O'REILLY® ANAYA MULTIMEDIA

2ª Edición
Actualizado para TensorFlow 2

Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow

Conceptos, herramientas y técnicas para conseguir sistemas inteligentes

Funciona con

Aurélien Géron

Índice de contenidos

Introducción **23**

El *tsunami* del *machine learning* ... 23
Machine learning en tus proyectos ... 24
Objetivo y enfoque ... 24
Prerrequisitos ... 25
Hoja de ruta ... 25
Cambios en la segunda edición ... 26
Otros recursos ... 27
Convenciones ... 28
Ejemplos de código ... 29
Utilizar ejemplos de código ... 29
Sobre la imagen de cubierta ... 30

Parte I. Fundamentos del *machine learning* **31**

Capítulo 1. El paisaje del *machine learning* **33**

¿Qué es el *machine learning*? ... 34
¿Por qué utilizar *machine learning*? ... 35
Ejemplos de aplicaciones ... 37
Tipos de sistemas de *machine learning* ... 39
Aprendizaje supervisado/no supervisado ... 39
Aprendizaje por lotes y aprendizaje *online* ... 45
Aprendizaje basado en instancias frente a aprendizaje basado en modelos ... 48

Principales retos del *machine learning* 54
Cantidad insuficiente de datos de entrenamiento 54
Datos de entrenamiento no representativos 55
Datos de mala calidad 57
Características irrelevantes 57
Sobreajustar los datos de entrenamiento 57
Subajustar los datos de entrenamiento 60
Retroceder 60
Probar y validar 61
Ajuste de hiperparámetros y selección del modelo 61
Discrepancia de datos 62
Ejercicios 63

Capítulo 2. Proyecto de *machine learning* de principio a fin 65

Trabajar con datos reales 65
Tener una visión amplia 67
Enmarcar el problema 67
Seleccionar una medida del rendimiento 69
Comprobar las suposiciones 71
Obtener los datos 72
Crear el espacio de trabajo 72
Descargar los datos 75
Echar un vistazo rápido a la estructura de los datos 76
Crear un conjunto de prueba 80
Descubrir y visualizar los datos para tener un mayor entendimiento 84
Visualizar datos geográficos 84
Buscar correlaciones 86
Experimentar con combinaciones de atributos 89
Preparar los datos para algoritmos de *machine learning* 90
Limpiar datos 91
Manejar atributos de texto y categóricos 93
Transformadores personalizados 95
Escalado de características 96
Pipelines de transformación 97
Seleccionar un modelo y entrenarlo 99
Entrenar y evaluar con el modelo de entrenamiento 99
Una evaluación mejor utilizando la validación cruzada 101
Perfeccionar el modelo 103
Búsqueda exhaustiva 103
Búsqueda aleatorizada 105

Métodos de ensamblaje 106
Analizar los mejores modelos y sus errores 106
Evaluar el sistema en el conjunto de prueba 107
Lanzar, monitorizar y mantener el sistema 108
¡Pruébalo! 111
Ejercicios 111

Capítulo 3. Clasificación 113

MNIST 113
Entrenar un clasificador binario 115
Medidas del rendimiento 116
Medir la exactitud utilizando la validación cruzada 116
Matriz de confusión 118
Precisión y sensibilidad 119
Compensación precisión/sensibilidad 121
La curva ROC 124
Clasificación multiclase 127
Análisis de errores 129
Clasificación multietiqueta 133
Clasificación multisalida 134
Ejercicios 135

Capítulo 4. Entrenar modelos 137

Regresión lineal 138
La ecuación normal 140
Complejidad computacional 142
Descenso de gradiente 143
Descenso de gradiente por lotes 146
Descenso de gradiente estocástico 149
Descenso de gradiente por minilotes 152
Regresión polinomial 153
Curvas de aprendizaje 155
Modelos lineales regularizados 158
Regresión de arista 159
Regresión Lasso 161
Red elástica 164
Detención temprana 164
Regresión logística 166
Calcular probabilidades 166
Entrenamiento y función de pérdida 167

Límites de decisión 168
Regresión *softmax* 171
Ejercicios 174

Capítulo 5. Máquinas de vectores soporte 177

Clasificación SVM lineal 177
Clasificación de margen blando 178
Clasificación SVM no lineal 180
Kernel polinomial 182
Características de similitud 183
Kernel de función de base radial gaussiana 184
Complejidad computacional 185
Regresión SVM 186
Entre bambalinas 187
Predicciones y función de decisión 188
Objetivo de entrenamiento 189
Programación cuadrática 190
El problema dual 191
SVM kernelizadas 192
SVM *online* 194
Ejercicios 195

Capítulo 6. Árboles de decisión 197

Entrenar y visualizar árboles de decisión 197
Hacer predicciones 198
Estimación de probabilidades de clase 200
El algoritmo de entrenamiento CART 201
Complejidad computacional 202
¿Impureza de Gini o entropía? 202
Hiperparámetros de regularización 203
Regresión 204
Inestabilidad 206
Ejercicios 207

Capítulo 7. Ensamblaje y *random forests* 209

Clasificadores de votación 209
Bagging y *pasting* 212
Bagging y *pasting* en Scikit-Learn 213
Evaluación fuera de la bolsa 215

Parches aleatorios y subespacios aleatorios 216
Random forests 216
Extra-Trees 217
Importancia de las características 217
Boosting 218
AdaBoost 219
Gradient Boosting 222
Stacking 226
Ejercicios 228

Capítulo 8. Reducción de dimensionalidad 231

La maldición de la dimensionalidad 232
Enfoques principales para la reducción de dimensionalidad 233
Proyección 233
Aprendizaje de variedades 235
PCA 237
Preservar la varianza 237
Componentes principales 238
Proyección para bajar a *d* dimensiones 239
Utilizar Scikit-Learn 239
Ratio de varianza explicada 240
Elegir el número adecuado de dimensiones 240
PCA para compresión 241
PCA aleatorizado 242
PCA gradual 242
Kernel PCA 243
Seleccionar un *kernel* y ajustar hiperparámetros 244
LLE 246
Otras técnicas de reducción de dimensionalidad 248
Ejercicios 249

Capítulo 9. Técnicas de aprendizaje no supervisado 251

Agrupamiento 252
K-Medias 254
Límites de K-Medias 263
Utilizar agrupamiento para la segmentación de imágenes 264
Utilizar agrupamiento para el preprocesamiento 265
Utilizar agrupamiento para aprendizaje semisupervisado 267
DBSCAN 269
Otros algoritmos de agrupamiento 272

Mezclas gaussianas 273
Detección de anomalías utilizando mezclas gaussianas 279
Seleccionar el número de grupos 280
Modelos bayesianos de mezcla gaussiana 283
Otros algoritmos para detección de anomalías y de novedades 286
Ejercicios 287

Parte II. Redes neuronales y *deep learning* 31

Capítulo 10. Introducción a las redes neuronales artificiales con Keras 291

De las neuronas biológicas a las artificiales 292
Neuronas biológicas 293
Cálculos lógicos con neuronas 294
El perceptrón 295
El perceptrón multicapa y la retropropagación 299
PMC de regresión 302
PMC de clasificación 303
Implementación de PMC con Keras 305
Instalación de TensorFlow 2 306
Creación de un clasificador de imágenes utilizando la API secuencial 307
Creación de un PMC de regresión con la API secuencial 316
Creación de modelos complejos con la API funcional 317
Utilización de la API de subclasificación para crear modelos dinámicos 321
Guardar y restaurar un modelo 322
Utilización de retrollamadas 323
Uso de TensorBoard para visualización 324
Ajuste de los hiperparámetros de una red neuronal 327
Número de capas ocultas 331
Número de neuronas por capa oculta 332
Tasa de aprendizaje, tamaño de lote y otros hiperparámetros 332
Ejercicios 334

