithm for Deep Belief Nets”, G. Hinton, S. Osindero, Y. Teh (2006).

Por fim, utilizando as unidades ocultas da RBM 2 como entradas, a RBM 3 é treinada e, bem como unidades extras visíveis utilizadas para representar os rótulos de destino (por exemplo, um vetor one-hot representando a classe da instância), ela aprende a associar características de alto nível a rótulos de treinamento. Este é o passo supervisionado.

Ao final do treinamento, se você fornecer uma nova instância para a RBM 1, o sinal se propagará até a RBM 2, depois até o topo da RBM 3 e, em seguida, voltará às unidades de rótulo; esperamos que a etiqueta apropriada se acenda. É assim que uma DBN pode ser utilizada para classificação.

Um grande benefício dessa abordagem semissupervisionada é que você não precisa de muitos dados rotulados de treinamento. Somente uma pequena quantidade de instâncias de treinamento rotuladas por classe será necessária se as RBMs não supervisionadas fizerem um bom trabalho. Da mesma forma, um bebê aprende a reconhecer objetos sem supervisão, então, quando você aponta para uma cadeira e diz “cadeira”, o bebê associa a palavra “cadeira” à classe de objetos que ele já aprendeu a reconhecer por si mesmo. Você não precisa apontar para todas as cadeiras e dizer “cadeira”; apenas alguns exemplos serão suficientes (apenas o suficiente para que o bebê possa ter certeza de que você está realmente se referindo à cadeira, não à sua cor ou a uma das partes da cadeira).

Muito surpreendentemente, as DBNs também podem funcionar em sentido inverso. Se você ativar uma das unidades de etiqueta, o sinal se propagará até as unidades ocultas da RBM 3, depois para a RBM 2 e, em seguida, RBM 1, e uma nova instância será gerada pelas unidades visíveis da RBM 1, que geralmente parecerá uma instância regular da classe cuja unidade de rótulo você ativou. Esta capacidade gerativa das DBNs é bastante poderosa. Por exemplo, ela foi usada para gerar legendas automaticamente para imagens e vice-versa: primeiro uma DBN é treinada (sem supervisão) para aprender características em imagens, e outra DBN é treinada (novamente sem supervisão) para aprender características em conjuntos de legendas (por exemplo, “carro” geralmente vem com “automóvel”). Em seguida, uma RBM é empilhada sobre as duas DBNs e treinada com um conjunto de imagens juntamente com suas legendas; aprende a associar características de alto nível em imagens com características de alto nível em legendas. Em seguida, se você alimentar a DBN com uma imagem de um carro, o sinal se propagará pela rede, até a RBM de nível superior e voltará para a parte inferior da legenda DBN, produzindo uma legenda. Devido à natureza estocástica das RBMs e DBNs, a legenda continuará mudando aleatoriamente, mas será apropriada para a imagem. Se você gerar algumas centenas de legendas, as mais geradas provavelmente serão uma boa descrição da imagem.<sup>3</sup>

---

<sup>3</sup> Veja este vídeo de Geoffrey Hinton para mais detalhes e uma demonstração: <http://goo.gl/7Z5QiS>.

## Mapas Auto-Organizáveis

*Mapas Auto-Organizáveis* (SOM) são bem diferentes de todos os outros tipos de redes neurais que discutimos até agora. Eles são utilizados para produzir uma representação de dimensão inferior de um conjunto de dados de alta dimensão, geralmente para visualização, agrupamento ou classificação. Como mostrado na Figura E-5, os neurônios estão espalhados por um mapa (tipicamente 2D para visualização), mas pode ser qualquer número de dimensões que você quiser), e cada neurônio tem uma conexão ponderada com cada entrada (note que o diagrama mostra apenas duas entradas, mas normalmente há um número muito grande já que o ponto principal dos SOMs é reduzir a dimensionalidade).

