

O'REILLY®

Wydanie II
Aktualizacja
do modułu TensorFlow 2

Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow

powered by



Helion

Aurélien Géron

Tytuł oryginału: Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow:
Concepts, Tools, and Techniques to Build Intelligent Systems, 2nd Edition

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-6003-7

© 2020 Helion SA

Authorized Polish translation of the English edition of *Hands-On Machine Learning with Scikit-Learn, Keras and TensorFlow*, 2E ISBN 9781492032649 © 2019 Kiwisoft S.A.S.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicielami.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/uczem2_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	15
<hr/>	
Część I. Podstawy uczenia maszynowego	25
<hr/>	
1. Krajobraz uczenia maszynowego	27
Czym jest uczenie maszynowe?	28
Dlaczego warto korzystać z uczenia maszynowego?	28
Przykładowe zastosowania	31
Rodzaje systemów uczenia maszynowego	33
Uczenie nadzorowane i uczenie nienadzorowane	34
Uczenie wsadowe i uczenie przyrostowe	40
Uczenie z przykładów i uczenie z modelu	43
Główne problemy uczenia maszynowego	48
Niedobór danych uczących	50
Niereprezentatywne dane uczące	50
Dane kiepskiej jakości	51
Nieistotne cechy	52
Przetrenowanie danych uczących	52
Niedotrenowanie danych uczących	54
Podsumowanie	54
Testowanie i ocenianie	55
Strojenie hiperparametrów i dobór modelu	55
Niezgodność danych	56
Ćwiczenia	57

2. Nasz pierwszy projekt uczenia maszynowego	59
Praca z rzeczywistymi danymi	59
Przeanalizuj całokształt projektu	61
Określ zakres problemu	61
Wybierz metrykę wydajności	63
Sprawdź założenia	65
Zdobądź dane	65
Stwórz przestrzeń roboczą	66
Pobierz dane	68
Rzut oka na strukturę danych	70
Stwórz zbiór testowy	74
Odkrywaj i wizualizuj dane, aby zdobywać nowe informacje	78
Wizualizowanie danych geograficznych	78
Poszukiwanie korelacji	80
Eksperymentowanie z kombinacjami atrybutów	83
Przygotuj dane pod algorytmy uczenia maszynowego	84
Oczyszczanie danych	84
Obsługa tekstu i atrybutów kategorialnych	87
Niestandardowe transformatory	89
Skalowanie cech	90
Potoki transformujące	90
Wybór i uczenie modelu	92
Trenowanie i ocena modelu za pomocą zbioru uczącego	92
Dokładniejsze ocenianie za pomocą sprawdzianu krzyżowego	93
Wyreguluj swój model	96
Metoda przeszukiwania siatki	96
Metoda losowego przeszukiwania	98
Metody zespołowe	98
Analizuj najlepsze modele i ich błędy	98
Oceń system za pomocą zbioru testowego	99
Uruchom, monitoruj i utrzymuj swój system	100
Teraz Twoja kolej!	103
Ćwiczenia	103
3. Klasyfikacja	105
Zbiór danych MNIST	105
Uczenie klasyfikatora binarnego	107
Miary wydajności	108
Pomiar dokładności za pomocą sprawdzianu krzyżowego	108
Macierz pomyłek	110
Precyzyja i pełność	111

Kompromis pomiędzy precyzją a pełnością	112
Wykres krzywej ROC	116
Klasyfikacja wieloklasowa	119
Analiza błędów	121
Klasyfikacja wieloetykietowa	124
Klasyfikacja wielowyjsciowa	125
Ćwiczenia	127
4. Uczenie modeli	129
Regresja liniowa	130
Równanie normalne	131
Złożoność obliczeniowa	134
Gradient prosty	135
Wsadowy gradient prosty	138
Stochastyczny spadek wzduż gradientu	141
Schodzenie po gradiencie z minigrupami	143
Regresja wielomianowa	145
Krzywe uczenia	146
Regularyzowane modele liniowe	150
Regresja grzbietowa	150
Regresja metodą LASSO	153
Metoda elastycznej siatki	155
Wczesne zatrzymywanie	156
Regresja logistyczna	157
Szacowanie prawdopodobieństwa	158
Funkcje ucząca i kosztu	159
Granice decyzyjne	160
Regresja softmax	162
Ćwiczenia	166
5. Maszyny wektorów nośnych	167
Liniowa klasyfikacja SVM	167
Klasyfikacja miękkiego marginesu	168
Nieliniowa klasyfikacja SVM	170
Jądro wielomianowe	171
Cechy podobieństwa	172
Gaussowskie jądro RBF	173
Złożoność obliczeniowa	175
Regresja SVM	175
Mechanizm działania	177
Funkcja decyzyjna i prognozy	177
Cel uczenia	178

Programowanie kwadratowe	180
Problem dualny	181
Kernelizowane maszyny SVM	182
Przyrostowe maszyny SVM	185
Ćwiczenia	186
6. Drzewa decyzyjne	187
Uczenie i wizualizowanie drzewa decyzyjnego	187
Wyliczanie prognoz	188
Szacowanie prawdopodobieństw przynależności do klas	190
Algorytm uczący CART	191
Złożoność obliczeniowa	192
Wskaźnik Giniego czy entropia?	192
Hiperparametry regularyzacyjne	193
Regresja	194
Niestabilność	196
Ćwiczenia	197
7. Uczenie zespołowe i losowe lasy	199
Klasyfikatory głosujące	199
Agregacja i wklejanie	202
Agregacja i wklejanie w module Scikit-Learn	203
Ocena OOB	205
Rejony losowe i podprzestrzeń losowe	206
Losowe lasy	206
Zespół Extra-Trees	207
Istotność cech	207
Wzmacnianie	209
AdaBoost	209
Wzmacnianie gradientowe	212
Kontaminacja	217
Ćwiczenia	219
8. Redukcja wymiarowości	223
Klątwa wymiarowości	224
Główne strategie redukcji wymiarowości	225
Rzutowanie	225
Uczenie rozmaitościowe	227
Analiza PCA	228
Zachowanie wariancji	229
Główne składowe	230

Rzutowanie na d wymiarów	231
Implementacja w module Scikit-Learn	232
Współczynnik wariancji wyjaśnionej	232
Wybór właściwej liczby wymiarów	232
Algorytm PCA w zastosowaniach kompresji	233
Losowa analiza PCA	234
Przyrostowa analiza PCA	235
Jądrowa analiza PCA	236
Wybór jądra i strojenie hiperparametrów	236
Algorytm LLE	239
Inne techniki redukowania wymiarowości	241
Ćwiczenia	241
9. Techniki uczenia nienadzorowanego	243
Analiza skupień	244
Algorytm centroidów	246
Granice algorytmu centroidów	255
Analiza skupień w segmentacji obrazu	256
Analiza skupień w przetwarzaniu wstępny	257
Analiza skupień w uczeniu półnadzorowanym	259
Algorytm DBSCAN	262
Inne algorytmy analizy skupień	265
Mieszaniny gaussowskie	266
Wykrywanie anomalii za pomocą mieszanin gaussowskich	271
Wyznaczanie liczby skupień	273
Modele bayesowskie mieszanin gaussowskich	275
Inne algorytmy służące do wykrywania anomalii i nowości	279
Ćwiczenia	280
Część II. Sieci neuronowe i uczenie głębokie	283
10. Wprowadzenie do sztucznych sieci neuronowych i ich implementacji z użyciem interfejsu Keras	285
Od biologicznych do sztucznych neuronów	286
Neurony biologiczne	287
Operacje logiczne przy użyciu neuronów	288
Perceptron	289
Perceptron wielowarstwowy i propagacja wsteczna	293
Regresyjne perceptrony wielowarstwowe	297
Klasyfikacyjne perceptrony wielowarstwowe	298

Implementowanie perceptronów wielowarstwowych za pomocą interfejsu Keras	300
Instalacja modułu TensorFlow 2	301
Tworzenie klasyfikatora obrazów za pomocą interfejsu sekwencyjnego	302
Tworzenie regresyjnego perceptronu wielowarstwowego za pomocą interfejsu sekwencyjnego	311
Tworzenie złożonych modeli za pomocą interfejsu funkcyjnego	312
Tworzenie modeli dynamicznych za pomocą interfejsu podklasowego	316
Zapisywanie i odczytywanie modelu	318
Stosowanie wywołań zwrotnych	318
Wizualizacja danych za pomocą narzędzia TensorBoard	320
Dostrajanie hiperparametrów sieci neuronowej	323
Liczba warstw ukrytych	326
Liczba neuronów w poszczególnych warstwach ukrytych	327
Współczynnik uczenia, rozmiar grupy i pozostałe hiperparametry	328
Ćwiczenia	330
11. Uczenie głębokich sieci neuronowych	333
Problemy zanikających/eksplodujących gradientów	334
Inicjalizacje wag Glorota i He	334
Nienasycające funkcje aktywacji	336
Normalizacja wsadowa	340
Obcinanie gradientu	346
Wielokrotne stosowanie gotowych warstw	347
Uczenie transferowe w interfejsie Keras	348
Nienadzorowane uczenie wstępne	350
Uczenie wstępne za pomocą dodatkowego zadania	350
Szybsze optymalizatory	352
Optymalizacja momentum	352
Przyspieszony spadek wzduż gradientu (algorytm Nesterova)	353
AdaGrad	355
RMSProp	356
Optymalizatory Adam i Nadam	357
Harmonogramowanie współczynnika uczenia	359
Regularyzacja jako sposób zapobiegania przetrenowaniu	364
Regularyzacja ℓ_1 i ℓ_2	364
Porzucanie	365
Regularyzacja typu Monte Carlo (MC)	368
Regularyzacja typu max-norm	370
Podsumowanie i praktyczne wskazówki	371
Ćwiczenia	372

12. Modele niestandardowe i uczenie za pomocą modułu TensorFlow	375
Krótkie omówienie modułu TensorFlow	375
Korzystanie z modułu TensorFlow jak z biblioteki NumPy	379
Tensory i operacje	379
Tensory a biblioteka NumPy	381
Konwersje typów	381
Zmienne	381
Inne struktury danych	382
Dostosowywanie modeli i algorytmów uczenia	383
Niestandardowe funkcje straty	383
Zapisywanie i wczytywanie modeli zawierających elementy niestandardowe	384
Niestandardowe funkcje aktywacji, inicjalizatory, regularizatory i ograniczenia	386
Niestandardowe wskaźniki	387
Niestandardowe warstwy	389
Niestandardowe modele	392
Funkcje straty i wskaźniki oparte na elementach wewnętrznych modelu	394
Obliczanie gradientów za pomocą różniczkowania automatycznego	396
Niestandardowe pętle uczenia	399
Funkcje i grafy modułu TensorFlow	402
AutoGraph i kreślenie	404
Reguły związane z funkcją TF	405
Ćwiczenia	406
13. Wczytywanie i wstępne przetwarzanie danych za pomocą modułu TensorFlow	409
Interfejs danych	410
Łączanie przekształceń	410
Tasowanie danych	412
Wstępne przetwarzanie danych	415
Składanie wszystkiego w całość	416
Pobieranie wstępne	417
Stosowanie zestawu danych z interfejsem tf.keras	418
Format TFRecord	419
Skompresowane pliki TFRecord	420
Wprowadzenie do buforów protokołów	420
Bufory protokołów w module TensorFlow	422
Wczytywanie i analizowanie składni obiektów Example	423
Obsługa list za pomocą bufora protokołów SequenceExample	424
Wstępne przetwarzanie cech wejściowych	425
Kodowanie cech kategoryalnych za pomocą wektorów gorącojedynkowych	426
Kodowanie cech kategoryalnych za pomocą wektorów właściwościowych	428
Warstwy przetwarzania wstępnego w interfejsie Keras	431

TF Transform	433
Projekt TensorFlow Datasets (TFDS)	435
Ćwiczenia	436
14. Głębokie widzenie komputerowe za pomocą splotowych sieci neuronowych	439
Struktura kory wzrokowej	440
Warstwy splotowe	441
Filtry	443
Stosy map cech	444
Implementacja w module TensorFlow	446
Zużycie pamięci operacyjnej	448
Warstwa łącząca	449
Implementacja w module TensorFlow	451
Architektury splotowych sieci neuronowych	452
LeNet-5	454
AlexNet	455
GoogLeNet	458
VGGNet	461
ResNet	461
Xception	465
SENet	466
Implementacja sieci ResNet-34 za pomocą interfejsu Keras	468
Korzystanie z gotowych modeli w interfejsie Keras	469
Gotowe modele w uczeniu transferowym	471
Klasyfikowanie i lokalizowanie	473
Wykrywanie obiektów	474
W pełni połączone sieci splotowe	476
Sieć YOLO	478
Segmentacja semantyczna	481
Ćwiczenia	484
15. Przetwarzanie sekwencji za pomocą sieci rekurencyjnych i splotowych	487
Neurony i warstwy rekurencyjne	488
Komórki pamięci	490
Sekwencje wejść i wyjść	491
Uczenie sieci rekurencyjnych	492
Prognozowanie szeregów czasowych	493
Wskaźniki bazowe	494
Implementacja prostej sieci rekurencyjnej	494
Głębokie sieci rekurencyjne	496
Prognozowanie kilka taktów w przód	497

Obsługa długich sekwencji	500
Zwalczanie problemu niestabilnych gradientów	501
Zwalczanie problemu pamięci krótkotrwałej	503
Ćwiczenia	511
16. Przetwarzanie języka naturalnego za pomocą sieci rekurencyjnych i mechanizmów uwagi	513
Generowanie tekstów szekspirowskich za pomocą znakowej sieci rekurencyjnej	514
Tworzenie zestawu danych uczących	515
Rozdzielanie zestawu danych sekwencyjnych	515
Dzielenie zestawu danych sekwencyjnych na wiele ramek	516
Budowanie i uczenie modelu Char-RNN	518
Korzystanie z modelu Char-RNN	519
Generowanie sztucznego tekstu szekspirowskiego	519
Stanowe sieci rekurencyjne	520
Analiza sentymentów	522
Maskowanie	526
Korzystanie z gotowych reprezentacji właściwościowych	527
Sieć typu koder – dekoder służąca do neuronowego tłumaczenia maszynowego	529
Dwukierunkowe warstwy rekurencyjne	532
Przeszukiwanie wiązkowe	533
Mechanizmy uwagi	534
Mechanizm uwagi wizualnej	537
Liczy się tylko uwaga, czyli architektura transformatora	539
Współczesne innowacje w modelach językowych	546
Ćwiczenia	548
17. Uczenie reprezentacji za pomocą autokoderów i generatywnych sieci przeciwnaturalnych	551
Efektywne reprezentacje danych	552
Analiza PCA za pomocą niedopełnionego autokodera liniowego	554
Autokodery stosowe	555
Implementacja autokodera stosowego za pomocą interfejsu Keras	556
Wizualizowanie rekonstrukcji	557
Wizualizowanie zestawu danych Fashion MNIST	558
Nienadzorowane uczenie wstępne za pomocą autokoderów stosowych	558
Wiązanie wag	560
Uczenie autokoderów pojedynczo	561
Autokodery splotowe	562
Autokodery rekurencyjne	563
Autokodery odszumiające	564
Autokodery rzadkie	566