Capítulo 11. Entrenar redes neuronales profundas 337

Los problemas de desvanecimiento/explosión de gradientes 338
Inicialización de Glorot y He 339
Funciones de activación sin saturación 340

Normalización de lotes 344
Recorte de gradiente 350
Reutilizar redes preentrenadas 351
Aprendizaje por transferencia con Keras 352
Preentrenamiento no supervisado 354
Preentrenar con una tarea auxiliar 355
Optimizadores más rápidos 356
Optimización *Momentum* 356
Gradiente acelerado de Nesterov 358
AdaGrad 359
RMSProp 360
Optimización Adam y Nadam 361
Programación de la tasa de aprendizaje 364
Evitar el sobreajuste mediante la regularización 368
Regularizaciones ℓ_1 y ℓ_2 368
Dropout 369
Monte Carlo (MC) Dropout 372
Regularización *max-norm* 375
Resumen y directrices prácticas 375
Ejercicios 377

Capítulo 12. Modelos personalizados y entrenamiento con TensorFlow 379

Un *tour* rápido por TensorFlow 380
Utilizar TensorFlow como NumPy 383
Tensores y operaciones 383
Tensores y NumPy 385
Conversiones de tipos 385
Variables 386
Otras estructuras de datos 386
Personalizar modelos y algoritmos de entrenamiento 387
Funciones de pérdida personalizadas 387
Guardar y cargar modelos que contengan componentes personalizados 388
Funciones de activación, inicializadores, regularizadores y restricciones personalizados 390
Métricas personalizadas 391
Capas personalizadas 394
Modelos personalizados 397

Pérdidas y métricas basadas en componentes internos del modelo.... 399
Calcular gradientes utilizando diferenciación automática.................... 401
Bucles de entrenamiento personalizados 404
Grafos y funciones de TensorFlow 407
AutoGraph y trazado 409
Reglas de las funciones TF.. 410
Ejercicios 411

Capítulo 13. Cargar y preprocesar datos con TensorFlow 413

La API Data...414
Encadenar transformaciones 414
Mezclar los datos 416
Preprocesar los datos...419
Juntarlo todo...420
Precarga...420
Utilizar el conjunto de datos con tf.keras 422
El formato TFRecord 424
Archivos TFRecord comprimidos 424
Breve introducción a los búferes de protocolo............................. 425
Protobufs de TensorFlow 426
Cargar y analizar ejemplos.. 427
Manejar listas de listas usando el protobuf SequenceExample 428
Preprocesar las características de entrada....................................... 429
Codificar características categóricas usando vectores *one-hot* 430
Codificar características categóricas utilizando *embeddings* 432
Capas de preprocesamiento de Keras 436
TF Transform 438
El proyecto TensorFlow Datasets (TFDS) 440
Ejercicios 441

Capítulo 14. *Deep learning* para visión por ordenador usando redes neuronales convolucionales 443

La arquitectura de la corteza visual 444
Capas convolucionales...445
Filtros 447
Apilar múltiples mapas de características 448
Implementación en TensorFlow 450
Requisitos de memoria 452
Capas de *pooling* 453
Implementación en TensorFlow 455

Arquitecturas de RNC 457
LeNet-5 459
AlexNet 460
GoogLeNet 463
VGGNet..466
ResNet 466
Xception...469
SENet...471
Implementar una RNC ResNet-34 usando Keras 473
Utilizar modelos preentrenados desde Keras 474
Modelos preentrenados para aprendizaje por transferencia..................... 476
Clasificación y localización 478
Detección de objetos..480
Redes completamente convolucionales 482
You Only Look Once (YOLO) 484
Segmentación semántica 487
Ejercicios 491

Capítulo 15. Procesar secuencias utilizando RNR y RNC 493

Capas y neuronas recurrentes 494
Celdas de memoria...496
Secuencias de entrada y salida 497
Entrenamiento de RNR 498
Predicción de una serie temporal..499
Métricas de referencia..500
Implementación de una RNR sencilla 501
RNR profundas..502
Predecir varios pasos de tiempo más adelante 503
Manejar secuencias largas 507
Enfrentarse al problema de los gradientes inestables........................... 507
Enfrentarse al problema de la memoria a corto plazo..........................509
Ejercicios 517

Capítulo 16. Procesamiento de lenguaje natural con RNR y atención 519

Generar texto shakespeariano utilizando una RNR a nivel de carácter...... 520
Crear el conjunto de datos de entrenamiento..521
Cómo dividir un conjunto de datos secuenciales 522
Partir el conjunto de datos secuenciales en múltiples ventanas........... 523
Crear y entrenar el modelo de RNR a nivel de carácter......................... 525

Utilizar el modelo Char-RNN ... 525
Generar texto shakespeariano falso ... 525
RNR con estado ... 527
Análisis de sentimiento ... 529
Enmascaramiento ... 533
Reutilización de *embeddings* preentrenados ... 535
Una red codificador-descodificador para la traducción automática neuronal ... 536
RNR bidireccionales ... 539
Haz local (*beam search*) ... 540
Mecanismos de atención ... 542
Atención visual ... 545
Solo necesitas atención: la arquitectura Transformer ... 546
Innovaciones recientes en modelos de lenguaje ... 554
Ejercicios ... 557

Capítulo 17. Aprendizaje de representación y aprendizaje generativo utilizando autocodificadores y GAN 559

Representaciones de datos eficientes ... 560
Realizar PCA con un autocodificador lineal incompleto ... 562
Autocodificadores apilados ... 563
Implementación de un autocodificador apilado usando Keras ... 564
Visualizar las reconstrucciones ... 565
Visualizar el conjunto de datos Fashion MNIST ... 566
Preentrenamiento no supervisado utilizando autocodificadores apilados ... 567
Atar pesos ... 568
Entrenar autocodificadores de uno en uno ... 569
Autocodificadores convolucionales ... 571
Autocodificadores recurrentes ... 572
Autocodificador con eliminación de ruido ... 572
Autocodificadores dispersos ... 574
Autocodificadores variacionales ... 577
Generar imágenes de Fashion MNIST ... 581
Redes generativas antagónicas ... 583
Las dificultades de entrenar GAN ... 587
GAN convolucionales profundas ... 588
Crecimiento progresivo de GAN ... 591
StyleGAN ... 594
Ejercicios ... 596

Capítulo 18. Aprendizaje por refuerzo 599

Aprender a optimizar recompensas ... 600
Búsqueda de políticas ... 601
Introducción a OpenAI Gym ... 603
Políticas de redes neuronales ... 607
Evaluar acciones: el problema de la asignación de crédito ... 608
Gradientes de política ... 610
Procesos de decisión de Markov ... 614
Aprendizaje por diferencia temporal ... 618
Aprendizaje Q-Learning ... 619
Políticas de exploración ... 621
Q-Learning aproximado y Q-Learning profundo ... 622
Implementación de Q-Learning profundo ... 623
Variantes del Q-Learning profundo ... 627
Objetivos de valores Q fijos ... 627
Double DQN ... 628
Repetición de experiencia priorizada ... 629
Dueling DQN ... 630
La biblioteca TF-Agents ... 631
Instalación de TF-Agents ... 631
Entornos de TF-Agents ... 632
Especificaciones del entorno ... 633
Envolturas de entorno y preprocesamiento de Atari ... 634
Arquitectura de entrenamiento ... 636
Creación de la red Q profunda ... 638
Creación del agente DQN ... 640
Creación del búfer de repetición y el observador correspondiente ... 641
Creación de métricas de entrenamiento ... 643
Creación del *driver* de recogida ... 643
Creación del conjunto de datos ... 645
Creación del bucle de entrenamiento ... 648
Visión general de algunos algoritmos de aprendizaje por refuerzo populares ... 649
Ejercicios ... 651