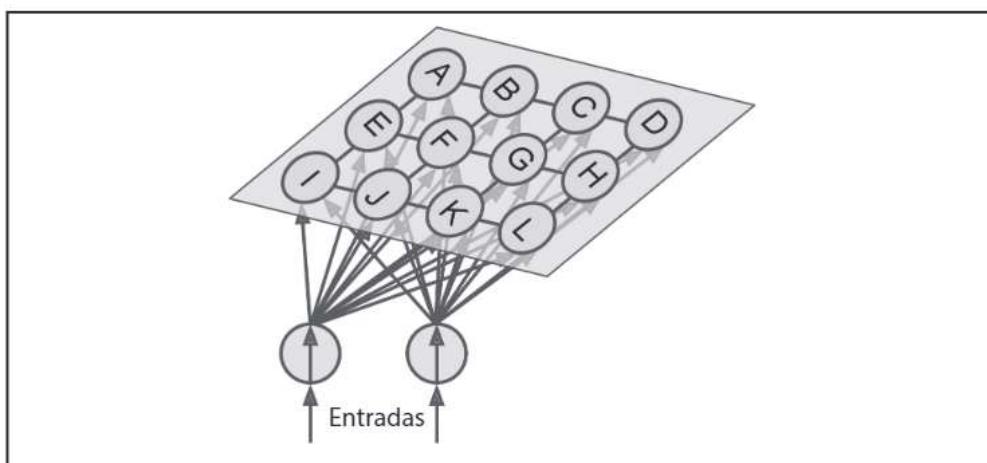


Figura E-5. Mapas auto-organizáveis

Uma vez que a rede é treinada, você pode alimentá-la com uma nova instância e isso ativará apenas um neurônio (ou seja, um ponto no mapa): aquele cujo vetor de peso está mais próximo do vetor de entrada. Em geral, as instâncias que estão próximas no espaço de entrada original ativarão os neurônios que estão próximos no mapa, tornando os SOMs úteis para visualização (em particular, você pode identificar facilmente os clusters no mapa), mas também para aplicativos como o reconhecimento da fala. Por exemplo, se cada instância representar a gravação de áudio de uma pessoa pronunciando uma vogal, então as diferentes pronúncias da vogal “a” ativarão os neurônios na mesma área do mapa, enquanto as instâncias da vogal “e” ativarão os neurônios em outra área, e sons intermediários geralmente ativarão neurônios intermediários no mapa.



Uma diferença importante das outras técnicas de redução da dimensionalidade discutidas no Capítulo 8 é que todas as instâncias são mapeadas para um número discreto de pontos no espaço de dimensão inferior (um ponto por neurônio). Esta técnica é melhor descrita como agrupamento em vez de redução da dimensionalidade quando existirem bem poucos neurônios.

O algoritmo de treinamento não é supervisionado. Ele funciona com todos os neurônios competindo uns contra os outros. Primeiro, todos os pesos são inicializados aleatoriamente. Em seguida, uma instância de treinamento é escolhida aleatoriamente e fornecida à rede. Todos os neurônios calculam a distância entre o vetor de peso e o vetor de entrada (isso é muito diferente dos neurônios artificiais que vimos até agora). O neurônio que mede a menor distância ganha e ajusta seu vetor de peso para estar um pouco mais próximo do vetor de entrada, tornando mais provável que ele ganhe competições futuras por outras entradas semelhantes a esta. Ele também recruta seus neurônios vizinhos e atualiza seu vetor de peso para ficar um pouco mais próximo do vetor de entrada (mas eles não atualizam seus pesos tanto quanto o neurônio vencedor). Em seguida, o algoritmo escolhe outra instância de treinamento e repete o processo várias vezes, fazendo com que os neurônios próximos se especializem gradualmente em entradas similares.<sup>4</sup>

---

<sup>4</sup> Você pode imaginar uma classe de crianças com habilidades semelhantes. Uma criança passa a ser um pouco melhor no basquete. Isso a motiva a praticar mais, especialmente com seus amigos. Depois de um tempo, esse grupo de amigos é tão bom no basquete que outras crianças não podem competir com eles. Mas tudo bem, porque as outras crianças se especializam em outros tópicos. Depois de um tempo, a classe está cheia de pequenos grupos especializados.