Autokodery wariacyjne	569
Generowanie obrazów Fashion MNIST	572
Generatywne sieci przeciwnostawne	574
Problemy związane z uczeniem sieci GAN	577
Głębokie splotowe sieci GAN	579
Rozrost progresywny sieci GAN	582
Sieci StyleGAN	585
Ćwiczenia	587
18. Uczenie przez wzmacnianie	589
Uczenie się optymalizowania nagród	590
Wyszukiwanie strategii	591
Wprowadzenie do narzędzia OpenAI Gym	593
Sieci neuronowe jako strategie	597
Ocenianie czynności: problem przypisania zasługi	598
Gradienty strategii	600
Procesy decyzyjne Markowa	604
Uczenie metodą różnic czasowych	607
Q-uczenie	609
Strategie poszukiwania	610
Przybliżający algorytm Q-uczenia i Q-uczenie głębokie	611
Implementacja modelu Q-uczenia głębokiego	612
Odmiany Q-uczenia głębokiego	616
Ustalone Q-wartości docelowe	616
Podwójna sieć DQN	617
Odtwarzanie priorytetowych doświadczeń	618
Walcząca sieć DQN	618
Biblioteka TF-Agents	619
Instalacja biblioteki TF-Agents	620
Środowiska TF-Agents	620
Specyfikacja środowiska	621
Funkcje opakowujące środowisko i wstępne przetwarzanie środowiska Atari	622
Architektura ucząca	625
Tworzenie Q-sieci głębokiej	627
Tworzenie agenta DQN	629
Tworzenie bufora odtwarzania i związanego z nim obserwatora	630
Tworzenie wskaźników procesu uczenia	631
Tworzenie sterownika	632
Tworzenie zestawu danych	633
Tworzenie pętli uczenia	636
Przegląd popularnych algorytmów RN	637
Ćwiczenia	639

19. Wielkoskalowe uczenie i wdrażanie modeli TensorFlow	641
Eksploatacja modelu TensorFlow	642
Korzystanie z systemu TensorFlow Serving	642
Tworzenie usługi predykcyjnej na platformie GCP AI	650
Korzystanie z usługi prognozowania	655
Wdrażanie modelu na urządzeniu mobilnym lub wbudowanym	658
Przyspieszanie obliczeń za pomocą procesorów graficznych	661
Zakup własnej karty graficznej	662
Korzystanie z maszyny wirtualnej wyposażonej w procesor graficzny	664
Colaboratory	665
Zarządzanie pamięcią operacyjną karty graficznej	666
Umieszczanie operacji i zmiennych na urządzeniach	669
Przetwarzanie równolegle na wielu urządzeniach	671
Uczenie modeli za pomocą wielu urządzeń	673
Zrównoleglanie modelu	673
Zrównoleglanie danych	675
Uczenie wielkoskalowe za pomocą interfejsu strategii rozpraszania	680
Uczenie modelu za pomocą klastra TensorFlow	681
Realizowanie dużych grup zadań uczenia	
za pomocą usługi Google Cloud AI Platform	684
Penetracyjne strojenie hiperparametrów w usłudze AI Platform	686
Ćwiczenia	688
Dziękuję!	688
A Rozwiązywanie ćwiczeń	691
B Lista kontrolna projektu uczenia maszynowego	725
C Problem dualny w maszynach wektorów nośnych	731
D Różniczkowanie automatyczne	735
E Inne popularne architektury sieci neuronowych	743
F Specjalne struktury danych	751
G Grafy TensorFlow	757

Przedmowa

Fenomen uczenia maszynowego

W 2006 roku Geoffrey Hinton i in. opublikowali artykuł (<https://www.cs.toronto.edu/~hinton/absps/ncfast.pdf>)¹ objaśniający mechanizm uczenia głębokiej sieci neuronowej zdolnej do rozpoznawania odręcznie zapisywanych znaków z niespotykaną dotąd precyzją (>98%). Technika ta została nazwana **uczeniem głębokim** (ang. *deep learning*). Głęboka sieć neuronowa stanowi uproszczony (bardzo) model ludzkiej kory mózgowej, składający się z poszczególnych warstw sztucznych neuronów. W tamtych czasach panowało powszechnie przekonanie o niemożności uczenia głębokich sieci neuronowych² i większość badaczy porzuciła tę koncepcję pod koniec lat dziewięćdziesiątych. Wspomniany artykuł ponownie wzniecił zainteresowanie w społeczności naukowej i już krótki czas po jego upowszechnieniu pojawiło się wiele nowych publikacji udowadniających, że uczenie głębokie jest nie tylko możliwe, lecz także pozwala na niesamowite osiągnięcia daleko wykraczające poza zakres standardowych technik **uczenia maszynowego** (ang. *machine learning* — ML); oczywiście, nie byłoby to możliwe bez olbrzymiej mocy obliczeniowej i potężnych ilości danych. Entuzjazm ten wkrótce rozprzestrzenił się również na inne dziedziny uczenia maszynowego.

Mniej więcej w ciągu kolejnej dekady uczenie maszynowe zdobiło przemysł: stanowi ono źródło magii cechującej wiele zaawansowanych produktów, odpowiada za wyniki wyszukiwania w przeglądarkach, jest sercem funkcji rozpoznawania mowy w smartfonach i doboru polecanych filmów, a także jest w stanie pokonać mistrza świata w grze Go. Zanim się zorientujesz, będzie również odpowiedzialne za prowadzenie Twojego samochodu.

Uczenie maszynowe w Twoich projektach

Zrozumiałe jest zatem Twoje podekscytowanie uczeniem maszynowym oraz chęć dołączenia do zabawy!

¹ Geoffrey E. Hinton i in., *A Fast Learning Algorithm for Deep Belief Nets*, „Neural Computation” 18 (2006): 1527 – 1554.

² Pomimo faktu, że głębokie sieci neuronowe autorstwa Yanna LeCuna już od początku lat dziewięćdziesiątych dobrze sobie radziły z rozpoznawaniem obrazów; były to jednak wyspecjalizowane sieci.

Być może marzysz o wstawieniu mózgu do własnoręcznie skonstruowanego robota? Może chcesz, aby był w stanie rozpoznawać twarze albo nauczył się chodzić po domu?

A może Twoja firma przechowuje olbrzymie ilości danych (wpisy dzienników zdarzeń, dane finansowe, produkcyjne, pomiary czujników, dane statystyczne, raporty działu HR itd.) i chcesz zwiększyć prawdopodobieństwo wyszukiwania w nich ukrytych skarbów (gdybyś wiedziała/wiedział, gdzie ich szukać). Dzięki uczeniu maszynowemu możesz osiągnąć między innymi następujące rezultaty (https://en.wikipedia.org/wiki/Machine_learning#Applications):

- kategorie klientów oraz określanie najlepszych strategii marketingowych dla każdej z tych grup,
- zalecanie poszczególnym klientom produktów na podstawie danych zebranych od podobnych klientów,
- wykrywanie potencjalnie nielegalnych transakcji,
- przewidywanie przyszłorocznych obrotów.

Bez względu na powód postanowiłaś/postanowiłeś poznać tajemnice uczenia maszynowego i zaimplementować uzyskaną wiedzę w swoich projektach. Doskonalały pomysły!

Cel i sposób jego osiągnięcia

Podczas pisania tej książki założyłem, że masz, Czytelniku, bardzo niewielkie pojęcie o uczeniu maszynowym. Moim celem jest wyjaśnienie pojęć, omówienie zagadnień w przystępny, intuicyjny sposób oraz zapewnienie Ci narzędzi umożliwiających zaimplementowanie programów zdolnych do **uczenia się z dostarczanych danych**.

Przyjrzymy się dużej liczbie technik, począwszy od najprostszych i najpopularniejszych (np. regresja liniowa) aż do niektórych technik uczenia głębokiego pozwalających na regularne wygrywanie branżowych konkursów.

Nie będziemy bawić się w tworzenie własnych wersji każdego algorytmu, lecz skorzystamy z gotowych bibliotek środowiska Python:

- Biblioteka *Scikit-Learn* (<http://scikit-learn.org/stable/>) jest bardzo przystępna, a jednocześnie zawiera wiele wydajnych algorytmów uczenia maszynowego; z tego powodu znakomicie nadaje się dla osób rozpoczynających przygodę z uczeniem maszynowym.
- Biblioteka *TensorFlow* (<https://www.tensorflow.org/>) jest nieco bardziej skomplikowana; służy ona do rozproszonych obliczeń numerycznych. Dzięki niej możliwe jest wydajne uczenie i używanie bardzo dużych sieci neuronowych poprzez rozdzielenie obliczeń pomiędzy setki serwerów zaopatrzonych w wiele procesorów graficznych (GPU). Biblioteka TensorFlow (TF) została stworzona przez firmę Google i znalazła miejsce w jego licznych wielkoskalowych zastosowaniach. W listopadzie 2015 roku została ona objęta licencją otwartego oprogramowania (ang. *open-source*).
- Biblioteka *Keras* (<https://keras.io>) stanowi wysokopoziomowy interfejs API uczenia głębokiego, znacznie upraszczający trenowanie i uruchamianie sieci neuronowych. Może ona działać jako dodatek do biblioteki TensorFlow, Theano lub Microsoft Cognitive Tools (znanej wcześniej jako CNTK). TensorFlow zawiera własną implementację interfejsu Keras o nazwie *tf.keras*, zapewniającą obsługę niektórych zaawansowanych funkcji TF (np. możliwość wydajnego wczytywania danych).

Kładę w tej książce nacisk na praktyczne ćwiczenia, dzięki czemu będziesz w stanie rozwinać intuicyjne postrzeganie pojęć z zakresu uczenia maszynowego za pomocą namacalnych, działających przykładów oraz niewielkiego teoretycznego wsparcia. Możesz zapoznać się z treścią bez korzystania z laptopa, zalecam jednak eksperymentowanie z przykładowym kodem źródłowym dostępnym w postaci notatników Jupyter — <ftp://ftp.helion.pl/przyklady/uczem2.zip>. Oryginalne materiały (aktualizowane od czasu do czasu) dostępne są natomiast na stronie <https://github.com/ageron/handson-ml2>.

Wymogi wstępne

Zakładam tu, że masz już pewne doświadczenie w programowaniu za pomocą języka Python, a także że znasz jego główne biblioteki naukowe, zwłaszcza *NumPy* (<http://www.numpy.org/>), *pandas* (<http://pandas.pydata.org/>) i *Matplotlib* (<http://matplotlib.org/>).

Ponadto osoby pragnące zrozumieć operacje wykonywane przez algorytmy uczenia maszynowego powinny również mieć opanowaną matematykę na poziomie akademickim (analiza matematyczna, algebra liniowa, rachunek prawdopodobieństwa i statystyka).

Jeśli jeszcze nie znasz środowiska Python, warto zapoznać się z samouczkiem Python na stronie <http://learnpython.org/pl/> oraz z oficjalną dokumentacją (<https://docs.python.org/3/tutorial/>).

Osoby niezaznajomione z notatnikami Jupyter znajdą w rozdziale 2. opis procesu instalacji i podstawowe informacje na ich temat; warto zaopatrzyć się w to potężne narzędzie.

Z kolei w dołączonych do tej książki notatnikach Jupyter znajdziesz informacje i poradniki omawiające biblioteki naukowe środowiska Python. Dostępny jest również krótki kurs algebry liniowej.

Zawartość książki

Niniejsza książka składa się z dwóch części. W części I, „Podstawy uczenia maszynowego”, zajmujemy się następującymi zagadnieniami:

- Czym jest uczenie maszynowe? Jakie problemy staramy się rozwiązywać za jego pomocą? Jakie rozróżniamy główne kategorie i fundamentalne pojęcia tworzące jego systemy?
- Główne etapy typowego projektu uczenia maszynowego.
- Uczenie poprzez dopasowywanie modelu do danych.
- Optymalizowanie funkcji kosztu.
- Oczyszczanie przygotowywanie danych oraz zarządzanie nimi.
- Dobór i projektowanie cech.
- Wybór modelu i strojenie hiperparametrów za pomocą sprawdzianu krzyżowego (walidacji krzyżowej, krosvalidacji).
- Wyzwania stojące przed uczeniem maszynowym, zwłaszcza niedotrenowanie i przetrenowanie (kompromis pomiędzy obciążeniem a wariancją).

- Najpowszechniej stosowane algorytmy uczenia maszynowego: regresja liniowa i wielomianowa, regresja logistyczna, metoda k-najbliższych sąsiadów, maszyny wektorów nośnych, drzewa decyzyjne, losowe lasy oraz metody uczenia zespołowego.
- Redukowanie wymiarowości danych uczących w celu zniwelowania „klątwy wymiarowości”.
- Inne techniki uczenia nienadzorowanego, w tym analiza skupień, szacowanie gęstości i wykrywanie anomalii.

Część II, „Sieci neuronowe i uczenie głębokie”, została poświęcona następującym tematom:

- Omówienie sieci neuronowych i ich zastosowań.
- Tworzenie i uczenie sieci neuronowych za pomocą bibliotek TensorFlow i Keras.
- Najważniejsze architektury sieci neuronowych: sieci jednokierunkowe (gęste) do przetwarzania danych tabelarycznych, splotowe wykorzystywane w widzeniu komputerowym, rekurencyjne i długiej pamięci krótkotrwałej (LSTM) do przetwarzania sekwencji, kodery/dekodery i transformatory używane w przetwarzaniu języka naturalnego, a także autokodery i generatywne sieci przeciwwstawne (GAN) stosowane w uczeniu generatywnym.
- Techniki uczenia głębokich sieci neuronowych.
- Mechanizm tworzenia agenta (np. bota w grze) poznającego dobre strategie metodą prób i błędów za pomocą uczenia przez wzmacnianie.
- Skuteczne wczytywanie i wstępne przetwarzanie olbrzymich ilości danych.
- Uczenie i wdrażanie wielkoskalowych modeli TensorFlow.