Capítulo 19. Entrenar y desplegar modelos de TensorFlow a escala 653

Servir un modelo de TensorFlow ... 654
Utilizar TensorFlow Serving ... 654

Crear un servicio de predicción en AI Platform de GCP 662
Utilización del servicio de predicción 667
Desplegar un modelo en un dispositivo móvil o incrustado 670
Uso de GPU para acelerar la computación 673
Conseguir tu propia GPU 674
Usar una máquina virtual equipada con GPU 676
Colaboratory 677
Gestionar la RAM de la GPU 678
Colocar operaciones y variables en dispositivos 680
Ejecución paralela en múltiples dispositivos 682
Entrenar modelos en múltiples dispositivos 684
Paralelismo del modelo 685
Paralelismo de datos 687
Entrenar y escalar con la API Distribution Strategies 691
Entrenar un modelo en un clúster de TensorFlow 693
Ejecutar tareas de entrenamiento grandes en AI Platform de Google Cloud 696
Ajuste de hiperparámetros de caja negra en AI Platform 697
Ejercicios 699
¡Gracias! 700

Parte III. Apéndices 701

Apéndice A. Soluciones de los ejercicios 703

Capítulo 1: El paisaje del *machine learning* 703
Capítulo 2: Proyecto de *machine learning* de principio a fin 705
Capítulo 3: Clasificación 705
Capítulo 4: Entrenar modelos 705
Capítulo 5: Máquinas de vectores soporte 708
Capítulo 6: Árboles de decisión 709
Capítulo 7: Ensamblaje y *random forests* 710
Capítulo 8: Reducción de dimensionalidad 712
Capítulo 9: Técnicas de aprendizaje no supervisado 713
Capítulo 10: Introducción a las redes neuronales artificiales con Keras 715
Capítulo 11: Entrenar redes neuronales profundas 717
Capítulo 12: Modelos personalizados y entrenamiento con TensorFlow 719
Capítulo 13: Cargar y preprocesar datos con TensorFlow 721
Capítulo 14: *Deep learning* para visión por ordenador usando redes neuronales convolucionales 724
Capítulo 15: Procesar secuencias utilizando RNR y RNC 727
Capítulo 16: Procesamiento de lenguaje natural con RNR y atención 729

Capítulo 17: Aprendizaje de representación y aprendizaje generativo utilizando autocodificadores y GAN 731
Capítulo 18: Aprendizaje por refuerzo 733
Capítulo 19: Entrenar y desplegar modelos de TensorFlow a escala 736

Apéndice B. Lista de comprobación de proyectos de *machine learning* 739

Enmarcar el problema y tener una visión amplia 739
Obtener los datos 740
Explorar los datos 740
Preparar los datos 741
Seleccionar modelos prometedores 742
Perfeccionar el sistema 742
Presentar la solución 743
¡Lanzar! 743

Apéndice C. Problema dual con SVM 745

Apéndice D. Diferenciación automática 749

Diferenciación manual 749
Aproximación mediante diferencias finitas 750
Diferenciación automática hacia delante 751
Diferenciación automática inversa 753

Apéndice E. Otras arquitecturas de RNA populares 757

Redes de Hopfield 757
Máquinas de Boltzmann 758
Máquinas de Boltzmann restringidas 760
Redes de creencia profunda 761
Mapas autoorganizados 764

Apéndice F. Estructuras de datos especiales 767

Cadenas 767
Tensores irregulares 768
Tensores dispersos 769
Matrices tensoriales 770
Conjuntos 771
Colas 772

Apéndice G. Grafos de TensorFlow 773

Funciones TF y funciones concretas....773
Explorar definiciones y grafos de función....775
Atención al trazado....776
Utilizar AutoGraph para capturar estructuras de control....778
Manejar variables y otros recursos en funciones TF....779
Utilizar funciones TF con tf.keras (o no)....780

Índice alfabético 783

- Repositorios de datos abiertos populares:
  - UC Irvine Machine Learning Repository (`http://archive.ics.uci.edu/ml/`).
  - Conjuntos de datos de Kaggle (`https://www.kaggle.com/datasets`).
  - Conjuntos de datos AWS de Amazon (`https://registry.opendata.aws/`).
- Portales meta (ofrecen listas de repositorios de datos abiertos):
  - Data Portals (`http://dataportals.org/`).
  - OpenDataMonitor (`http://opendatamonitor.eu/`).
  - Quandl (`http://quandl.com/`).
- Otras páginas que ofrecen listas con muchos repositorios de datos abiertos populares:
  - Lista de conjuntos de datos para *machine learning* de Wikipedia (`https://homl.info/9`).
  - Quora.com (`https://homl.info/10`).
  - El *subreddit* de conjuntos de datos (`https://www.reddit.com/r/datasets`).

En este ejemplo vamos a utilizar el conjunto de datos de los precios de las casas de California (*California Housing Prices*) del repositorio StatLib[2] (véase la figura 2.1). Este conjunto de datos se basa en los datos del censo de California de 1990. No es exactamente reciente (en aquella época, una casa bonita en el Área de la Bahía aún era asequible), pero tienen muchas cualidades para el aprendizaje, así que vamos a fingir que son datos recientes. Para fines didácticos, he añadido un atributo categórico y he eliminado algunas características.

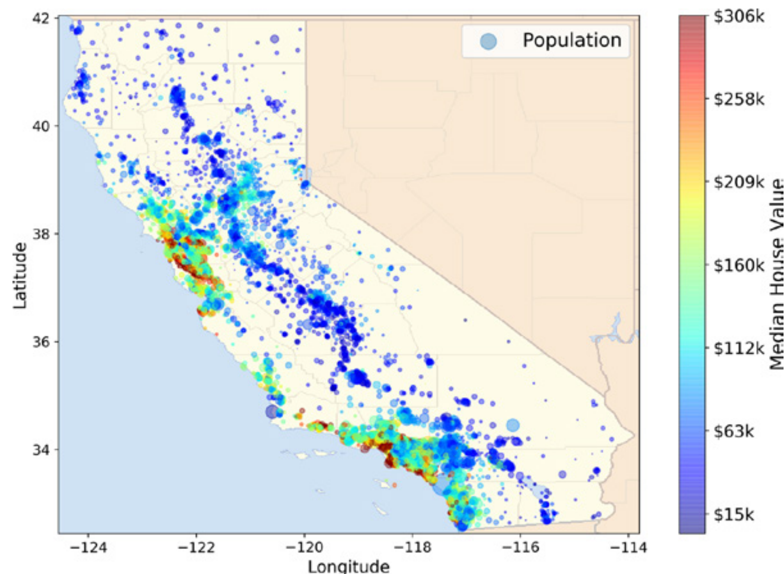

Figura 2.1. Precios de las casas de California.

2. El conjunto de datos original apareció en la publicación de R. Kelley Pace y Ronald Barry, "Sparse Spatial Autoregressions", en *Statistics & Probability Letters 33*, n.º 3 (1997): 291-297.

Tener una visión amplia

¡Bienvenido al Departamento de vivienda del *machine learning*! Tu primera tarea será utilizar el censo de California para crear un modelo de precios de casas en el estado. Estos datos incluyen métricas como la población, los ingresos medios y los precios medios de las casas para cada grupo de bloques en California. Los grupos de bloques son la unidad geográfica más pequeña para la que la Oficina del censo de EE. UU. publica datos simples (un grupo de bloques suele tener una población de entre 600 y 3.000 personas). Por acortar, vamos a llamarlos "distritos".