# Índice

## A

AdaBoost 196  
 Algoritmo 23  
     AdaGrad 306, 312  
     de aprendizado Perceptron 266  
     de IPCA 221  
     de Iteração Q-Value 476  
     de posicionamento dinâmico 331  
     de retropropagação 268, 283  
     de treinamento 264, 363, 378  
     de treinamento CART 171, 175  
     estocástico 222  
     genético 460  
     off-policy 479  
     out-of-core 121  
     Q-Learning 478  
     RMSProp 307, 312  
     ruim 23  
     TD Learning 478  
 Ambiente simulado 461  
 Amostragem estratificada 53  
 Análise dos Componentes Principais (PCA)  
     215  
 API 194  
     C++ 235  
     do Python 235  
     Python 271  
 Aprendizado 13, 81  
     baseado  
         em instância 18, 25  
         em modelo 19, 25  
     de regras de associação 13  
     em lote 15  
     não supervisionado 10  
     offline 15  
     online 16, 81, 86, 167

por Diferença Temporal (TD) 477  
 por reforço 14, 39  
 semissupervisionado 13  
 supervisionado 83  
 Área abaixo da curva (AUC) 96  
 Arquitetura  
     CNN  
         AlexNet 384  
         LeNet-5 382  
         GoogLeNet 385  
         ResNet 390  
     Árvores de Decisão 10, 74, 171, 181, 190  
         para regressão 179  
     Árvore-Extra 194  
 Autoencoder 431  
     de Remoção de Ruídos 444  
     do MNIST 435  
     empilhado 435  
     Esparsos 446  
     supercompleto 444  
     variacionais 449

## B

Bagging 189, 197  
 Boosting 196  
 Broadcasting 273

## C

Camadas  
     average pooling 381  
     convolucionais 371, 483  
     de gargalo 386  
     de pooling 371  
     desconvolucionais 393  
     max pooling 388  
     pooling 380

- Canais de transformação 71
- Características 158
  - esparsas 158
  - operacionais do receptor (ROC) 95
- Célula
  - de memória 400
  - GRU 423
  - LSTM 420
- Classe 89
  - negativa 89
  - prevista 89
  - real 89
- Classificação 105
  - de margens
    - largas 149
    - rígidas 150
    - suaves 151
  - multilabel 104
  - multioutput-multiclass 105
- Classificadores 149
  - binários 86, 97
  - lineares 149
  - multiclasses 97
- Cloud Machine Learning 327
- Cluster 336, 348
- Coeficiente
  - de correlação padrão 58
  - de Gini 177, 456
  - linear 110
- Compensação da precisão/revocação 91
- Complexidade computacional 114
- Compute Unified Device Architecture (CUDA) 327
- Conjunto 31, 50, 153
  - de dados 52, 153, 380
    - assimétricos 88
    - linearmente separável 153
    - MNIST 83, 195, 214, 270
    - não lineares 110, 153
    - simples 153
  - de exploração 55
  - de hiperparâmetros 356
  - de teste 31, 50, 85
  - de testes 275
  - de treinamento 4, 31, 50, 85, 109, 126, 275, 288, 313
- de validação 31, 73, 128, 275
- Contêineres de recurso 341
- Cronograma de aprendizado 122, 311
- Curva
  - de aprendizado 128
  - ROC 96
- D**
- Dados 23, 177
  - de séries 397
  - de treinamento 4, 23, 111, 138, 177, 276, 295, 300, 348
  - do MNIST 409
  - ruins 23
- Data
  - augmentation 107, 319, 320
  - snooping bias 51
- Decaimento exponencial 293
- Decision Stump 199
- Decomposição em Valores Singulares (SVD) 217
- DeepMind 480
- Dependências de controle 336
- Derivada parcial 118
- Desvio padrão 48
- Detecção de anomalias 13
- Diferenciação
  - automática 235
  - simbólica 242
- Divergência Kullback-Leibler 146, 447
- Dropout 314, 384, 418, 445
- Dying ReLUs 287
- E**
- Eficiência de parâmetro 277
- Elastic Net 137
- End-of-sequence token (EOS) 407
- Ensemble 200, 326
  - de Árvores Extremamente Aleatorizadas 194
  - Learning 75, 185
  - method 185
- Entropia 177
  - cruzada 145, 271, 274, 383, 448