W pierwszej części będziemy głównie korzystać z biblioteki Scikit-Learn, natomiast w drugiej posłużymy się bibliotekami TensorFlow i Keras.



Nie wskakuj od razu do głębokiej wody: metody uczenia głębokiego z pewnością stanowią jeden z najbardziej fascynujących obszarów uczenia maszynowego, zanim jednak do nich przejdziesz, musisz opanować podstawy. Poza tym większość problemów można całkiem skutecznie rozwiązać za pomocą prostszych technik, takich jak metody losowych czy metody zespołowe (zostały one omówione w części I). Uczenie głębokie sprawdza się najlepiej w rozwiązywaniu złożonych problemów, np. rozpoznawaniu obrazów, mowy czy też przetwarzaniu języka naturalnego — przy założeniu, że dysponujesz odpowiednią ilością danych, mocy obliczeniowej i cierpliwości.

Zmiany wprowadzone w nowym wydaniu

W drugim wydaniu niniejszej książki postawiliśmy sobie sześć głównych celów:

1. Omówienie dodatkowych tematów z zakresu uczenia maszynowego: więcej informacji dotyczących nienadzorowanego uczenia maszynowego (w tym analizy skupień, wykrywanie anomalii, szacowania gęstości i modeli mieszanych), dodatkowe techniki uczenia sieci głębokich (w tym sieci samonormalizujące), dodatkowe metody widzenia maszynowego (w tym takie jak Xception, SENet, wykrywanie obiektów za pomocą systemu YOLO i segmentacja semantyczna za

pomocą sieci R-CNN), obsługa sekwencji za pomocą sieci splotowych (CNN, w tym WaveNet), przetwarzanie języka naturalnego przy użyciu sieci rekurencyjnych (RNN), splotowych i transformatorowych, a także sieci GAN.

2. Opis dodatkowych bibliotek i interfejsów API (Keras, Data API, TF-Agents używane w uczeniu przez wzmacnianie), a także trenowania oraz wdrażania wielkoskalowych modeli TF za pomocą takich narzędzi jak Distribution Strategies API, TF-Serving czy usługa Google Cloud AI Platform. Znajdziesz tu również krótkie omówienie bibliotek TF Transform, TFLite, TF Add-ons/Seq2Seq i TensorFlow.js.
3. Opis najnowszych wyników badań poświęconych uczeniu głębokiemu.
4. Przeniesienie wszystkich rozdziałów poświęconych bibliotece TensorFlow na grunt biblioteki TensorFlow 2, a także stosowanie implementacji interfejsu Keras (*tf.keras*) tam, gdzie to możliwe.
5. Zaktualizowanie listingów zgodnie z najnowszymi wersjami bibliotek Scikit-Learn, NumPy, pandas, Matplotlib i innych.
6. Poprawienie czytelności niektórych sekcji i usunięcie pewnych błędów wychwyconych dzięki znakomitej spostrzegawczości Czytelników.

Niektóre rozdziały zostały dodane, inne zmodyfikowane, a kolejność części z nich została zmieniona. Więcej informacji na temat zmian wprowadzonych w wydaniu drugim znajdziesz pod adresem https://github.com/ageron/handson-ml2/blob/master/changes_in_2nd_edition.md.

Dodatkowe zasoby

Dostępnych jest bardzo wiele znakomitych zasobów ułatwiających zapoznanie się z technikami uczenia maszynowego. Na przykład w serwisie Coursera znajdziemy doskonały kurs autorstwa Andrew Ng (<https://www.coursera.org/learn/machine-learning/>), który wymaga jednak olbrzymiego nakładu czasu (kilku miesięcy/kilkunastu tygodni).

W internecie istnieje również mnóstwo interesujących witryn poświęconych uczeniu maszynowemu, z których warto, oczywiście, wymienić zdumiewający podręcznik użytkownika (http://scikit-learn.org/stable/user_guide.html) opisujący bibliotekę Scikit-Learn. Warto również sprawdzić serwis Dataquest (<https://www.dataquest.io/>), zawierający bardzo przyjemne, interaktywne samouczki, a także blogi wymienione na forum Quora (<https://www.quora.com/What-are-the-best-regularly-updated-machine-learning-blogs-or-resources-available>). Na koniec należy wspomnieć o witrynie Deep Learning (<http://deeplearning.net/>), zawierającej bogatą bibliografię.

Nie wyczerpujemy w ten sposób wszystkich pozycji stanowiących wprowadzenie do świata uczenia maszynowego. Mogę polecić zwłaszcza następujące:

- Joel Grus, *Data science od podstaw. Analiza danych w Pythonie* (Helion). W książce tej zostały opisane podstawy uczenia, a także zawarte są „czyste” implementacje niektórych głównych algorytmów w języku Python.
- Stephen Marsland, *Machine Learning: An Algorithmic Perspective* (Chapman & Hall). Znakomite wprowadzenie do uczenia maszynowego, w którym dokładnie opisano wiele zagadnień; także tutaj kod został napisany od podstaw (ale z użyciem biblioteki NumPy).

- Sebastian Raschka, *Python. Uczenie maszynowe* (Helion). Jest to również doskonałe wprowadzenie do uczenia maszynowego; autor korzysta tu z otwartych bibliotek języka Python (PyLearn2 i Theano).
- François Chollet, *Deep Learning. Praca z językiem Python i biblioteką Keras* (Helion). Jest to bardzo praktyczna książka, która opisuje szeroki wachlarz zagadnień w jasny i zwięzły sposób, jak przystało na twórcę doskonałej biblioteki Keras. Autor przedkłada przykładowy kod ponad aparat matematyczny.
- Andriy Burkov, *The Hundred-Page Machine Learning Book* (Lightning Source). Jest to pozycja bardzo krótka, ale omawiająca zdumiewająco bogaty zakres tematów, w które Czytelnik zostaje wprowadzony w przystępny sposób przy jednoczesnym poświęcaniu uwagi wzorom matematycznym.
- Yaser S. Abu-Mostafa, Malik Magdon-Ismail i Hsuan-Tien Lin, *Learning from Data* (MLBook). W książce tej autorzy kładą nacisk na aspekt teoretyczny uczenia maszynowego; zawiera ona mnóstwo informacji, zwłaszcza dotyczących kompromisu pomiędzy obciążeniem a wariancją (patrz rozdział 4.).
- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach, 3rd Edition* (Pearson). Znajomita (i obszerna) pozycja, poruszająca nieprawdopodobną liczbę zagadnień, w tym również tematykę uczenia maszynowego. Pozwala ona spojrzeć na uczenie maszynowe z innej perspektywy.

Doskonałą metodą nauki jest również zarejestrowanie się na stronie *Kaggle.com*, gdzie poprzez różne konkursy i turnieje możesz rozwijać swoje umiejętności, rozwiązyując rzeczywiste problemy wraz z jednymi z najbardziej profesjonalnych specjalistów w dziedzinie uczenia maszynowego.

Konwencje stosowane w książce

W tej książce napotkasz następujące konwencje typograficzne:

Kursywa

Adresy URL, adresy e-mail, nazwy i rozszerzenia plików.

Pogrubienie

Nowe pojęcia.

Czcionka o stałej szerokości

Listingi aplikacji.

Inna czcionka o stałej szerokości

Oznaczone są w ten sposób pojawiające się w tekście elementy programów, takie jak nazwy zmiennych/funkcji, bazy danych, typy danych, zmienne, instrukcje oraz słowa kluczowe.

Pogrubiona czcionka o stałej szerokości

Polecenia lub inny tekst wprowadzany bezpośrednio przez użytkownika, a także tekst, który powinien zostać zastąpiony wartościami zdefiniowanymi przez użytkownika lub przez określony kontekst.

Jeszcze inną czcionką

Wyniki wyświetlane przez wykonany kod.



Ta ikona reprezentuje wskazówkę lub sugestię.



Ta ikona oznacza ogólną uwagę.



Ta ikona oznacza ostrzeżenie lub przestroगę.

Kod źródłowy

Dostępny jest zbiór materiałów dodatkowych, takich jak kod źródłowy i rozwiązania ćwiczeń, który możesz pobrać na stronie [ftp://ftp.helion.pl/przykłady/uczem2.zip³](ftp://ftp.helion.pl/przykłady/uczem2.zip).

W niektórych przykładach zostają pominięte powtarzalne sekcje lub szczegóły, które są oczywiste lub nieistotne w kontekście uczenia maszynowego. W ten sposób możemy skoncentrować się na ważnych fragmentach kodu, a także oszczędzamy miejsce, co pozwala omówić dokładniej większą liczbę zagadnień. Jeżeli interesują Cię pełne listingi, znajdziesz je w notatnikach Jupyter.

Tam, gdzie kod źródłowy generuje jakieś wyniki, oznaczam go znakami zachęty Pythona (`>>> i ...`) znanymi z powłoki Python, co pozwala wyraźnie odróżnić kod od jego rezultatów. Na przykład ten kod definiuje funkcję `square()`, a następnie oblicza i wyświetla kwadrat liczby 3:

```
>>> def square(x):
...     return x ** 2
...
>>> result = square(3)
>>> result
9
```

Jeśli kod nie generuje rezultatów, nie stosujemy znaków zachęty. Czasami jednak wynik może być podany w formie komentarza, na przykład:

```
def square(x):
    return x ** 2

result = square(3) # Wynik wynosi 9
```

³ Oryginalne materiały są umieszczone pod adresem <https://github.com/ageron/handson-ml2>. Warto zaglądać tam od czasu do czasu, gdyż autor często wprowadza poprawki i uzupełnienia w udostępnionych notatnikach Jupyter — przyp. tłum.

Korzystanie z kodu źródłowego

Niniejsza książka ma na celu pomóc Ci w pracy i wykonywaniu zadań. Generalnie, jeżeli znajduje się tu przykładowy kod, możesz go używać w swoich programach i dokumentach. Nie musisz kontaktować się z nami z prośbą o pozwolenie, chyba że kopujesz znaczną część kodu. Na przykład napisanie programu wykorzystującego kilka różnych fragmentów kodu nie wymaga naszego pozwolenia. Sprzedaż lub wysyłanie przykładów na nośnikach optycznych wymaga uzyskania pozwolenia. Odpowiedź na zadane pytanie poprzez powołanie się na tę książkę i zacytowanie fragmentu kodu nie wymagają pozwolenia. Umieszczenie znacznej ilości kodu pochodzącego z tej książki w dokumentacji Twojego produktu wymaga uzyskania pozwolenia.

Nie wymagamy atrybucji, ale bylibyśmy za nią wdzięczni. Składa się na nią zazwyczaj tytuł książki, autor, wydawca i numer ISBN, na przykład: „*Uczenie maszynowe z użyciem Scikit-Learn, Keras i TensorFlow. Wydanie drugie*, Aurélien Géron, Helion, ISBN 978-83-283-4373-3”.

Jeżeli uważasz, że używasz kodu źródłowego niezgodnie z opisanymi powyżej zasadami, napisz do nas na adres permissions@oreilly.com.

Podziękowania

Nawet w najśmiesznych marzeniach nie spodziewałem się, że pierwsze wydanie niniejszej książki zyska tak szerokie grono Czytelników. Otrzymałem od Was wiele wiadomości, niektóre będące pytaniami, inne w uprzejmy sposób wskazujące błędy, a jeszcze inne będące wyrazami uznania. Nie jestem w stanie wyrazić wdzięczności za tak olbrzymie wsparcie! Bardzo Wam wszystkim dziękuję! Nie bójcie się informować mnie o problemach z plikami w serwisie GitHub (<https://github.com/ageron/handson-ml2/issues>) czy o błędach w listingach, zadać mi pytanie lub zamieścić erratę (<https://www.oreilly.com/catalog/errata.csp?isbn=0636920142874>) w przypadku znalezienia nieprawidłowości w treści. Niektórzy Czytelnicy opowiedzieli również, w jaki sposób książka ta pomogła im dostać nową pracę, zrealizować ich pierwszy projekt lub rozwiązać określony problem, z jakim się borykali. Taka forma kontaktu z Czytelnikami jest dla mnie niezwykle motywująca. Jeżeli uznasz, że książka ta okazała się pomocna, z przyjemnością zapoznam się z Twoją historią, prywatnie (np. poprzez serwis LinkedIn: <https://www.linkedin.com/in/aurelien-geron/>) lub publicznie (np. poprzez wpis na Twitterze lub recenzję w serwisie Amazon: <https://www.amazon.com/Hands-Machine-Learning-Scikit-Learn-TensorFlow/dp/1492032646/> albo opinię na stronie wydawnictwa Helion: <https://helion.pl/ksiazki/uczenie-maszynowe-z-uzykiem-scikit-learn-i-tensorflow-aur-lien-g-ron, uczem2. ↵htm#format/d>).

Jestem również niesamowicie wdzięczny wszystkim zdumiewającym osobom, które znalazły czas w zabieganym życiu, aby z taką troską sprawdzić zawartość tej książki. W szczególności chciałbym podziękować François Cholletowi za przejrzenie wszystkich rozdziałów, w których wykorzystywane są biblioteki Keras i TensorFlow oraz przekazanie mnóstwa wnikliwych opinii. Interfejs Keras stanowi jeden z najważniejszych dodatków w tym wydaniu, dlatego korekta merytoryczna jego twórcy jest bezcenna. Szczerze polecam książkę jego autorstwa pt. *Deep Learning. Praca z językiem Python i biblioteką Keras* (<https://helion.pl/ksiazki/delepy.htm> (Helion)). Specjalne podziękowania kieruję również do Ankura Patela, który przejrzał każdy rozdział nowego wydania książki i podzielił się

znakomitymi opiniami, zwłaszcza w kwestii rozdziału 9., poświęconego technikom uczenia nienadzorowanego. Sam napisał książkę na ten temat: *Hands-On Unsupervised Learning Using Python: How to Build Applied Machine Learning Solutions from Unlabeled Data* (https://helion.pl/ksiazki/hands-on-unsupervised-learning-using-python-how-to-build-applied-machine-learning-solutions-from-un-ankur-a-patel,e_11id.htm#format/e) (O'Reilly). Wielkie dzięki także dla Olzhasa Akpambetova, który przejrzał wszystkie rozdziały z drugiej części książki, przetestował znaczną część kodu i zaproponował mnóstwo świetnych sugestii. Jestem wdzięczny Markowi Daoustowi, Jonowi Krohnowi, Dominicowi Monnowi i Joshowi Pattersonowi za dogłębne zapoznanie się z drugą częścią książki i przekazanie fachowych opinii. Nie pominęły żadnej literki i podzielili się mnóstwem użytecznych informacji.