El modelo debería aprender a partir de estos datos y ser capaz de predecir el precio medio de las casas en cualquier distrito si se le dan todas las demás métricas.

> **Truco:** Como eres un científico de datos bien organizado, lo primero que deberías hacer es sacar tu lista de comprobación de proyectos de *machine learning*. Puedes empezar con la que aparece en el apéndice B; debería servir bastante bien para la mayoría de proyectos de *machine learning*, pero asegúrate de adaptarla a tus necesidades. En este capítulo, pasaremos por muchos puntos de la lista, pero también nos saltaremos algunos, bien porque son evidentes, bien porque los trataremos en capítulos posteriores.

Enmarcar el problema

Lo primero que debes preguntar a tu jefe es cuál es exactamente el objetivo empresarial. Es probable que crear un modelo no sea la meta final. ¿Cómo espera la empresa utilizar ese modelo y beneficiarse de él? Conocer el objetivo es importante porque determinará la manera de enmarcar el problema, qué algoritmos seleccionar, qué medida de rendimiento utilizar para evaluar el modelo y cuánto esfuerzo se dedicará a ajustarlo.

El jefe responde que la salida del modelo (una predicción del precio medio de las casas de un distrito) se introducirá en otro sistema de *machine learning* (véase la figura 2.2), junto con muchas otras señales.[3] Este sistema descendente determinará si merece la pena invertir en un área determinada o no. Hacer bien esto es fundamental, ya que afecta de forma directa a los ingresos.

> ***Pipelines***
>
> Una secuencia de datos que procesan componentes se llama *pipeline* de datos. Las *pipelines* son muy comunes en los sistema de *machine learning*, puesto que hay muchos datos que manipular y muchas transformaciones de datos que aplicar.

3. Una información introducida en un sistema de *machine learning* suele llamarse "señal", en referencia a la teoría de la información de Claude Shannon, que desarrolló en Bell Labs para mejorar las telecomunicaciones. Su teoría: nos conviene una relación señal/ruido alta.

La función de pérdida sobre el conjunto de entrenamiento completo es la pérdida media sobre todas las instancias de entrenamiento. Puede escribirse en una sola expresión llamada pérdida logística, que se muestra en la ecuación 4.17.

Ecuación 4.17. Función de pérdida de regresión logística (pérdida logística).

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\Sigma_{i=1}^{m}\left[y^{(i)}log\left(\hat{p}^{(i)}\right) + \left(1 - y^{(i)}\right)log\left(1 - \hat{p}^{(i)}\right)\right]$$

La mala noticia es que no se conoce ninguna ecuación de forma cerrada para calcular el valor de $\boldsymbol{\theta}$ que minimiza esta función de pérdida (no hay equivalente de la ecuación normal). La buena noticia es que esta función de pérdida es convexa, así que está garantizado que el descenso de gradiente (o cualquier otro algoritmo de optimización) encontrará el mínimo global (si la tasa de aprendizaje no es demasiado grande y esperas el tiempo suficiente). Las derivadas parciales de la función de pérdida con respecto al jº parámetro de modelo θ_j están dadas por la ecuación 4.18.

Ecuación 4.18. Derivadas parciales de la función de pérdida logística.

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(\sigma\left(\boldsymbol{\theta}^\intercal \mathbf{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Esta ecuación se parece mucho a la ecuación 4.5: para cada instancia, computa el error de predicción y los multiplica por el jº valor de característica y, después, calcula la media sobre todas las instancias de entrenamiento. Una vez que tienes el vector de gradiente que contiene todas las derivadas parciales, puedes utilizarlo en el algoritmo de descenso de gradiente por lotes. Ya está: ahora sabes cómo entrenar un modelo de regresión logística. Para el descenso de gradiente estocástico, tomarías las instancias de una en una, mientras que, para el descenso de gradiente por minilotes, utilizarías un minilote cada vez.

Límites de decisión

Vamos a utilizar el conjunto de datos iris para ilustrar la regresión logística. Se trata de un famoso conjunto de datos que contienen la longitud y la anchura del pétalo y el sépalo de 150 flores de iris de tres especies diferentes: *Iris setosa*, *Iris versicolor* e *Iris virginica* (véase la figura 4.22).

Vamos a intentar crear un clasificador para detectar el tipo *Iris virginica* que se base solo en la característica de la anchura del pétalo. Primero, cargamos los datos:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, 3:] # anchura del pétalo
>>> y = (iris["target"] == 2).astype(np.int) # 1 si es Iris virginica, 0 si no
```


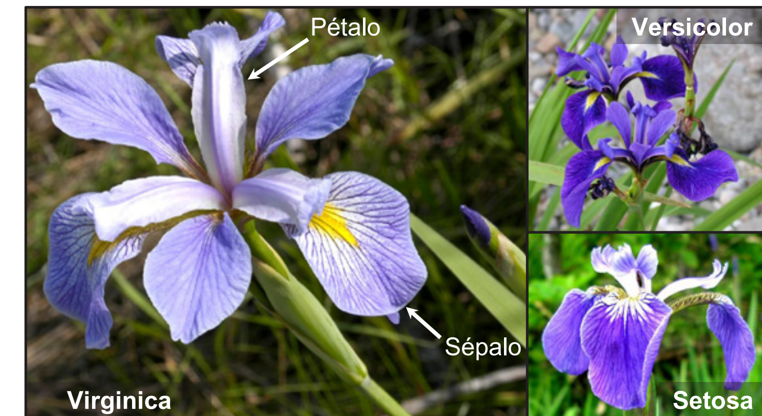

Figura 4.22. Flores de tres especies de la planta iris.[14]

Ahora vamos a entrenar un modelo de regresión logística:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Vamos a echar un vistazo a las probabilidades estimadas del modelo para flores con anchuras de pétalos que vayan de 0 cm a 3 cm (véase la figura 4.23):[15]

```
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")
# + más código Matplotlib para hacer que la imagen quede bonita
```

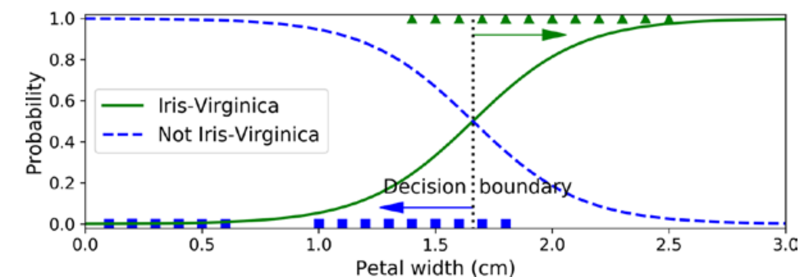

Figura 4.23. Probabilidades estimadas y límite de decisión.

14. Las fotos reproducidas corresponden a páginas de Wikipedia. Foto de *Iris virginica* de Frank Mayfield (Creative Commons BY-SA 2.0 (`https://creativecommons.org/licenses/by-sa/2.0/`)), foto de *Iris versicolor* de D. Gordon E. Robertson (Creative Commons BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0/`)), foto de *Iris setosa* de dominio público.

15. La función de NumPy `reshape()` permite que una dimensión sea -1, lo que significa "no especificado": el valor se infiere de la longitud de la matriz y de las dimensiones restantes.

Árboles de decisión

Al igual que las SVM, los árboles de decisión son algoritmos de *machine learning* versátiles que pueden realizar tanto tareas de clasificación como de regresión, incluso tareas multisalida. Son algoritmos potentes, capaces de ajustar conjuntos de datos complejos. Por ejemplo, en el capítulo 2, entrenamos un modelo `DecisionTreeRegressor` con el conjunto de datos de los precios de la vivienda en California, ajustándolo a la perfección (en realidad, sobreajustándolo).

Los árboles de decisión son también los componentes fundamentales de los *random forests* (véase el capítulo 7), que son uno de los algoritmos de *machine learning* más potentes que existen en la actualidad.

En este capítulo empezaremos explicando cómo entrenar, visualizar y hacer predicciones con árboles de decisión. Después veremos el algoritmo de entrenamiento CART, utilizado por Scikit-Learn, y comentaremos cómo regularizar y utilizar árboles de decisión para tareas de regresión. Por último, hablaremos de algunas de las limitaciones de los árboles de decisión.

Entrenar y visualizar árboles de decisión

Para entender los árboles de decisión, vamos a construir uno para ver cómo hace las predicciones. El siguiente código entrena un `DecisionTreeClassifier` con el conjunto de datos iris (véase el capítulo 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # longitud y anchura del pétalo
```