Erro 51  
 de generalização 31, 51  
 de reconstrução 220, 225  
 de treinamento (MSE) 247  
 Médio Absoluto (MAE) 41  
 Médio Quadrático (MSE) 111, 242, 412, 433  
 Escala min-max 69  
 Escalonamento das características 68  
 Escopos do nome 250  
 Espaço de parâmetro 118  
 Estratégia 98  
     um contra todos (OvA) 98  
     um contra um (OvO) 98  
 Exploding gradients 283  
 Extração de características 12

**F**

Feature engeneering 27  
 Fila first-in first-out 343  
 Florestas Aleatórias 10, 74, 171  
 Follow The Regularized Leader (FTRL) 310  
 Forward pass 268, 379, 405  
 Fronteira de decisão 143, 214, 265  
 Frozen layers 299  
 Função 21, 62, 91, 117, 140, 155  
     convexa 117  
     de ativação 268, 286  
         ELU 436  
         exponential linear unit (ELU) 288  
         logística 448  
         logística (sigmoide) 466  
         ReLU 286  
         tanh 402  
     de Base Radial (RBF) 155  
     de custo 21, 117  
     de decisão 91  
     de exploração 480  
     degrau 263, 268  
     de Heaviside 264  
     de similaridade 155  
     de transformação 62  
     de utilidade 21  
     hinge loss 168, 169  
     logística 279  
     ReLU 269

sigmóide 140  
 softmax 144, 269  
 tangente  
     hiperbólica 279  
     tanh 268

**G**

Global average pooling 388  
 Google Brain 233  
 Gradient  
     Boosting 199  
     Clipping 294  
 Gradiente 397  
     Acelerado  
         de Nesterov (NAG) 305  
     de política 457  
     vanishing/exploding 397  
 Gradiente Descendente (GD) 109, 115, 152,  
     168, 241, 269, 304, 469  
     em Lote 119, 135, 241  
     em Minilote 124, 138, 244, 271  
     Estocástico (SGD) 86, 121, 141, 152, 266, 478  
 Grid search 76, 120, 155, 194, 223, 277

**H**

Histograma 49, 60

**I**

In-graph replication 357  
 Inicialização  
     aleatória 115, 284  
     He 286, 290, 410, 436  
     Xavier 285  
 Inter-op thread pools 334  
 Intra-op thread pools 334

**K**

Keep probability 316  
 Kernel 333, 373  
     da camada 272  
     de convolução 373, 382  
     linear 222  
     multithread 335  
     PCA (kPCA) 222