W trakcie pisania wydania drugiego miałem szczęście uzyskać znaczącą pomoc od członków zespołu TensorFlow, zwłaszcza od Martina Wickego, który niestrudzenie odpowiadał na dziesiątki moich pytań, a opracowanie pozostałych zlecił właściwym osobom, takim jak: Karmel Allison, Paige Bailey, Eugene Brevdo, William Chargin, Daniel „Wolff” Dobson, Nick Felt, Bruce Fontaine, Goldie Gadde, Sandeep Gupta, Priya Gupta, Kevin Haas, Konstantinos Katsiapis, Viacheslav Kovalevskyi, Allen Lavoie, Clemens Mewald, Dan Moldovan, Sean Morgan, Tom O’Malley, Alexandre Passos, Andre Susano Pinto, Anthony Platanios, Oscar Ramirez, Anna Revinskaya, Saurabh Saxena, Ryan Sepassi, Jiri Simsa, Xiaodan Song, Christina Sorokin, Dustin Tran, Todd Wang, Pete Warden (który był korektorem również wydania pierwszego), Edd Wilder-James i Yuefeng Zhou. Każda z tych osób miała olbrzymi wkład w tę książkę. Bardzo dziękuję zarówno Wam, jak i pozostałym członkom zespołu TensorFlow, nie tylko za pomoc, lecz również za stworzenie tak wspaniałej biblioteki! Specjalne podziękowania kieruję do Irene Giannoumis i Roberta Crowe'a z zespołu TFX za dogłębne przyjrzenie się rozdziałom 13. i 19.

Na osobne podziękowania zasługuję fantastyczny zespół redakcyjny z wydawnictwa O'Reilly, zwłaszcza Nicole Taché, osoba zawsze uśmiechnięta, pomocna i radosna, która podzieliła się bardzo ważnymi uwagami — nie mógłbym wymarzyć sobie lepszej redaktorki. Jestem bardzo wdzięczny Michelle Cronin, gdyż okazała się bardzo pomocna (i cierpliwa) na początku pracy nad wydaniem drugim, a także Kristen Brown, redaktor prowadzącej wydania drugiego, która sprawowała pieczę nad wszystkimi etapami (zajmowała się też koordynowaniem poprawek i aktualizacji we wszystkich wznowieniach wydania pierwszego). Dziękuję także Rachel Monaghan i Amandzie Kersey odpowiedzialnym za adiustację (odpowiednio w pierwszym i drugim wydaniu) oraz Johnny'emu O'Toole'owi, który odpowiadał za relacje z firmą Amazon i dostarczył wiele odpowiedzi na moje pytania. Równie ważni są dla mnie Marie Beaugureau, Ben Lorica, Mike Loukides i Laurel Ruma ze względu na ich niezachwianą wiarę w ten projekt oraz pomoc w zdefiniowaniu jego zakresu. Dziękuję Mattowi Hackerowi i całemu zespołowi Atlas za wyjaśnienie wszystkich wątpliwości natury technicznej dotyczących formatowania, edytora tekstu asciidoc i aplikacji LaTeX. Ponadto jestem niezmiernie wdzięczny Nickowi Adamsowi, Rebecce Demarest, Rachel Head, Judith McConville, Helen Monroe, Karen Montgomery, Rachel Roumeliotis i całemu wydawnictwu O'Reilly za wkład w powstanie tej książki.

Chciałbym podziękować koleżankom i kolegom z firmy Google, w szczególności zespołowi zajmującemu się klasyfikowaniem filmów umieszczanych w serwisie YouTube, za podzielenie się swoją wiedzą na temat uczenia maszynowego. Bez ich pomocy nigdy nie rozpoczęłbym pisania tej książki.

Na specjalne podziękowania zasługuję moi osobiści „nauczyciele”: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn i Rich Washington. Równie mocno dziękuję wszystkim osobom, z którymi miałem przyjemność pracować w firmie YouTube, a także znakomitym zespołom badawczym firmy Google w Mountain View. Jestem bardzo wdzięczny Martinowi Andrewsowi, Samowi Witteveenowi i Jasonowi Zamanowi za wprowadzenie mnie do grupy Google Developer Experts w Singapurze, z uprzejmym wsparciem Soonsona Kwona, a także za znakomite dyskusje poświęcone uczeniu głębokiemu i bibliotece TensorFlow. Każda osoba mieszkająca w Singapurze i interesująca się uczeniem głębokim powinna udać się na spotkanie Deep Learning Singapore (<https://www.meetup.com/pl-PL/TensorFlow-and-Deep-Learning-Singapore/>). Szczególne podziękowania należą się Jasonowi za podzielenie się wiedzą na temat struktury TFLite omówionej w rozdziale 19.

Nigdy nie zapomnę o osobach recenzujących pierwsze wydanie niniejszej książki, do których należą: David Andrzejewski, Lukas Biewald, Justin Francis, Vincent Guilbeau, Eddy Hung, Karim Matrah, Grégoire Mesnil, Salim Sémaoune, Iain Smears, Michel Tessier, Ingrid von Glehn, Pete Warden oraz oczywiście mój brat Sylvain. Specjalne podziękowania dla Haesuna Parka, który przekazał mi mnóstwo przydatnych uwag i wychwycił kilka błędów podczas tłumaczenia pierwszego wydania na język koreański. Przełożył on także notatniki Jupyter, nie wspominając o dokumentacji biblioteki TensorFlow. Nie mówię po koreańsku, ale wnioskując po jakości uwag, wszystkie jego przekłady muszą być zaiste znakomite! Haesun również miał swój udział w rozwiązaniach niektórych ćwiczeń umieszczonych w wydaniu drugim.

Na koniec chciałbym przekazać wyrazy bezgranicznej wdzięczności mojej ukochanej żonie Emmanuelle i trójce naszych wspaniałych pociech, Alexandre'owi, Rémi'emu i Gabrielle, za motywowanie mnie do pracy nad tą książką. Jestem im również wdzięczny za nienasyconą ciekawosć: wyjaśnienie żonie i dzieciom najbardziej skomplikowanych pojęć zawartych w tej książce pomogło mi uporządkować myśli i bezpośrednio poprawiło jakość wielu jej fragmentów. A oni nadal przynoszą mi kawę i ciasteczka! Czy można marzyć o czymś więcej?

Podstawy uczenia maszynowego

Krajobraz uczenia maszynowego

Większość osób, słysząc o uczeniu maszynowym, wyobraża sobie robota: jedni myślą o posłusznym kamerdynerze, inni o zabójczym terminatorze. Zjawisko to jednak nie stanowi futurystycznej fantazji, lecz jest już elementem współczesnego świata. W rzeczywistości istnieje już od dziesiątek lat w pewnych wyspecjalizowanych zastosowaniach, takich jak techniki **optycznego rozpoznawania znaków** (ang. *Optical Character Recognition — OCR*). Jednak techniki uczenia maszynowego trafiły pod strzechy, poprawiając komfort życia milionów osób, dopiero w latach dziewięćdziesiątych — w postaci **filtrów spamu**. Niekoniecznie przypominają one Skynet¹, ale pod względem technicznym są klasyfikowane jako aplikacje wykorzystujące uczenie maszynowe (istotnie, filtry te uczą się tak skutecznie, że bardzo rzadko musisz własnoręcznie oznaczać wiadomości jako spam). W następnych latach nastąpił wysyp różnych innych zastosowań uczenia maszynowego, stanowiących trzon wielu aplikacji i usług, z których korzystamy na co dzień, począwszy od polecanych produktów aż do wyszukiwania głosowego.

Gdzie się zaczyna i kończy uczenie maszynowe? Co to właściwie oznacza, że dana maszyna **uczy się** określonego zagadnienia? Czy jeśli pobiorę na dysk artykuł z Wikipedii, to mój komputer „nauczył się” jego treści? Czy stał się nagle mądrzejszy? W tym rozdziale wyjaśnię, co uznajemy za uczenie maszynowe oraz dlaczego ta dziedzina może Cię zainteresować.

Następnie, zanim wyruszmy w wyprawę w głąb kontynentu Uczenie Maszynowe, zerkniemy na mapę i przyjrzymy się jego najważniejszym regionom i punktom orientacyjnym: porównamy uczenie nadzorowane z nienadzorowanym, przyrostowe z wsadowym, a także metody uczenia z przykładów z metodami uczenia z modelu. Potem przeanalizujemy cykl tworzenia typowego projektu uczenia maszynowego, zastanowimy się nad głównymi wyzwaniami, jakim należy stawić czoło, oraz przeanalizujemy sposoby oceny i strojenia szybkości modelu.

W niniejszym rozdziale wprowadzam wiele podstawowych pojęć (i terminów), które musi znać każdy szanujący się analityk danych. Informacje tu zawarte są bardzo ogólne (jest to jedyny rozdział pozbaowany listingów kodu) i raczej nieskomplikowane, zanim jednak przejdziesz do następnego rozdziału, upewnij się, że doskonale je rozumiesz. Zatem kawa w dłoń i do dzieła!

¹ Samoświadoma sztuczna inteligencja dążąca do zniszczenia ludzkości w serii filmów *Terminator — przyp. tłum.*



Jeśli znasz już podstawy uczenia maszynowego, możesz przejść od razu do rozdziału 2. Jeżeli jednak nie czujesz się pewnie, spróbuj najpierw odpowiedzieć na wszystkie pytania umieszczone na końcu rozdziału.

Czym jest uczenie maszynowe?

Uczeniem maszynowym nazywamy dziedzinę nauki (i sztukę) programowania komputerów w sposób umożliwiający im **uczenie się z danymi**.

Oto nieco ogólniejsza definicja:

[Uczenie maszynowe to] dziedzina nauki dającą komputerom możliwość uczenia się bez konieczności ich jawnego programowania.

— Arthur Samuel, 1959

A tu bardziej techniczna:

Mówimy, że program komputerowy uczy się na podstawie doświadczenia E w odniesieniu do jakiegoś zadania T i pewnej miary wydajności P, jeśli jego wydajność (mierzona przez P) wobec zadania T wzrasta wraz z nabywaniem doświadczenia E.

— Tom Mitchell, 1997

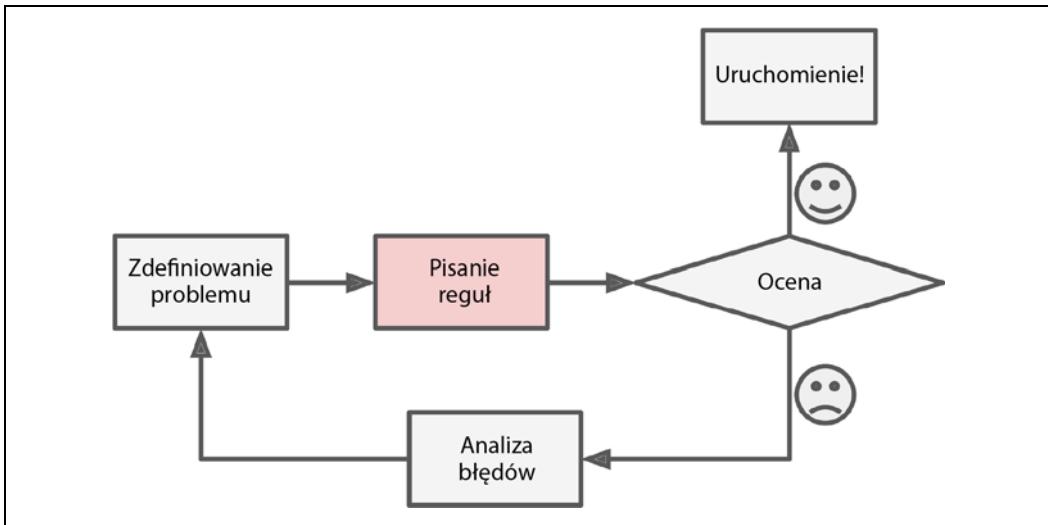
Filtr spamu jest programem wykorzystującym algorytmy uczenia maszynowego do rozpoznawania spamu na podstawie przykładowych wiadomości e-mail (tzn. spamu oznaczonego przez użytkowników oraz zwykłych wiadomości niebędących spamem, określanych mianem „ham”²). Przykładowe dane używane do trenowania systemu noszą nazwę **zbioru/zestawu uczącego** (ang. *training set*). Każdy taki element uczący jest nazywany **przykładem uczącym (próbką uczącą)**. Zgodnie z powyższą definicją naszym zadaniem T jest tu oznaczanie spamu, doświadczeniem E — **dane uczące**; pozostałe nam wyznaczyć miarę wydajności P. Może być nią na przykład stosunek prawidłowo oznaczonych wiadomości do przykładów nieprawidłowo sklasyfikowanych. Ta konkretna miara wydajności jest nazywana **dokładnością** (ang. *accuracy*) i znajduje często zastosowanie w zadaniach klasyfikujących.

Nawet jeśli pobierzesz całą Wikipedię na dysk, Twój komputer będzie przechowywał mnóstwo informacji, ale nie wykorzysta ich w żaden sposób. Dlatego pobrania kopii Wikipedii nie uznajemy za uczenie maszynowe.

Dlaczego warto korzystać z uczenia maszynowego?

Załóżmy, że chcemy napisać filtr spamu za pomocą tradycyjnych technik programistycznych (rysunek 1.1):

² Angielska nazwa niechcianych wiadomości pocztowych — „spam” (mielonka) — została zapożyczona ze słynnego skeczu grupy Monty Python pt. „Spam”. Nazwa standardowych wiadomości („ham”) oznacza w dosłownym tłumaczeniu szynkę — przyp. tłum.