un poco mejor con esas instancias, y así sucesivamente. El gráfico de la derecha representa la misma secuencia de predictores, pero la tasa de aprendizaje está dividida por la mitad (es decir, los pesos de instancias que se han clasificado mal se potencian la mitad en cada iteración). Como puedes ver, esta técnica de aprendizaje secuencial tiene algunas similitudes con el descenso de gradiente, pero, en vez de ajustar los parámetros de un solo predictor para minimizar la función de pérdida, AdaBoost añade predictores al ensamble, mejorándolo de manera gradual.

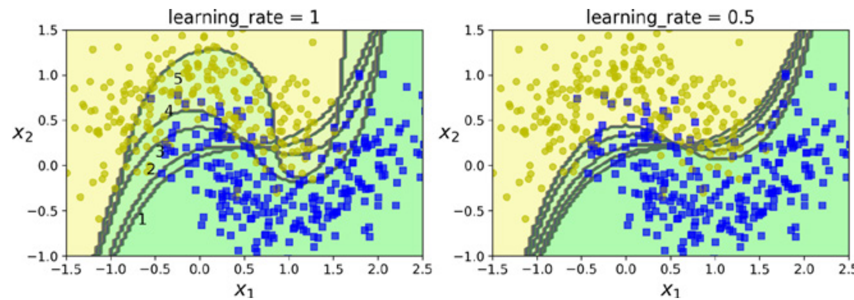

Figura 7.8. Límites de decisión de predictores consecutivos.

Una vez que todos los predictores se han entrenado, el ensamble hace predicciones de manera muy similar al *bagging* o el *pasting*, salvo porque los predictores tienen diferentes pesos, dependiendo de su exactitud general en el conjunto de entrenamiento ponderado.

> **Advertencia:** Esta técnica de aprendizaje secuencial tiene un inconveniente importante: no puede usarse en paralelo (o solo en parte), puesto que cada predictor solo puede entrenarse después de que el predictor anterior se haya entrenado y evaluado. Como resultado, no escala tan bien como el *bagging* o el *pasting*.

Vamos a fijarnos con más detalle en el algoritmo AdaBoost. Cada peso de instancia $w^{(i)}$ se configura inicialmente como $1/m$. Se entrena un primer predictor y su tasa de error ponderada r_1 se calcula en el conjunto de entrenamiento; véase la ecuación 7.1.

Ecuación 7.1. Tasa de error ponderada del j^o predictor.

$$r_j = \frac{\displaystyle\sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^{m} w^{(i)}}{\displaystyle\sum_{i=1}^{m} w^{(i)}} \quad \text{donde } \hat{y}_j^{(i)} \text{ es la predicción del } j^{\text{o}} \text{ predictor para la } i^{\text{a}} \text{ instancia.}$$

El peso del predictor α_j se calcula a continuación utilizando la ecuación 7.2, donde η es el hiperparámetro de la tasa de aprendizaje (se establece como 1 por defecto).[15]

15. El algoritmo AdaBoost original no usa un hiperparámetro de tasa de aprendizaje.

Cuanto más exacto sea el predictor, más alto será su peso. Si está haciendo suposiciones de forma aleatoria, entonces su peso se acercará a cero. Sin embargo, si se equivoca muy a menudo (es decir, es menos exacto que las suposiciones aleatorias), entonces su peso será negativo.

Ecuación 7.2. Peso del predictor.

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

A continuación, el algoritmo AdaBoost actualiza los pesos de las instancias, utilizando la ecuación 7.3, lo cual potencia los pesos de las instancias mal clasificadas.

Ecuación 7.3. Regla de actualización de pesos.

$$\text{para } i = 1, 2, \cdots, m$$
$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{si } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp\left(\alpha_j\right) & \text{si } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Todos los pesos de instancias se normalizan (es decir, se dividen entre $\Sigma^m_{i=1} w^{(i)}$).

Por último, se entrena un nuevo predictor utilizando los pesos actualizados y se repite todo el proceso (se calcula el peso del nuevo predictor, se actualizan los pesos de instancias, entonces se entrena otro predictor, y así sucesivamente). El algoritmo se detiene cuando se alcanza el número de predicciones deseado o cuando se encuentra un predictor perfecto.

Para hacer predicciones, AdaBoost simplemente computa las predicciones de todos los predictores y las pondera utilizando los pesos α_j del predictor. La clase predicha es la que recibe la mayoría de los votos ponderados (véase la ecuación 7.4).

Ecuación 7.4. Predicciones de AdaBoost.

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x}) = k}}^{N} \alpha_j \quad \text{donde } N \text{ es el número de predictores.}$$

Scikit-Learn utiliza una versión multiclase de AdaBoost llamada SAMME (`https://homl.info/27`)[16] (*Stagewise Additive Modeling using a Multiclass Exponential loss function*). Cuando solo hay dos clases, SAMME es equivalente a AdaBoost. Si los predictores pueden estimar probabilidades de clases (es decir, si tienen un método `predict_proba()`), Scikit-Learn puede usar una variante de SAMME llamada SAMME.R (la R significa "Real"), que depende de las probabilidades de clase más que de las predicciones y, por lo general, tiene mejor rendimiento.

El siguiente código entrena un clasificador AdaBoost basado en 200 árboles de un solo nivel utilizando la clase `AdaBoostClassifier` de Scikit-Learn (como ya supondrás, también hay una clase `AdaBoostRegressor`). Un árbol de un solo nivel es un árbol de decisión con `max_depth=1`; dicho de otro modo, un árbol compuesto de un solo nodo de decisión más dos nodos terminales. Este es el estimador de base predeterminado para la clase `AdaBoostClassifier`:

16. Para obtener más detalles, consulta Ji Zhu *et al.*, "Multi-Class AdaBoost", *Statistics and Its Interface* 2, n.º 3 (2009): 349–360.

¿Ya estás convencido? No deberías: ¡he hecho trampa! He probado varias configuraciones hasta que he encontrado una que ha mostrado una gran mejora. Si intentas cambiar las clases o la semilla aleatoria, verás que, por lo general, la mejora se detiene, incluso desaparece o se invierte. Lo que he hecho se llama "torturar a los datos hasta que confiesen". Cuando un artículo parece demasiado positivo, deberías sospechar: quizá la nueva técnica llamativa no ayude mucho, en realidad (de hecho, puede incluso empeorar el rendimiento), pero los autores han probado muchas variantes y han informado solo de los mejores resultados (que podrían ser pura suerte), sin mencionar cuántos fracasos han tenido por el camino. La mayor parte del tiempo no es algo que se haga con malicia, pero es parte de la razón por la que muchos resultados en la ciencia no pueden reproducirse nunca.