- polinomial de 2º grau 166
- RBF 197, 222
- RBF Gaussiano 157, 166
- sigmóide 222
- L**
  - Lei dos grandes números 187
  - Locally Linear Embedding (LLE) 225
- M**
  - Maldição da dimensionalidade 209
  - Manifold Learning 213, 225
  - Mapa de características 374, 385
  - Máquinas
    - de Vetores de Suporte (SVM) 10, 149, 222
    - Restritas de Boltzmann (RBM) 14, 301
  - Margem 161
    - rígida 161
    - suave 161
  - Matriz 88
    - de confusão 88, 100
    - de parâmetro 144
  - Medida
    - de desempenho 275
    - de impureza 173
    - de similaridade 18
  - Método
    - dos Mínimos Quadrados 112, 119, 241
    - Random
      - Patches 193
      - Subspaces 193
    - MLP 269, 277, 434
    - MNIST 209, 259, 382
  - Modelo 174
    - de caixa
      - branca 174
      - preta 174
    - gerador 431
    - linear 177
    - não paramétrico 177
    - paramétrico 178
  - N**
    - Neocognitron 370
    - Neurônio
      - artificial 262
  - biológico 287
  - de entrada 264
  - de viés 264, 267
  - recorrente 398
  - Normalização
    - de resposta local (LRN) 384
    - em Lote (BN) 290, 291, 303, 390, 419
  - Número de consultas por segundo (QPS) 357
  - O**
    - One-hot encoding 66
    - OpenAI gym 462
    - Operação
      - de leitura 349
      - dequeue [desenfileiramento] 344
      - enqueue [enfileiramento] 344
    - Otimização
      - Adam 308, 312
      - Momentum 303, 311
        - de Nesterov 305
      - restrita 162
    - Out-of-core learning 17
  - P**
    - Parada antecipada 138, 313
    - Paralelismo
      - de dados 361, 364
      - do modelo 359
    - Pasting 189, 197
    - PCA 181, 216, 218
      - Incremental (IPCA) 221
      - Randomizado 222
    - Percentis 48
    - Perceptron Multicamada (MLP) 259, 263, 266, 432
    - Pipeline 69
      - de dados 38
      - de entrada multithread 355
      - de treinamento 355
      - humano 91
    - Posicionador simples 331
    - Precisão do classificador 89
    - Problema
      - de programação quadrática (QP) 163
    - Internal Covariate Shift 290

- Processamento de linguagem natural (PNL) 424
- Processos de decisão de Markov (MDP)
  - 458, 473, 477
- Protocol buffers 339
- Protocolo gRPC 338
- R**
- Raiz do Erro Quadrático Médio (RMSE) 39, 72, 111
- Randomized Search 78
- Rastro alongado 50
- Recozimento simulado 122
- Redes Neurais 10, 174, 233, 250, 275, 301, 340, 357
  - artificiais 283
  - artificiais (RNA) 259
  - biológicas (RNB) 262
  - convolucionais (CNN) 369, 381, 391, 408
  - de crenças profundas [DBN] 14
  - feedforward (FNN) 269, 399
  - profundas (DNN) 28, 267, 270, 275, 283, 295, 309, 312, 480
  - recorrentes (RNN) 284, 397, 409, 415, 424
- Redução da dimensionalidade 12
- Regra
  - de Hebb 265
- Regressão 9, 10, 39, 153
  - de Floresta Aleatória 206
  - de Ridge 132
  - k-Nearest Neighbors 22
  - Lasso 135, 168, 310
  - Linear xiv, 10, 21, 72, 109, 124, 206, 242, 307
  - Logística 110, 139, 186, 223, 264
  - Multivariada 39
  - Polinomial 110, 125, 133, 153
  - Softmax 110, 144, 274
  - SVM 158
  - Univariante 39
- Regularização 28, 178
  - $\ell_1$  e  $\ell_2$  313
  - max-norm 317
  - Ridge 133
- Residual
  - learning 389
  - Network (ResNet) 389
- Retropropagação
- através do tempo (BPTT) 407
- truncada através do tempo 419
- Reverse-mode autodiff 243
- Reverse pass 379, 405
- Revocação 89
- Rótulos 8, 62, 80
- Ruído de amostragem 25
- S**
- Sabedoria das multidões 185
- SAMME 199
- Scikit-Learn xiv, 42, 83, 114, 151, 171, 188, 218, 234, 265, 376
- Sessão
  - distribuída 341
  - local 341
- SGD 123
- Sistema 37, 42
  - downstream 37, 42
- Skip connections 320
- Smoothing term 291
- Sobreajuste 27, 110
- Solucionadores off-the-shelf 163
- Stacking 204, 206
- Stochastic Gradient Boosting 203
- String kernels 157
- Subajuste 29, 72
- SVM 186, 197
- T**
- Taxa
  - de aprendizado 17, 115, 201, 304
  - adaptativa 307
  - de desconto r 467
  - de dilatação 392
  - de dropout 315, 317
  - de falsos positivos (FPR) 95
  - de variância explicada 218
  - de verdadeiros negativos (TNR) 95
  - de verdadeiros positivos (TPR) 89, 95
- Técnica 39, 201
  - de regularização 314, 319
- Encolhimento 201
- MapReduce 39
- TensorFlow xiv, 233, 259, 273, 292, 301, 325, 354, 359, 376, 415, 461, 469, 485