Rysunek 1.1. Tradycyjne podejście

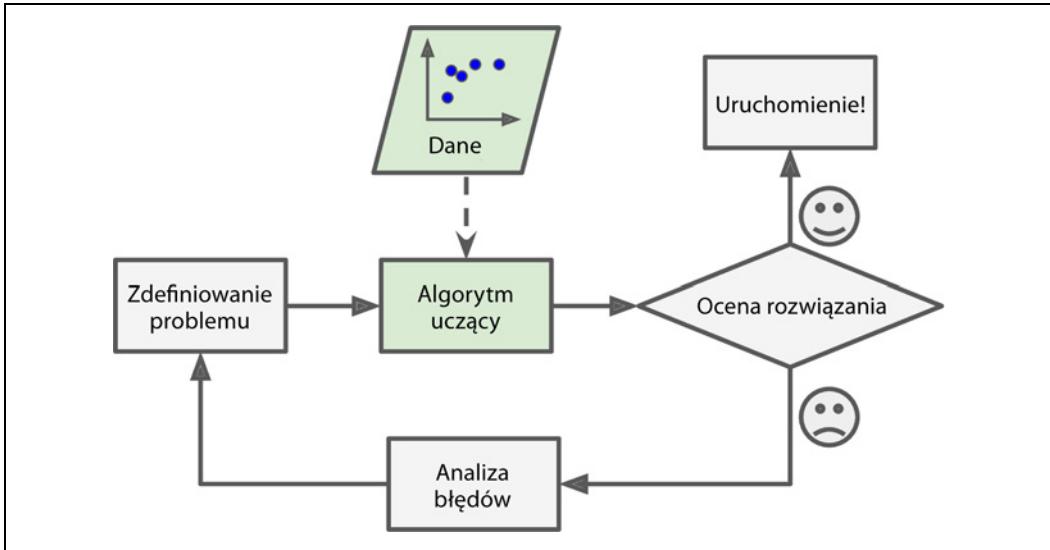
1. Najpierw musielibyśmy zastanowić się, jak wygląda klasyczny spam. Pewnie zauważysz, że niektóre wyrazy lub wyrażenia (np. „karta kredytowa”, „darmowe”, „wspaniałe”, „bez limitów”) bardzo często występują w temacie wiadomości. Być może odkryjesz również kilka innych szablonów w nazwie nadawcy, ciele wiadomości i innych obszarach e-maila.
2. Nastepnym etapem byłoby napisanie algorytmu wykrywającego każdy z zaobserwowanych szablonów. Program oznaczałby wiadomości jako spam, jeśli wykrywałby w nich kilka spośród zdefiniowanych wzorców.
3. Teraz należałoby przetestować ten program i usprawniać go poprzez ciągłe powtarzanie kroków 1. i 2.

Problem ten jest skomplikowany, a omawiany program będzie się rozrastał o coraz większą liczbę skomplikowanych reguł — jego własnoręczne utrzymanie stałoby się w końcu bardzo uciążliwe.

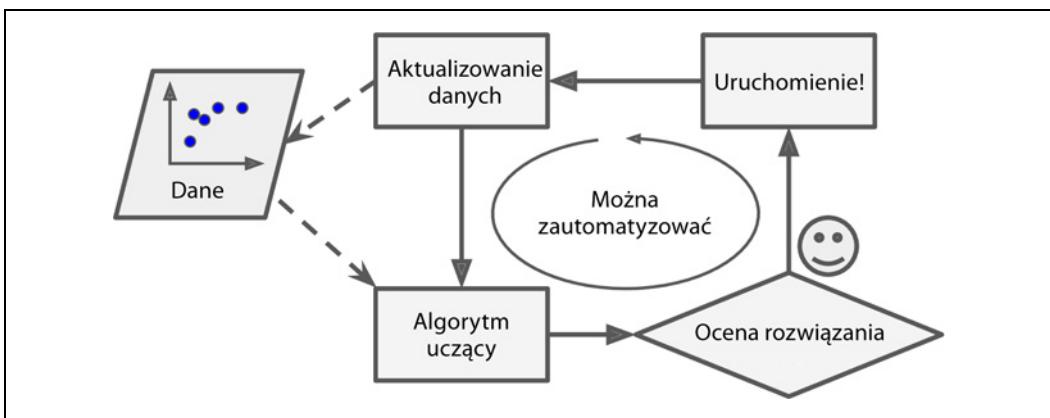
Z drugiej strony filtr spamu bazujący na algorytmach uczenia maszynowego automatycznie rozpoznaje, które słowa i wyrażenia stanowią dobre czynniki prognostyczne, poprzez wykrywanie podejrzanej często powtarzających się wzorców wyrazów w przykładowym spamie (w porównaniu do przykładów standardowych wiadomości) (rysunek 1.2). Kod takiej aplikacji jest znacznie krótszy, łatwiejszy do utrzymywania i prawdopodobnie dokładniejszy.

A gdyby spamerzy zauważyli, że blokowane są wszystkie wiadomości zawierające wyrażenie „bez limitu”, mogą je zmodyfikować na „bez ograniczeń”? Filtr spamu stworzony za pomocą tradycyjnych technik programistycznych musiałby wtedy zostać zaktualizowany tak, aby oznaczał wiadomości zawierające nowe wyrażenie. Gdyby spamerzy próbowali cały czas w jakiś sposób oszukiwać Twój filtr spamu, musiałabyś/musiałbyś bez przerwy dopisywać nowe reguły.

Natomiast filtr spamu stworzony za pomocą technik uczenia maszynowego automatycznie „zauważy” nagły wzrost częstotliwości pojawiania się wyrażenia „bez ograniczeń” w spamie oznaczonym przez użytkowników i zacznie go blokować bez Twojego udziału (rysunek 1.3).



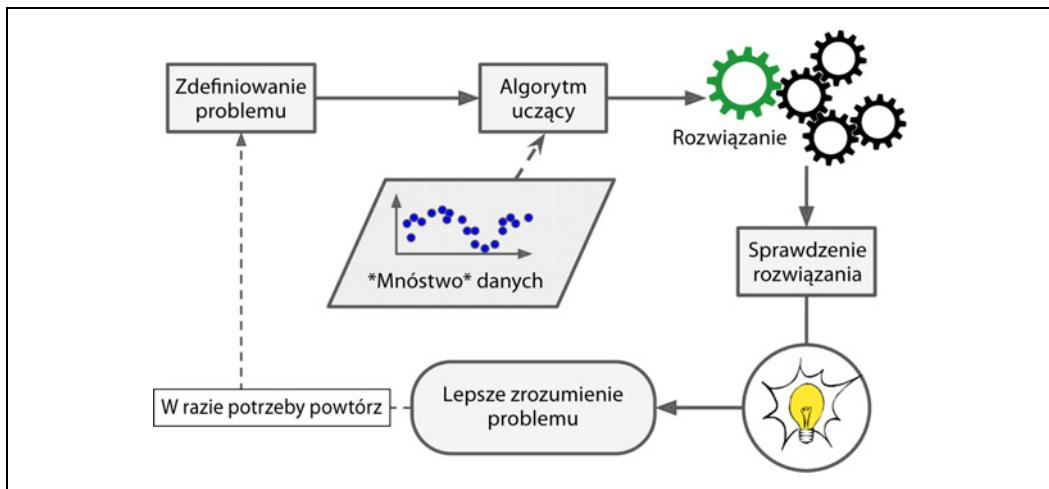
Rysunek 1.2. Podejście wykorzystujące uczenie maszynowe



Rysunek 1.3. Automatyczne dostosowywanie się do zmian

Kolejnym obszarem, w którym wyróżnia się mechanizm uczenia maszynowego, są problemy, które są zbyt złożone dla tradycyjnych metod lub dla których nie istnieją gotowe algorytmy. Przyjrzyjmy się na przykład rozpoznaniu mowy. Załóżmy, że chcesz stworzyć prostą aplikację zdolną do rozróżniania wyrazów „raz” i „trzy”. Zwróć uwagę, że słowo „trzy” zaczyna się od dźwięku o wysokim tonie („T”), dlatego moglibyśmy stworzyć algorytm wykrywający wysokość dźwięku i użyć go do rozróżniania obydwu wyrazów. Oczywiście technika ta zupełnie nie nadaje się do rozpoznawania tysięcy słów wypowiadanych w hałaśliwych miejscach przez miliony różnych osób porozumiewających się w dziesiątkach języków. Najlepszym rozwiązaniem (przynajmniej przy obecnym stanie naszej wiedzy) jest napisanie samouczającego się algorytmu na podstawie przykładowych nagrań danego słowa.

Uczenie maszynowe pomaga również w trenowaniu ludzi (rysunek 1.4): możemy przeglądać algorytmy, aby sprawdzić, czego się nauczyły (chociaż nie zawsze jest to łatwo sprawdzić). Przykładowo po wytrenowaniu filtru spamu do rozpoznawania odpowiedniej ilości spamu można z łatwością sprawdzić listę wyuczonych słów i kombinacji wyrazów uznawanych za najlepsze czynniki prognostyczne. Czasem w ten sposób możemy wykryć niespodziewane korelacje lub nowe trendy, dzięki czemu jesteśmy w stanie lepiej pojąć dany problem.



Rysunek 1.4. Uczenie maszynowe pomaga w trenowaniu ludzi

Wykorzystanie technik uczenia maszynowego do analizowania olbrzymich ilości danych może pomóc w wykrywaniu nieoczywistych wzorców. Proces ten nazywamy **wydobywaniem danych** (ang. *data mining*).

Podsumowując, uczenie maszynowe nadaje się znakomicie do:

- problemów, których rozwiązanie wymaga częstego dostrajania algorytmu lub korzystania z długich list reguł — często jeden algorytm uczenia maszynowego upraszcza aplikację i poprawia jej szybkość w stosunku do podejścia tradycyjnego;
- złożonych problemów, których nie można rozwiązać tradycyjnymi metodami — najlepsze algorytmy uczenia maszynowego są w stanie znaleźć rozwiązanie;
- zmiennych środowisk — mechanizm uczenia maszynowego potrafi dostosować się do nowych danych;
- pomagania człowiekowi w analizowaniu skomplikowanych zagadnień i olbrzymich ilości danych.

Przykładowe zastosowania

Przyjrzyjmy się konkretnym zastosowaniom zadań uczenia maszynowego, a także realizowanym w ich ramach technikom:

Analizowanie obrazów produktów znajdujących się na taśmie produkcyjnej w celu ich automatycznego klasyfikowania

Jest to klasyfikowanie obrazów realizowane zazwyczaj przez splotowe (konwolucyjne) sieci neuronowe (ang. *Convolutional Neural Networks*, CNN, zob. rozdział 14.).

Wykrywanie guzów na skanach mózgu

Jest to segmentacja semantyczna, w której każdy piksel zostaje sklasyfikowany (ponieważ interesuje nas dokładne położenie i kształt guza), także realizowana zazwyczaj przez sieć splotową.

Automatyczne klasyfikowanie nowych informacji ze świata

Jest to przetwarzanie języka naturalnego (ang. *Natural Language Processing*, NLP), a dokładniej klasyfikacja tekstu, którą można realizować za pomocą sieci rekurencyjnych (ang. *Recurrent Neural Networks*, RNN), splotowych lub transformatorów (zob. rozdział 16.).

Automatyczne oznaczanie obraźliwych komentarzy na forach dyskusyjnych

Jest to również przetwarzanie języka naturalnego, w którym wykorzystywane są te same narzędzia.

Automatyczne podsumowywanie długich dokumentów

Jest to dział przetwarzania języka naturalnego zwany streszczeniem tekstu, realizowany za pomocą tych samych narzędzi.

Tworzenie czatbota lub asystenta osobistego

Wykorzystywanych jest tu wiele składników przetwarzania języka naturalnego, w tym rozumienie języka naturalnego (ang. *Natural Language Understanding*, NLU) i moduły pytań – odpowiedzi.

Przewidywanie przyszłoroczknych dochodów firmy na podstawie wielu wskaźników wydajności

Jest to zadanie regresyjne (tzn. przewidywanie wartości), które może być realizowane za pomocą dowolnego modelu regresyjnego, np. modelu regresji liniowej lub regresji wielomianowej (zob. rozdział 4.), regresyjnej maszyny wektorów nośnych (zob. rozdział 5.), regresyjnego lasu losowego (zob. rozdział 7.) czy sztucznej sieci neuronowej (zob. rozdział 10.). Jeżeli chcesz brać pod uwagę wcześniejsze sekwencje wskaźników wydajności, możesz rozważyć użycie sieci rekurencyjnych, splotowych lub transformatorów (zob. rozdziały 15. i 16.).

Wprowadzenie do aplikacji funkcji reagowania na polecenia głosowe

Jest to rozpoznawanie mowy, wymagające przetwarzania próbek dźwiękowych: są one długimi i skomplikowanymi sekwencjami, dlatego zazwyczaj zaprzegamy do tego zadania sieci rekurencyjne, splotowe lub transformatory (zob. rozdziały 15. i 16.).

Wykrywanie oszustw popełnionych z wykorzystaniem kart kredytowych

Jest to wykrywanie anomalii (zob. rozdział 9.).

Segmentowanie klientów na podstawie dokonywanych przez nich zakupów, co pozwala wyznaczać odmienne strategie dla poszczególnych segmentów

Jest to analiza skupień (zob. rozdział 9.).

Wyświetlanie skomplikowanego, wielowymiarowego zestawu danych na zrozumiałym i szczegółowym diagramie

Jest to wizualizacja danych, w której często wykorzystuje się techniki redukcji wymiarowości (zob. rozdział 8.).

Zalecanie klientowi produktu, którym może być zainteresowany, na podstawie jego wcześniejszych zakupów

Jest to system rekomendacji. Jedną ze strategii jest wprowadzanie historii zakupów (i innych informacji o kliencie) do sztucznej sieci neuronowej (zob. rozdział 10.) i uzyskanie w rezultacie najbardziej prawdopodobnego następnego zakupu. Taka sieć neuronowa jest domyślnie uczena na sekwencjach poprzednich zakupów wszystkich klientów.