¿Por qué he hecho trampa? Resulta que el aprendizaje por transferencia no funciona muy bien con redes densas pequeñas, presumiblemente porque las redes pequeñas aprenden pocos patrones, que tienen pocas probabilidades de ser útiles en otras tareas. El aprendizaje por transferencia funciona mejor con redes convolucionales profundas, que tienden a aprender detectores de características que son mucho más generales (sobre todo en las capas inferiores). Volveremos a hablar del aprendizaje por transferencia en el capítulo 14, utilizando las técnicas que acabamos de ver (¡y esta vez no haré trampa, lo prometo!).

Preentrenamiento no supervisado

Supongamos que quieres abordar una tarea compleja para la que no tienes muchos datos de entrenamiento etiquetados, pero, por desgracia, no puedes encontrar un modelo entrenado en una tarea similar. ¡No pierdas la esperanza! Primero, deberías intentar reunir más datos de entrenamiento etiquetados, pero, si no puedes, quizá aún puedas realizar un preentrenamiento no supervisado (véase la figura 11.5). Lo cierto es que, a menudo, es barato recopilar ejemplos de entrenamiento no etiquetados, pero es caro etiquetarlos. Si puedes reunir un montón de datos de entrenamiento sin etiquetar, puedes intentar utilizarlos para entrenar un modelo no supervisado, como un autocodificador o una red generativa antagónica (consulta el capítulo 17). Después, puedes reutilizar las capas inferiores del autocodificador o las capas inferiores de la discriminadora de la GAN, añadir la capa de salida para tu tarea en la parte superior y ajustar la red final utilizando aprendizaje supervisado (es decir, con los ejemplos de entrenamiento etiquetados).

Esta técnica fue la que Geoffrey Hinton y su equipo utilizaron en 2006 y que llevó al resurgimiento de las redes neuronales y al éxito del *deep learning*. Hasta 2010, el preentrenamiento no supervisado, por lo general con máquinas de Boltzmann restringidas (véase el apéndice E), era la norma para las redes profundas y, hasta que no se encontró una solución para el problema del desvanecimiento de gradientes, no se popularizó el entrenamiento de RNP solo con aprendizaje supervisado. El preentrenamiento no supervisado (que hoy utiliza autocodificadores o GAN en vez de máquinas de Boltzmann restringidas) sigue siendo una buena opción cuando tenemos que resolver una tarea compleja, no hay ningún modelo similar que podamos reutilizar y hay pocos datos de entrenamiento etiquetados, pero muchos sin etiquetar.

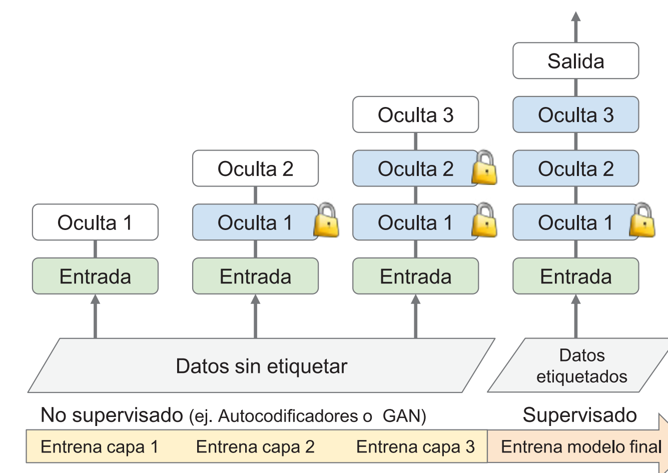

Figura 11.5. En el entrenamiento no supervisado, un modelo se entrena en los datos sin etiquetar (o en todos los datos) utilizando una técnica de aprendizaje no supervisado y, a continuación, se ajusta para la tarea final con los datos etiquetados utilizando una técnica de aprendizaje supervisado; la parte no supervisada puede entrenar las capas de una en una, como se muestra aquí, o puede entrenar directamente el modelo completo.

Ten en cuenta que, en los inicios del *deep learning*, era difícil entrenar modelos profundos, así que la gente usaba una técnica llamada "preentrenamiento *greedy* por capas" (que se representa en la figura 11.5). Primero entrenaban un modelo no supervisado con una sola capa, por lo general una máquina de Boltzmann restringida, después congelaban esa capa y añadían otra encima de ella, luego entrenaban el modelo otra vez (a los efectos, solo entrenaban la capa nueva), a continuación, congelaban la capa nueva y añadían otra encima de ella, volvían a entrenar el modelo, y así sucesivamente. Hoy en día, las cosas son mucho más sencillas: por lo general, la gente entrena el modelo no supervisado completo de una sola vez (es decir, en la figura 11.5, empieza directamente en el paso tres) y utiliza autocodificadores o GAN en vez de máquinas de Boltzmann restringidas.

Preentrenar con una tarea auxiliar

Si no tienes muchos datos de entrenamiento etiquetados, una última opción es entrenar una primera red neuronal con una tarea auxiliar para la que puedas obtener o generar con facilidad datos de entrenamiento etiquetados y, después, reutilizar las capas inferiores de esa red para la tarea real. Las capas inferiores de la primera red neuronal aprenderán detectores de características que probablemente podrá reutilizar la segunda red neuronal.

Por ejemplo, si quieres crear un sistema para reconocer caras, puede que solo tengas unas pocas fotos de cada individuo; está claro que no es suficiente para entrenar un buen clasificador. Recopilar cientos de miles de fotos de cada persona no resultaría práctico. Sin embargo, podrías recopilar un montón de imágenes de personas aleatorias en la web y entrenar

Como ha ocurrido antes, no sería demasiado difícil empaquetar toda esta lógica en una bonita clase independiente. Su método `adapt()` tomaría una muestra de los datos y extraería todas las categorías distintas que contuviese. Crearía una tabla de consulta para asignar cada categoría a su índice (incluyendo categorías desconocidas usando agrupamientos OOV). Después, su método `call()` usaría la tabla de consulta para asignar las categorías de entrada a sus índices. Vale, más buenas noticias: para cuando leas esto, es probable que Keras incluya una capa llamada `keras.layers.TextVectorization`, que será capaz de hacer justo eso: su método `adapt()` extraerá el vocabulario de una muestra de datos y su método `call()` convertirá cada categoría en su índice en el vocabulario. Podrías añadir esta capa al principio del modelo, seguida de una capa `Lambda` que aplicaría la función `tf.one_hot()`, si quieres convertir estos índices en vectores *one-hot*.

Sin embargo, puede que esta no sea la mejor solución. El tamaño de cada vector *one-hot* es la longitud del vocabulario más el número de agrupamientos OOV. Eso está bien cuando solo hay unas pocas categorías posibles, pero, si el vocabulario es grande, es mucho más eficiente codificarlas utilizando *embeddings*.

> **Truco:** Como regla de oro, si el número de categorías es inferior a 10, por lo general, la codificación *one-hot* es la mejor opción (¡aunque el uso puede variar!). Si el número de categorías es superior a 50 (que suele ser el caso cuando utilizamos agrupamientos de *hash*), suele ser preferible utilizar *embeddings*. Si tienes entre 10 y 50 categorías, te conviene experimentar con ambas opciones y ver cuál funciona mejor para tu caso de uso.