**T**eorema

- de convergência do Perceptron 265
- de Mercer 166
- Termo de polarização 110
- Tipagem duck typing 68
- Transfer learning 295
- Truque do kernel 154, 164, 222

**U**

- Unidades lineares retificadas (ReLU) 251, 384, 419

**V**

- Validação cruzada 32, 73, 87, 100, 128, 191, 225, 277, 289
- K-fold 73, 87
- Vanishing gradients 283, 289, 302, 388
- Variável de folga 163
- Vetor 111, 150
  - de características 111
  - de parâmetro 111
  - de suporte 150
  - gradiente 118, 146, 243
    - MSE 133
    - momentum 304
    - one-hot 486
    - subgradiente 136
  - Viés de falta de resposta 26
  - Violações de margem 151

**W**

- Word embeddings 425

**Z**

- Zero padding 372
- Zoológico de Modelos 301

## Sobre o Autor

---

Aurelien Géron é consultor de Aprendizado de Máquinas. Ex-funcionário do Google, ele liderou a equipe de classificação de vídeo do YouTube de 2013 a 2016. Ele também foi fundador e Diretor de Tecnologia da Wifirst de 2002 a 2012, o provedor de internet Wireless líder na França e fundador e Diretor de Tecnologia da Polyconseil em 2001, a empresa que agora gerencia o serviço Autolib de compartilhamento de carros elétricos.

Antes disso, ele trabalhou como engenheiro em vários segmentos: finanças (JP Morgan e Société Générale), defesa (Departamento de Defesa Nacional do Canadá) e assistência médica (transfusão de sangue). Publicou alguns livros técnicos (sobre C++, WiFi e Arquitetura da Internet), e foi professor de Ciência da Computação em uma escola francesa de engenharia.

Alguns fatos divertidos: ele ensinou seus três filhos a contar em binário com seus dedos (até 1023), estudou microbiologia e genética evolutiva antes de entrar para engenharia de software, e seu paraquedas não abriu no segundo pulo.

## Colofão

---

O animal na capa de *Mãos à Obra: Aprendizado de Máquinas com Scikit-Learn e TensorFlow* é a salamandra de fogo do Extremo Oriente (*Salamandra infraimmaculata*), um anfíbio encontrado no Oriente Médio. Elas possuem uma pele negra que apresenta manchas amarelas nas costas e cabeça, que são uma coloração de advertência destinada a manter os predadores à distância. Salamandras adultas podem ter mais de 30cm de comprimento.

As salamandras de fogo do Extremo Oriente vivem em matas subtropicais e florestas próximas a rios ou outros corpos de água doce, passando a maior parte das suas vidas em terra firme, mas depositando seus ovos na água.

Sua subsistência é basicamente constituída de uma dieta de insetos, minhocas e pequenos crustáceos, mas, ocasionalmente, se alimentam de outras salamandras. Os machos da espécie são conhecidos por viverem até 23 anos, enquanto as fêmeas podem viver até 21.

Embora ainda não estejam em risco de extinção, a população de salamandras de fogo do Extremo Oriente está diminuindo. As ameaças primárias incluem o represamento de rios (o que atrapalha a reprodução da espécie) e a poluição. Elas também estão ameaçadas pela recente introdução de peixes predatórios, como os peixes-mosquito, que foram destinados a controlar a população de mosquitos da região, mas também se alimentam de salamandras jovens.

Muitos animais das capas da O'Reilly estão ameaçados de extinção; todos eles são importantes para o mundo. Para ler mais sobre como ajudá-los, visite [animals.oreilly.com](http://animals.oreilly.com) (conteúdo em inglês).