Budowanie inteligentnego bota do gier

Zadanie to jest często realizowanie za pomocą uczenia przez wzmacnianie (zob. rozdział 18.), czyli działa uczenia maszynowego, w którym agenty (takie jak boty) uczą się wybierać działania maksymalizujące uzyskiwaną nagrodę w miarę upływu czasu (np. bot może otrzymywać nagrodę za każdym razem, gdy zabierze graczowi część punktów życia) w ramach danego środowiska (takiego jak np. gra). Słynny program AlphaGo, który pokonał najlepszego gracza w Go, bazuje właśnie na uczeniu przez wzmacnianie.

Lista ta jest znacznie dłuższa, mam jednak nadzieję, że dostrzegasz teraz niesamowity zakres i ogromną złożoność zadań realizowanych w ramach uczenia maszynowego, a także mnogość rodzajów technik, które można wykorzystać do poszczególnych zadań.

Rodzaje systemów uczenia maszynowego

Istnieje tak wiele typów uczenia maszynowego, że warto podzielić je na ogólne kategorie na podstawie następujących kryteriów:

- nadzór człowieka w procesie trenowania (uczenie nadzorowane, uczenie nienadzorowane, uczenie półnadzorowane i uczenie przez wzmacnianie);
- możliwość uczenia się w czasie rzeczywistym (uczenie przyrostowe i uczenie wsadowe);
- sposób pracy — proste porównywanie nowych punktów danych ze znany punktami lub, podobnie jak robią to naukowcy, wykrywanie wzorców w danych uczących i tworzenie modelu predykcyjnego (uczenie z przykładów i uczenie z modelu).

Kryteria te nie wykluczają się wzajemnie, możesz łączyć je w dowolny sposób. Na przykład nowoczesny filtr spamu może uczyć się na bieżąco przy użyciu modelu głębokiej sieci neuronowej, korzystając z przykładowych wiadomości e-mail — w ten sposób zaprzęgamy do pracy przyrostowy, bazujący na modelu i nadzorowany system uczenia maszynowego.

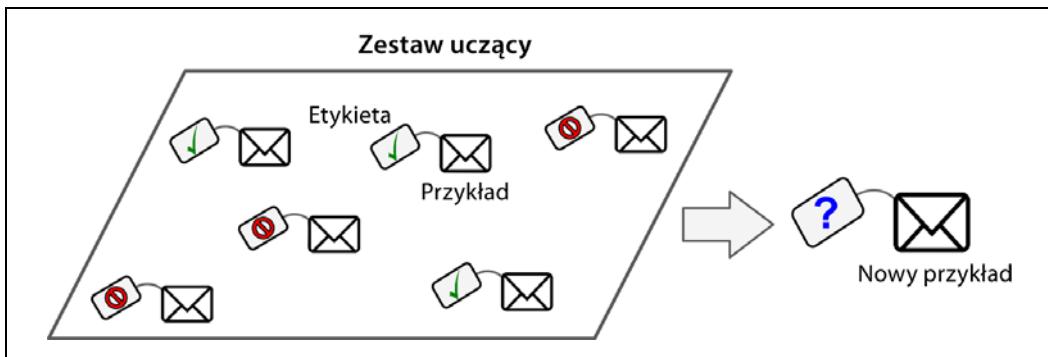
Przyjrzyjmy się nieco uważniej każdemu z wymienionych kryteriów.

Uczenie nadzorowane i uczenie nienadzorowane

Systemy uczenia maszynowego możemy podzielić na podstawie stopnia i rodzaju nadzorowania procesu uczenia. Pod tym względem uczenie maszynowe dzielimy na cztery zasadnicze rodzaje: uczenie nadzorowane, uczenie nienadzorowane, uczenie półnadzorowane i uczenie przez wzmacnianie.

Uczenie nadzorowane

W **uczeniu nadzorowanym** (ang. *supervised learning*) dane uczące przekazywane algorytmowi zawierają dołączone rozwiązania problemu, tzw. **etykiety** (ang. *labels*) (rysunek 1.5).



Rysunek 1.5. Zbiór danych uczących zaopatrzonych w etykiety, stosowany do klasyfikowania spamu (przykład uczenia nadzorowanego)

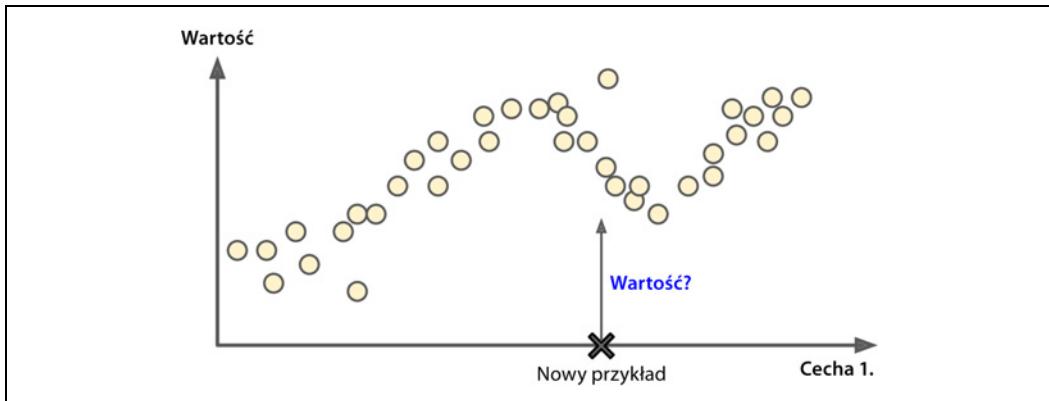
Klasycznym zadaniem uczenia nadzorowanego jest **klasyfikacja** (ang. *classification*). Filtr spamu stanowi tu dobry przykład: jest on trenowany za pomocą dużej liczby przykładowych wiadomości e-mail należących do danej **klasy** (spamu lub hamu), dzięki którym musi być w stanie klasyfikować nowe wiadomości.

Innym typowym zadaniem uczenia nadzorowanego jest przewidywanie **docelowej** wartości numerycznej, takiej jak cena samochodu, przy użyciu określonego zbioru cech (przebieg, wiek, marka itd.), zwanych **czynnikami prognostycznymi** lub **predykcyjnymi** (ang. *predictors*). Ten typ zadania nosi nazwę **regresji** (rysunek 1.6)³. Aby wyuczyć dany system, należy mu podać wiele przykładów samochodów, w tym zarówno etykiety (np. ceny), jak i czynniki prognostyczne.



W terminologii uczenia maszynowego **atrybutem** (ang. *attribute*) nazywamy typ danych (np. przebieg), natomiast **cecha** (ang. *feature*) w zależności od kontekstu ma kilka różnych znaczeń, zazwyczaj jednak mówiąc o cesze, mamy na myśli atrybut wraz z jego wartością (np. przebieg = 15 000). Wiele osób używa wymiennie słów **atrybut** i **cecha**.

³ Ciekawostka: ta dziwnie brzmiąca nazwa wywodzi się z teorii statystyki i została zaproponowana przez Francisza Galtona, gdy analizował reguły, zgodnie z którymi dzieci wysokich rodziców często bywają od nich niższe. Zjawisko redukcji wzrostu w kolejnym pokoleniu badacz ten nazwał **regresją do średniej**. Nazwą tą zostały następnie określone użytą przez niego metody służące do analizowania korelacji pomiędzy zmiennymi.



Rysunek 1.6. Problem regresyjny: przewidywanie wartości na podstawie cechy wejściowej (zazwyczaj występuje wiele cech wejściowych, a czasami również wiele wartości wynikowych)

Zwróć uwagę, że niektóre algorytmy regresyjne mogą być również używane do klasyfikowania danych i odwrotnie. Na przykład algorytm **regresji logistycznej** jest powszechnie stosowany w zadaniach klasyfikacyjnych, ponieważ może podawać wynikową wartość odpowiadającą prawdopodobieństwu przynależności do danej klasy (np. 20% szans na to, że dana wiadomość jest spamem).

Oto niektóre spośród najważniejszych algorytmów nadzorowanego uczenia maszynowego (opisywanych w tej książce):

- metoda k-najbliższych sąsiadów,
- regresja liniowa,
- regresja logistyczna,
- maszyny wektorów nośnych,
- drzewa decyzyjne i losowe lasy,
- sieci neuronowe⁴.

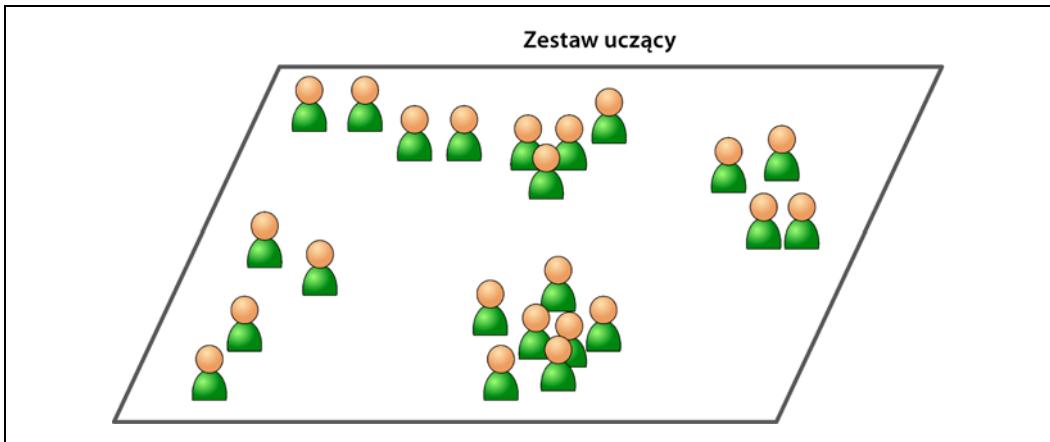
Uczenie nienadzorowane

Jak można się domyślać, w **uczeniu nienadzorowanym** (ang. *unsupervised learning*) dane uczące są nieoznaczone (rysunek 1.7). System próbuje uczyć się bez nauczyciela.

Oto kilka najważniejszych algorytmów uczenia nienadzorowanego (większość z nich została omówiona w rozdziałach 8. i 9.):

- analiza skupień (ang. *clustering*):
 - metoda k-średnich lub centroidów (ang. *k-means*),
 - hierarchiczna analiza skupień (ang. *hierarchical cluster analysis* , HCA),
 - DBSCAN,

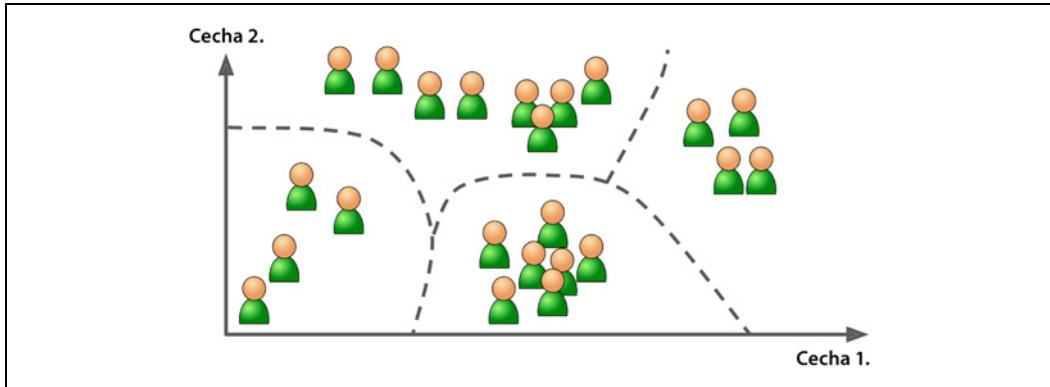
⁴ Architektury niektórych sieci neuronowych mogą być nienadzorowane (np. autokodery czy ograniczone maszyny Boltzmana). Istnieją także architektury półnadzorowane, takie jak głębokie sieci przekonań i nienadzorowane uczenie wstępne.



Rysunek 1.7. Zbiór nieoznakowanych danych uczących używanych w uczeniu nienadzorowanym

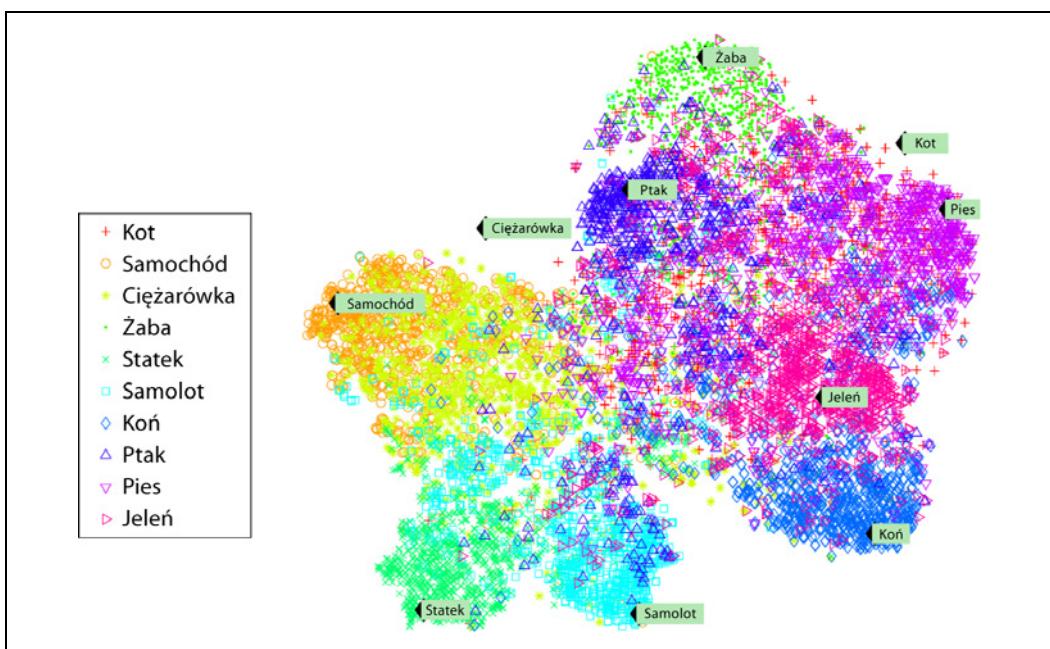
- wykrywanie anomalii i nowości (ang. *anomaly detection* i *novelty detection*):
 - jednoklasowa maszyna wektorów nośnych (ang. *support vector machine*, SVM),
 - las izolowany,
- wizualizacja i redukcja wymiarowości:
 - analiza głównych składowych (ang. *principal component analysis*, PCA),
 - jądrowa analiza głównych składowych,
 - lokalnie liniowe zanurzanie (ang. *locally linear embedding*, LLE),
 - stochastyczne zanurzanie sąsiadów przy użyciu rozkładu t (ang. *t-Distributed Stochastic Neighbor Embedding*, t-SNE),
- uczenie przy użyciu reguł asocjacyjnych:
 - algorytm Apriori,
 - algorytm Eclat.