Codificar características categóricas utilizando *embeddings*

Un *embedding* es un vector denso entrenable que representa una categoría. Por defecto, los *embeddings* se inicializan de manera aleatoria, así que, por ejemplo, la categoría `"NEAR BAY"` podría estar representada inicialmente por un vector aleatorio, como `[0.131, 0.890]`, mientras que la categoría `"NEAR OCEAN"` podría estar representada por otro vector aleatorio, como `[0.631, 0.791]`. En este ejemplo, utilizamos *embeddings* 2D, pero el número de dimensiones es un hiperparámetro que puedes ajustar. Puesto que estos *embeddings* son entrenables, mejorarán de forma gradual durante el entrenamiento y, como representan categorías bastante similares, el descenso de gradiente acabará acercándolas, mientras que tenderá a alejarlas del *embedding* de la categoría `"INLAND"` (véase la figura 13.4). En realidad, cuanto mejor sea la representación, más fácil será para la red neuronal hacer predicciones exactas, así que el entrenamiento tiende a hacer que los *embeddings* sean representaciones útiles de las categorías. Esto se denomina "aprendizaje de representaciones" (veremos otros tipos de aprendizaje de representaciones en el capítulo 17).

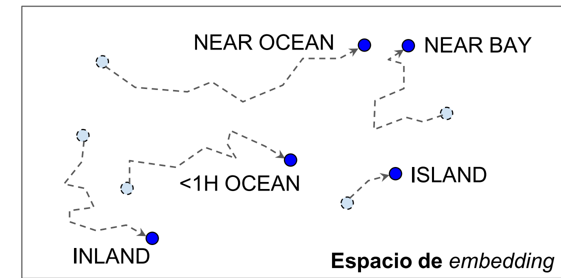

Figura 13.4. Los *embeddings* mejorarán de forma gradual durante el entrenamiento.

Embeddings de palabras

Los *embeddings* no solo suelen ser representaciones útiles para la tarea que tenemos entre manos, sino que, con frecuencia, los mismos *embeddings* pueden reutilizarse con éxito para otras tareas. El ejemplo más común son los *embeddings* de palabras (es decir, *embeddings* de palabras individuales): cuando trabajamos en una tarea de procesamiento del lenguaje natural, a menudo es mejor reutilizar *embeddings* de palabras preentrenados que entrenar unos propios. La idea de utilizar vectores para representar palabras data de los años 60 y se han utilizado muchas técnicas sofisticadas para generar vectores útiles, incluyendo el uso de redes neuronales. Pero las cosas despegaron realmente en 2013, cuando Tomaš Mikolov y otros investigadores de Google publicaron un artículo (`https://homl.info/word2vec`)[9] que describía una técnica eficiente para aprender *embeddings* de palabras usando redes neuronales, superando de forma significativa el rendimiento de intentos anteriores. Esto les permitía aprender *embeddings* en un corpus de texto muy grande: entrenaron una red neuronal para predecir las palabras cerca de cualquier palabra dada y obtuvieron una cantidad asombrosa de *embeddings* de palabras. Por ejemplo, los sinónimos tenían *embeddings* muy cercanos, y palabras relacionadas a nivel semántico, como Francia, España e Italia, acaban agrupadas juntas.

Sin embargo, no solo tiene que ver con la proximidad: los *embeddings* de palabras también se organizaban a lo largo de ejes significativos en el espacio de *embedding*. Veamos un ejemplo famoso: si computamos Rey – Hombre + Mujer (sumando y restando los vectores de *embedding* de estas palabras), el resultado se acercará mucho al *embedding* de la palabra "reina" (véase la figura 13.5). Dicho de otro modo, ¡los *embeddings* de palabras codifican el concepto de género! De manera similar, puedes computar Madrid – España + Francia y el resultado se acercará a París, lo que parece demostrar que la noción de capital también estaba codificada en los *embeddings*.

Por desgracia, a veces los *embeddings* de palabras capturan nuestros peores sesgos. Por ejemplo, aunque pueden aprender correctamente que Hombre es a Rey lo que Mujer es a Reina, también parecen aprender que Hombre es a Médico lo que Mujer es a Enfermera: ¡un sesgo

9. Tomaš Mikolov *et al.*, "Distributed Representations of Words and Phrases and Their Compositionality", actas de la 26.ª *International Conference on Neural Information Processing Systems* 2 (2013): 3111-3119.

19 Entrenar y desplegar modelos de TensorFlow a escala

Cuando ya tienes un bonito modelo que hace predicciones increíbles, ¿qué haces con él? Bueno, ¡hay que llevarlo a producción! Podría ser tan sencillo como ejecutar el modelo con un lote de datos y, tal vez, escribir un *script* que lo ejecute cada noche. Sin embargo, normalmente hay que hacer bastante más. Es posible que varias partes de tu infraestructura necesiten usar este modelo con datos en vivo, en cuyo caso es probable que te interese envolverlo en un servicio web: así, cualquier parte de la infraestructura puede consultar al modelo en cualquier momento con una sencilla API REST (u otro protocolo), como vimos en el capítulo 2. Pero, con el paso del tiempo, necesitas volver a entrenar el modelo regularmente con datos nuevos y llevar la versión actualizada a producción. Debes gestionar las versiones del modelo, pasando con gracia de un modelo al otro, con la posibilidad de volver al anterior si hay problemas, y quizá ejecutando varios modelos distintos en paralelo para realizar experimentos A/B.[1] Si tu producto tiene éxito, tu servicio puede empezar a recibir muchas consultas por segundo (QPS, *Queries Per Second*), y debe escalar para soportar esa carga. Una solución estupenda para escalar el servicio, como veremos en este capítulo, es usar TF Serving en nuestra estructura de hardware o en un servicio en la nube como AI Platform de Google Cloud. Se ocupará de servir tu modelo con eficiencia, gestionar bien las transiciones entre modelos, etc. Si usas la plataforma en la nube, también obtendrás muchas funciones adicionales, como herramientas de monitorización potentes.

Además, si tienes muchos datos de entrenamiento y modelos que requieren mucha computación, el tiempo de entrenamiento puede ser prohibitivamente largo. Si tu producto necesita adaptarse a cambios con rapidez, un entrenamiento largo puede ser un problema crítico (por

1. Un experimento A/B consiste en probar dos versiones diferentes de tu producto en distintos subconjuntos de usuarios para comprobar qué versión funciona mejor y ampliar conocimientos.

Ya tenemos todo lo necesario para avanzar en el grafo hasta el nodo de multiplicación en la función g. El cálculo nos dice que la derivada del producto de dos funciones u y v es $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$. Por tanto, podemos construir una gran parte del grafo de la derecha, que representa $0 \times x + y \times 1$.

Por último, podemos llegar al nodo de suma de la función g. Como hemos mencionado, la derivada de una suma de funciones es la suma de las derivadas de esas funciones. Por tanto, necesitamos crear un nodo de suma y conectarlo a las partes del grafo que ya hemos calculado. Obtenemos la derivada parcial correcta: $\partial g/\partial x = 0 + (0 \times x + y \times 1)$.

Sin embargo, esta ecuación puede simplificarse (mucho). Pueden aplicarse pasos de podar al grafo de computación para eliminar todas las operaciones innecesarias y obtenemos un grafo mucho más pequeño con solo un nodo: $\partial g/\partial x = y$. En este caso, la simplificación es bastante sencilla, pero, para una función más compleja, la diferenciación automática hacia delante puede producir un grafo enorme que puede ser difícil de simplificar y su rendimiento puede no ser óptimo.

Ten en cuenta que hemos empezado con un grafo de computación y la diferenciación automática hacia delante ha producido otro grafo de computación. Esto se denomina "diferenciación simbólica" y tiene dos características interesantes: en primer lugar, una vez que se ha producido el grafo de computación de la derivada, podemos utilizarlo tantas veces como queramos para calcular las derivadas de la función dada para cualquier valor de x e y; en segundo lugar, podemos ejecutar otra vez la diferenciación automática hacia delante en el grafo resultante para obtener derivadas de segundo orden si alguna vez es necesario (es decir, derivadas de derivadas). Podríamos incluso obtener derivadas de tercer orden, y así sucesivamente.