Załóżmy na przykład, że masz do dyspozycji mnóstwo danych dotyczących osób odwiedzających Twój blog. Możesz chcieć skorzystać z **analizy skupień**, aby określić grupy podobnych użytkowników (rysunek 1.8). W dowolnym momencie możesz sprawdzić, do której grupy zalicza się dana odwiedzająca osoba: powiązania pomiędzy poszczególnymi użytkownikami są określane bez Twojej pomocy. Algorytm ten może przykładowo zauważyc, że 40% odwiedzających osób to mężczyźni będący miłośnikami komiksów przeglądający Twoją stronę głównie wieczorami, podczas gdy grupa 20% innych użytkowników to młodociani czytelnicy fantastyki naukowej, którzy zaglądają na Twój blog wyłącznie w weekendy. Jeśli użyjesz algorytmu **hierarchicznej analizy skupień**, może on dodatkowo rozdzielić grupy na mniejsze podjednostki. W ten sposób możesz łatwiej określić, jakie wpisy umieszczać dla poszczególnych grup.



Rysunek 1.8. Analiza skupień

Dobrym przykładem algorytmów uczenia nienadzorowanego są również algorytmy **wizualizujące**: wprowadzasz mnóstwo złożonych, nieoznaczonych danych, które są następnie wyświetlane w postaci punktów na dwu- lub trójwymiarowym wykresie (rysunek 1.9). Algorytmy te starają się w maksymalnym stopniu zachować pierwotną strukturę danych (np. próbując rozdzielać poszczególne skupienia w przestrzeni wejściowej, redukując zjawiska nakładania się poszczególnych klastrów na siebie), dzięki czemu możesz łatwiej przeanalizować te dane i być może odkryć jakieś nieprzewidziane wzorce.



Rysunek 1.9. Przykład wizualizacji t-SNE wskazującej semantyczny podział skupień⁵

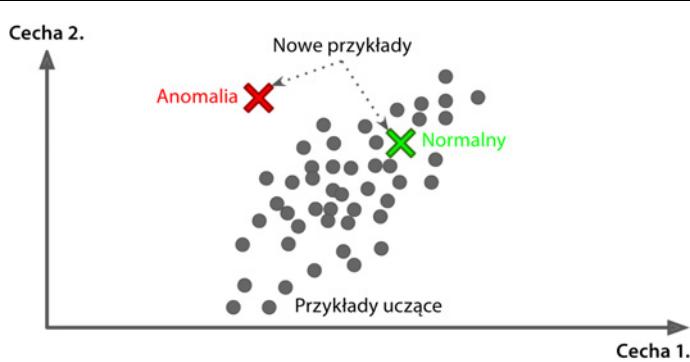
⁵ Zwróć uwagę, jak ładnie są rozdzielone pojazdy od zwierząt, konie znajdują się blisko jeleni, ale daleko od ptaków itd. Rysunek wykorzystany za zgodą Richarda Sochera i in., „Zero-Shot Learning Through Cross-Modal Transfer”, Proceedings of the 26th International Conference on Neural Information Processing Systems 1 (2013), s. 935 – 943.

Podobnym zadaniem jest **redukcja wymiarowości** (ang. *dimensionality reduction*), której celem jest uproszczenie danych bez utraty nadmiernej ilości informacji. Możemy to osiągnąć między innymi poprzez scalenie kilku skorelowanych cech w jedną. Przykładowo możemy skorelować przebieg samochodu z jego wiekiem, dzięki czemu algorytm będzie w stanie połączyć je w jedną cechę reprezentującą zużycie pojazdu. Proces ten jest nazywany **wydobywaniem cech** (ang. *feature extraction*).



Zawsze warto spróbować zredukować wymiarowość danych uczących przed dostarczeniem ich do algorytmu uczenia maszynowego (np. algorytmu uczenia nadzorowanego). Algorytm będzie działał wtedy szybciej, dane będą zabierały mniej miejsca na dysku i w pamięci operacyjnej, a w niektórych przypadkach wzrośnie także wydajność uczenia maszynowego.

Kolejnym ważnym nienadzorowanym zadaniem jest **wykrywanie anomalii** (ang. *anomaly detection*), takich jak np. nietypowe transakcje wykorzystujące kartę kredytową, co pozwala zapobiegać nielegalnym operacjom. Służy ono także do wychwytywania usterek produkcyjnych czy też automatycznego usuwania elementów odstających w danych uczących przed ich przekazaniem algorytmowi uczenia maszynowego. Takiemu systemowi prezentowane są głównie normalne przykłady w trakcie trenowania, zatem uczy się je rozpoznawać. Następnie, gdy natrafi na nowy przykład, jest w stanie stwierdzić, czy dane te są standardowe, czy raczej stanowią anomalię (rysunek 1.10). Bardzo podobnym zadaniem okazuje się **wykrywanie nowości** (ang. *novelty detection*): jego celem jest wykrywanie nowych przykładów, które różnią się od wszystkich pozostałych przykładów tworzących zestaw danych uczących. Konieczne jest w tym przypadku korzystanie z bardzo „czystego” zestawu danych, pozbawionego przykładów, które powinny być wykrywane przez algorytm. Na przykład jeżeli dysponujesz tysiącami zdjęć psów, z których 1% reprezentuje rasę chihuahua, to algorytm wykrywania nowości nie powinien traktować nowych zdjęć tej rasy jako nowinek. Z drugiej strony algorytm wykrywania anomalii może traktować zdjęcia rasy chihuahua jako tak rzadkie i tak odmienne od zdjęć innych ras, że może zaklasyfikować je jako anomalie (bez urazy dla przedstawicieli rasy chihuahua).



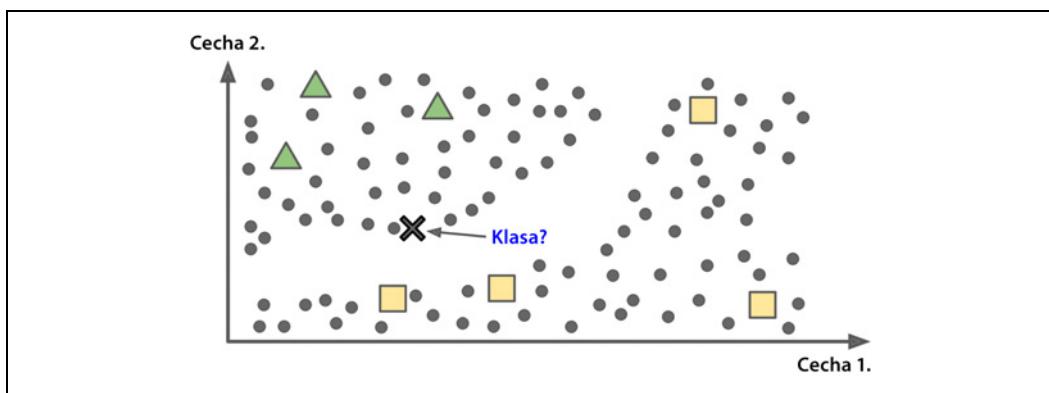
Rysunek 1.10. Wykrywanie anomalii

Jeszcze jednym powszechnie stosowanym zadaniem uczenia nienadzorowanego jest **uczenie przy użyciu reguł asocjacyjnych** (ang. *association rule learning*), którego celem jest analiza ogromnej ilości danych i wykrycie interesujących zależności pomiędzy atrybutami. Założymy, że jesteś właściwą właścicielką/właścicielem supermarketu. Przetworzenie danych dotyczących obrotów za pomocą algorytmu

reguł asocjacyjnych pozwoliłoby nam odkryć, że osoby kupujące keczup i czipsy zazwyczaj zaopatrują się również w steki. Dzięki tej wiedzy możesz na przykład umieścić te produkty niedaleko siebie.

Uczenie półnadzorowane

Oznaczanie danych etykietami jest zazwyczaj bardzo czasochłonne i kosztowne, dlatego często mamy do czynienia z danymi składającymi się z większości nieoznakowanych i tylko odrobiny oznaconych przykładów. Niektóre algorytmy radzą sobie z częściowo oznaconymi danymi. Jest to tak zwane **uczenie półnadzorowane** (ang. *semisupervised learning*) (rysunek 1.11).



Rysunek 1.11. Uczenie półnadzorowane z dwiema klasami (trójkątami i kwadratami): przykłady nieoznakowane (kółka) pomagają zaklasyfikować nowy przykład (krzyżyk) do trójkątów, a nie do kwadratów, pomimo tego, że znajduje się bliżej oznaconego kwadratu

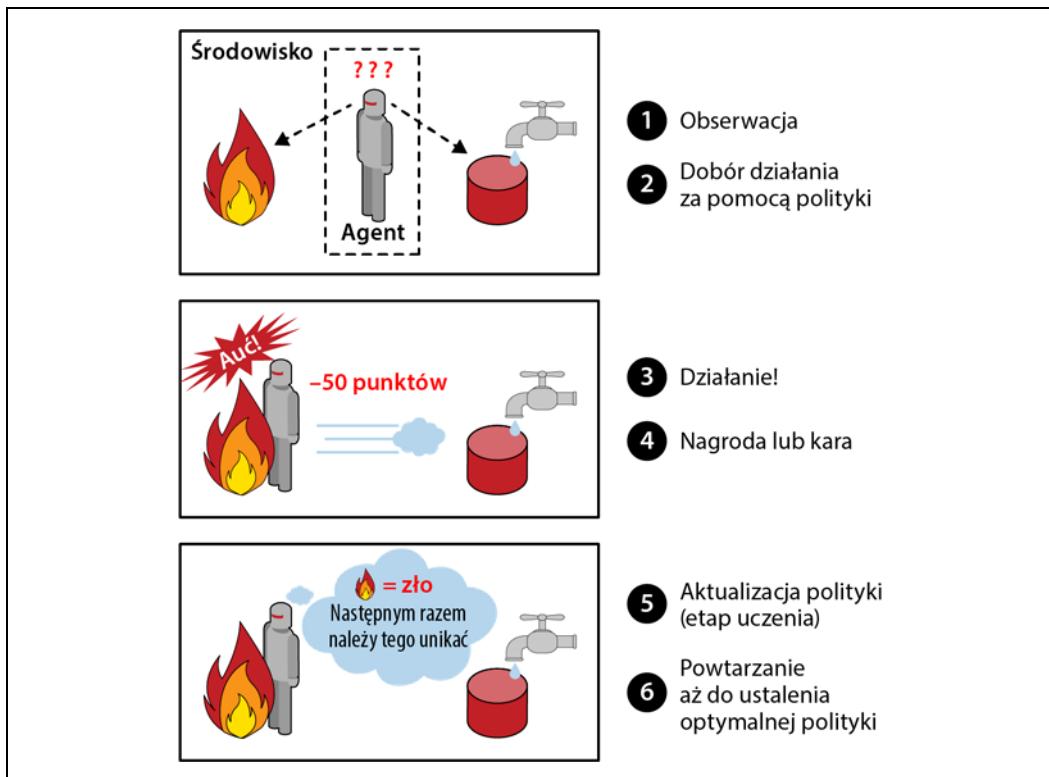
Dobrym przykładem zastosowania tego typu metod są niektóre usługi przechowywania obrazów, np. Zdjęcia Google. Po przesłaniu do takiej usługi wszystkich zdjęć rodzinnych rozpoznaje ona automatycznie, że na zdjęciach 1., 5. i 11. pojawia się osoba A, natomiast do fotografii 2., 5. i 7. pozuje osoba B. Na tym etapie mamy do czynienia z uczeniem nienadzorowanym (analizą skupień). Teraz wystarczy, że podasz imiona poszczególnych osób. Każda osoba potrzebuje tylko jednej etykiety⁶, co wystarczy do oznakowania wszystkich osób na każdym zdjęciu i znacznie ułatwia wyszukiwanie zdjęć.

Większość algorytmów półnadzorowanych stanowi kombinację algorytmów uczenia nadzorowanego i nienadzorowanego. Na przykład **głębokie sieci przekonań** (ang. *deep belief networks* — DBN) bazują na ułożonych warstwowo elementach nienadzorowanych, zwanych **ogranicznymi maszynami Boltzmanna** (ang. *restricted Boltzmann machines* — RBM). Maszyny te są uczone sekwencyjnie w nienadzorowany sposób, a następnie cały system zostaje dostrojony przy użyciu technik uczenia nadzorowanego.

⁶ Tak wygląda wyidealizowana sytuacja. W praktyce usługa ta często tworzy kilka skupień dla jednej osoby, a czasami myli dwie podobne do siebie osoby, dlatego należy każdej osobie przydzielić kilka etykiet oraz własnoręcznie oczyścić niektóre skupienia.

Uczenie przez wzmacnianie

Uczenie przez wzmacnianie (ang. *reinforcement learning*) to zupełnie inna bajka. System uczący, zwany w tym kontekście **agentem**, może obserwować środowisko, dobierać i wykonywać czynności, a także odbierać **nagrody** (lub **kary** będące ujemnymi nagrodami) (rysunek 1.12). Musi następnie samodzielnie nauczyć się najlepszej strategii (nazywanej **polityką**, ang. *policy*), aby uzyskiwać jak największą nagrodę. Polityka definiuje rodzaj działania, jakie agent powinien wybrać w danej sytuacji.



Rysunek 1.12. Uczenie przez wzmacnianie

Na przykład metody uczenia przez wzmacnianie są używane w wielu modelach robotów do nauki chodzenia. Również program AlphaGo firmy DeepMind był trenowany przez wzmacnianie: zrobiło się o nim głośno w maju 2017 roku, gdy został przezeń pokonany dotychczasowy mistrz świata w grę *Go* — Ke Jie. Aplikacja ta zdefiniowała politykę gwarantującą zwycięstwo, analizując miliony rozgrywek i rozgrywając mnóstwo partii przeciwko samej sobie. Warto zauważyć, że algorytmy uczące zostały wyłączone podczas partii z ludzkim przeciwnikiem; program AlphaGo korzystał jedynie z wyuczonej polityki.