Pero también es posible ejecutar una diferenciación automática hacia delante sin construir un grafo (es decir, de manera numérica, no simbólica), con solo calcular resultados intermedios sobre la marcha. Un modo de hacerlo es utilizar números duales, que son números extraños, pero fascinantes, con la forma $a + b\varepsilon$, donde a y b son números reales y ε es un número infinitesimal tal que $\varepsilon^2 = 0$ (pero $\varepsilon \neq 0$). Puedes imaginar el número dual $42 + 24\varepsilon$ como algo similar a 42,0000···000024 con un número infinito de ceros (pero, por supuesto, está simplificado solo para que te hagas una idea de lo que son los números duales). Un número dual se representa en la memoria como un par de flotantes. Por ejemplo, $42 + 24\varepsilon$ se representa mediante el par (42,0, 24,0).

Los números duales pueden sumarse, multiplicarse, etc., como muestra la ecuación D.3.

Ecuación D.3. Algunas operaciones con números duales.

$$\lambda(a + b\varepsilon) = \lambda a + \lambda b\varepsilon$$
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$
$$(a + b\varepsilon) \times (c + d\varepsilon) = ac + (ad + bc)\varepsilon + (bd)\varepsilon^2 = ac + (ad + bc)\varepsilon$$

Lo que es más importante: puede demostrarse que $h(a + b\varepsilon) = h(a) + b \times h'(a)\varepsilon$, así que calcular $h(a + \varepsilon)$ te da $h(a)$ y la derivada $h'(a)$ de un solo golpe. La figura D.2 muestra que la derivada parcial de $f(x, y)$ con respecto a x en $x = 3$ e $y = 4$ (que escribiremos como $\partial f/\partial x\ (3, 4)$) puede

calcularse utilizando números duales. Lo único que tenemos que hacer es calcular $f(3 + \varepsilon, 4)$; esto generará como salida un número dual cuyo primer componente es igual a $f(3, 4)$ y cuyo segundo componente es igual a $\partial f/\partial x\ (3, 4)$.

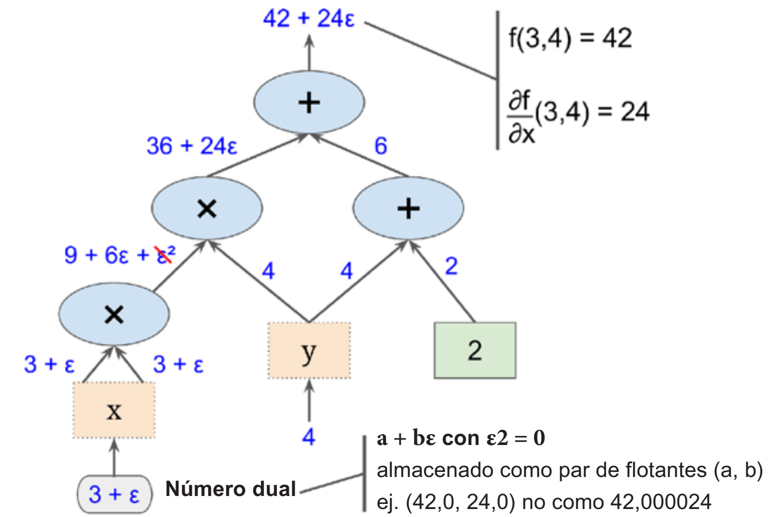

Figura D.2. Diferenciación automática hacia delante usando números duales.

Para calcular $\partial f/\partial x\ (3, 4)$ tendríamos que volver a pasar por el grafo, pero esta vez con $x = 3$ e $y = 4 + \varepsilon$.

Por tanto, la diferenciación automática hacia delante es mucho más exacta que la aproximación mediante diferencias finitas, pero tiene el mismo fallo principal, al menos cuando hay muchas entradas y pocas salidas (como ocurre cuando se trabaja con redes neuronales): si hubiese 1.000 parámetros, habría que pasar 1.000 veces por el grafo para calcular todas las derivadas parciales. Aquí es donde brilla la diferenciación automática inversa: puede calcularlas en solo dos pasos por el gráfico. Veamos cómo.

Diferenciación automática inversa

La diferenciación automática inversa es la solución implementada por TensorFlow. Primero, pasa por el grafo hacia delante (es decir, desde las entradas hacia la salida) para calcular el valor de cada nodo. Después, realiza un segundo paso, esta vez en la dirección opuesta (es decir, desde la salida hacia las entradas) para calcular todas las derivadas parciales. El nombre "inversa" viene de este segundo paso a través del grafo, donde los gradientes fluyen en la dirección inversa. La figura D.3 representa el segundo paso. Durante el primer paso, se han calculado todos los valores de los nodos, empezando desde $x = 3$ e $y = 4$. Puedes ver esos valores en la parte inferior derecha de cada nodo (por ejemplo, $x \times x = 9$). Los nodos se etiquetan de n_1 a n_7 para que haya mayor claridad. El nodo de salida es n_7: $f(3, 4) = n_7 = 42$.

Aprende Machine Learning con Scikit-Learn, Keras y TensorFlow

Gracias a varios logros innovadores, el *deep learning* ha dado un gran impulso a todo el campo del *machine learning*. Ahora, incluso programadores que no saben casi nada de esta tecnología pueden usar herramientas sencillas y eficaces para implementar programas capaces de aprender a partir de datos. La edición actualizada de este *best seller* utiliza ejemplos concretos, una teoría mínima y *frameworks* de Python listos para la producción para ayudarte a obtener una comprensión intuitiva de los conceptos y herramientas para crear sistemas inteligentes.

Aprenderás varias técnicas que podrás usar enseguida. Con ejercicios en cada capítulo para ayudarte a aplicar lo que has aprendido, lo único que necesitas para empezar es experiencia en programación. Todo el código está disponible en GitHub. Se ha actualizado a TensorFlow 2 y la versión más reciente de Scikit-Learn.

- Aprende los fundamentos del *machine learning* mediante un proyecto de principio a fin utilizando Scikit-Learn y pandas.
- Crea y entrena muchas arquitecturas de redes neuronales de clasificación y regresión utilizando TensorFlow 2.
- Descubre la detección de objetos, la segmentación semántica, los mecanismos de atención, los modelos de lenguaje, las GAN y mucho más.
- Explora la API de Keras, la API oficial de alto nivel para TensorFlow 2.
- Prepara modelos de TensorFlow para producción utilizando la API Data, API de estrategias de distribución, TF Transform y TF Serving en TensorFlow.
- Despliega en AI Platform de Google Cloud o en dispositivos móviles.
- Aprovecha técnicas de aprendizaje no supervisado, como la reducción de dimensionalidad, el agrupamiento y la detección de anomalías.
- Crea agentes de aprendizaje autónomos con aprendizaje por refuerzo, incluido el uso de la biblioteca TF-Agents.

"Un recurso excepcional para estudiar *machine learning*. Encontrará explicaciones lúcidas e intuitivas y un montón de trucos prácticos".

—**François Chollet**
creador de Keras,
autor de *Deep learning con Python*.

"Este libro es una gran introducción a la teoría y la práctica de la resolución de problemas con redes neuronales; se lo recomiendo a cualquiera que quiera aprender sobre *machine learning* práctico".

—**Pete Warden**
Mobile Lead for TensorFlow

Aurélien Géron es formador y asesor de *machine learning*. Antiguo *googler*, dirigió el equipo de clasificación de vídeos de YouTube desde 2013 a 2016. También fue fundador y CTO de Wifirst (proveedor de servicios de Internet inalámbrico líder en Francia) desde 2002 a 2012.

www.anayamultimedia.es

ISBN 978-84-415-4264-8