Uczenie wsadowe i uczenie przyrostowe

Innym kryterium stosowanym do klasyfikowania systemów uczenia maszynowego jest możliwość treningu przyrostowego przy użyciu strumienia nadsyłanych danych.

Uczenie wsadowe

W mechanizmie **uczenia wsadowego** (ang. *batch learning*) system nie jest w stanie trenować przyrostowo — do jego nauki muszą wystarczyć wszystkie dostępne dane. Wymaga to zazwyczaj poświęcenia dużej ilości czasu i zasobów, dlatego najczęściej proces ten jest przeprowadzany w trybie offline. System najpierw jest uczony, a następnie zostaje wdrożony do cyklu produkcyjnego i już więcej nie jest trenowany; korzysta jedynie z dotychczas zdobytych informacji. Zjawisko to bywa nazywane **uczeniem offline** (ang. *offline learning*).

Jeśli chcesz, aby system uczenia wsadowego brał pod uwagę nowe dane (np. nowe rodzaje spamu), musisz od podstaw wytrenować nową wersję systemu przy użyciu wszystkich dostępnych danych (nowych i starych), następnie wyłączyć stary system i zastąpić go nowym.

Na szczęście cały proces trenowania, oceniania i uruchamiania systemu uczenia maszynowego można dość łatwo zautomatyzować (co widać na rysunku 1.3), dlatego nawet układ uczenia wsadowego jest w stanie dostosowywać się do zmian. Wystarczy zaktualizować dane i z odpowiednią częstotliwością trenować system od podstaw.

Rozwiązań to jest proste i często skuteczne, ale trenowanie za pomocą pełnego zbioru danych uczących nierzaz trwa wiele godzin, dlatego zazwyczaj trenujemy nowy system raz na dobę, a czasami nawet raz w tygodniu. Jeśli Twój system musi dostosowywać się do szybko zmieniających się danych (np. podczas prognozowania cen akcji giełdowych), będziesz potrzebować dynamiczniejszego mechanizmu.

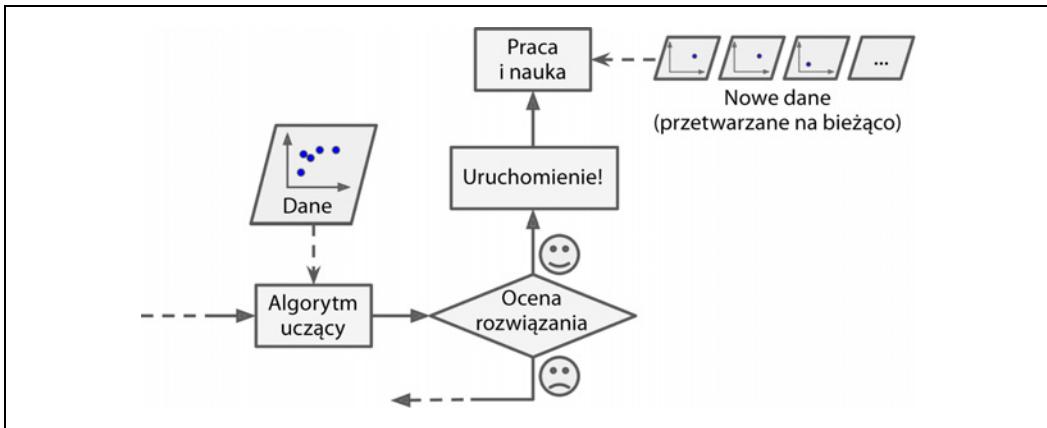
Ponadto uczenie się na pełnym zbiorze danych pochłania mnóstwo zasobów obliczeniowych (mocy procesora, miejsca i szybkości dysku, szybkości sieci itd.). Jeśli masz dużo danych i trenujesz system codziennie od podstaw, będzie Cię to kosztowało wiele pieniędzy. W przypadku danych o naprawdę pokaźnych rozmiarach korzystanie z algorytmu uczenia wsadowego może być wręcz niemożliwe.

Zwrócić na koniec uwagę, że jeśli system ma być zdolny do autonomicznej nauki przy użyciu ograniczonych zasobów (np. aplikacja na smartfon albo marsjański łazik), to magazynowanie olbrzymich ilości danych i przeznaczanie zasobów na codzienne wielogodzinne trenowanie systemu mogą zyczajnie uniemożliwić normalną pracę urządzenia.

Na szczęście istnieje lepsze rozwiązanie dla wymienionych powyżej przykładów: algorytmy zdolne do uczenia przyrostowego.

Uczenie przyrostowe

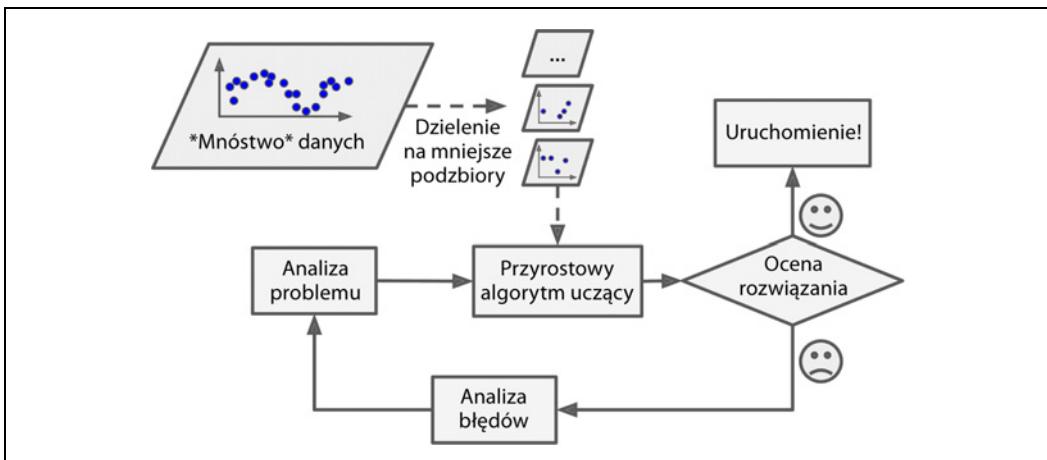
W procesie **uczenia przyrostowego** (ang. *online learning*) system jest trenowany na bieżąco poprzez sekwencyjne dostarczanie danych, które mogą być pojedyncze lub przyjmować postać tzw. **minipakietów/minigrup** (ang. *mini-batches*), czyli niewielkich zbiorów. Każdy krok uczący jest szybki i niezbyt kosztowny, dlatego system jest w stanie uczyć się na bieżąco przy użyciu nowych danych, gdy tylko się pojawią (rysunek 1.13).



Rysunek 1.13. W uczeniu przyrostowym model zostaje wytrenowany i wdrożony do środowiska produkcyjnego, gdzie dalej uczy się w miarę przetwarzania dostarczanych danych

Uczenie przyrostowe sprawdza się znakomicie w systemach odbierających ciągły strumień nowych danych (np. ceny akcji) oraz wymagających szybkiego lub autonomicznego dostosowywania się do nowych warunków. Metody te przydają się również przy ograniczonych zasobach obliczeniowych: dane użyte do nauki nie są już potrzebne i można się ich pozbyć (chyba że chcesz zachować możliwość cofnięcia się do wcześniejszego stanu i „odtworzyć ponownie” dane). W ten sposób możesz mocno zaoszczędzić na przestrzeni dyskowej.

Algorytmy uczenia przyrostowego są w stanie również trenować systemy za pomocą dużych zbiorów danych niemieszczących się w pamięci urządzenia (jest to tzw. **uczenie pozakorowe** — ang. *out-of-core learning*). Następuje wczytanie części danych i trenowanie systemu za ich pomocą, po czym proces zostanie powtórzony, aż do wczytania całego zbioru danych uczących (rysunek 1.14).



Rysunek 1.14. Przetwarzanie dużych ilości danych za pomocą algorytmu uczenia przyrostowego

Jednym z najważniejszych parametrów systemów uczenia przyrostowego jest szybkość, z jaką dostosowują się do zmieniających się danych — jest to tzw. **współczynnik uczenia** (ang. *learning rate*). Wy-

soka wartość tego współczynnika oznacza, że system będzie szybko dostosowywał się do nowych danych, ale jednocześnie dość szybko zapominał o starych danych (raczej nie chcesz, aby filtr spamu oznaczał jedynie najnowsze rodzaje spamu). Natomiast system o niskim współczynniku uczenia będzie cechował się większą bezwładnością, co oznacza, że będzie uczył się wolniej, ale jednocześnie będzie bardziej odporny na zasumienie nowych danych oraz na sekwencje niereprezentatywnych punktów danych (elementów odstających).

Z kolei dużym problemem uczenia przyrostowego jest stopniowy spadek wydajności systemu w przypadku dostarczenia mu nieprawidłowych danych. Klienci korzystający z danego systemu na pewno to zauważą. Przykładowo nieprawidłowe dane mogą pochodzić z uszkodzonego czujnika w robocie lub od osoby zasypujączej silnik przeglądarki danymi w celu podbicia swojego rankingu w wynikach wyszukiwania. Aby zmniejszyć to ryzyko, musisz uważnie obserwować swój system i świadomie wyłączyć proces uczenia (a także ewentualnie przywrócić stan do wcześniejszego) po wykryciu spadku szybkości algorytmu. Możesz także śledzić przychodzące dane i reagować na wszelkie nieprawidłowości (np. za pomocą algorytmu wykrywania anomalii).

Uczenie z przykładów i uczenie z modelu

Możemy również podzielić systemy uczenia maszynowego pod względem **uogólniania**. Większość zadań uczenia maszynowego polega na sporządzaniu prognoz. Oznacza to, że na podstawie określonej liczby próbek uczących system musi być w stanie uzyskiwać dobre prognozy (generalizować) wyników dla nie widzianych wcześniej przykładów. Uzyskanie dobrej szybkości algorytmu wobec danych uczących jest pożądane, ale niewystarczające — prawdziwy cel stanowi dobra wydajność wobec nowych przykładów.

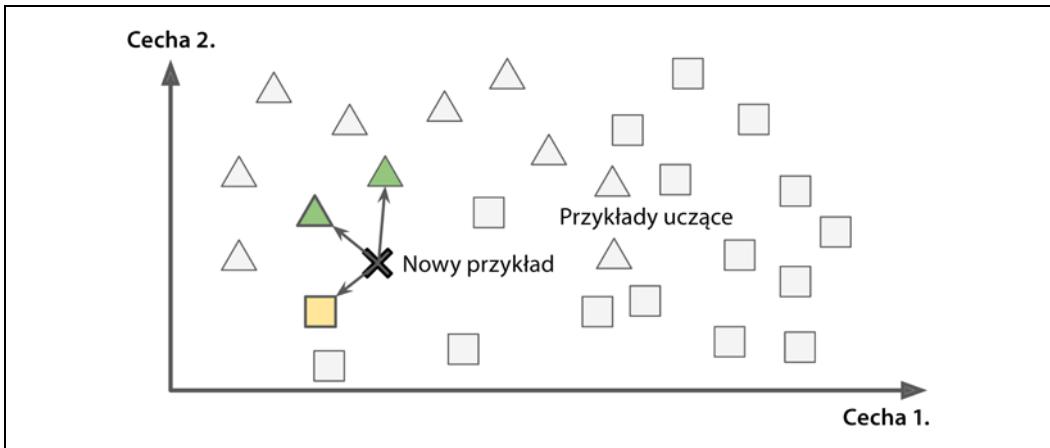
Znamy dwa główne mechanizmy uogólniania: uczenie z przykładów i uczenie z modelu.

Uczenie z przykładów

Bogdaj najprostszą formą nauki jest „wykuwanie na blachę”. Gdybyśmy stworzyli w ten sposób filtr spamu, oznaczałby po prostu wszystkie wiadomości identyczne z oznakowanymi przez użytkowników — rozwiązań nie najgorsze, ale zarazem nie najlepsze.

Zamiast oznaczać wiadomości identyczne z rozpoznanym wcześniej spamem możemy zaprogramować filtr w taki sposób, aby znakował również wiadomości bardzo podobne do powszechnie rozpoznawanych wzorców. Potrzebna staje się jakaś **miara podobieństwa** dwóch wiadomości e-mail. Może ją stanowić (w wielkim uproszczeniu) porównanie liczby takich samych słów występujących w obydwu wiadomościach. Filtr mógłby oznaczać wiadomość jako spam, jeśli będzie miała wiele słów wspólnych ze znany spamem.

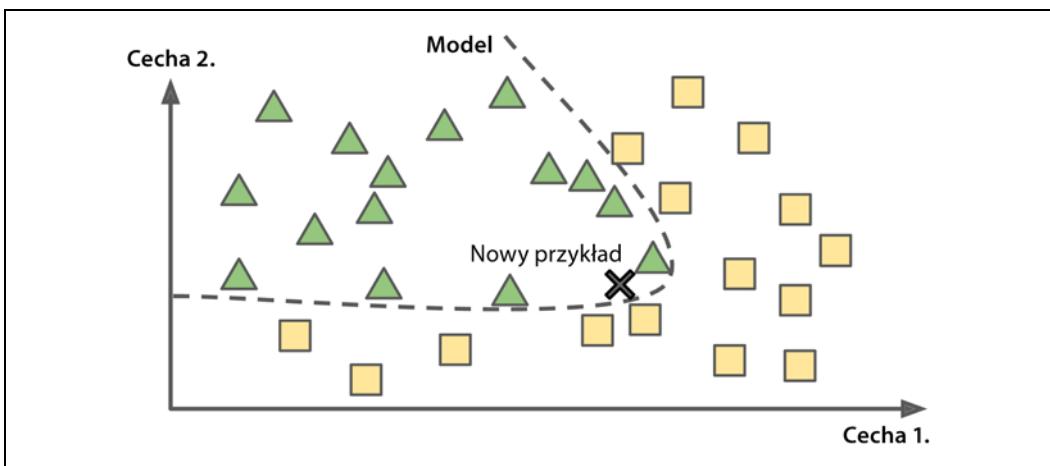
Mamy tu do czynienia z **uczeniem z przykładów** (ang. *instance-based learning*): system uczy się przykładów „na pamięć”, a następnie za pomocą miary podobieństwa porównuje je z wyuczonymi przykładami (lub ich podzbiorem). Na rysunku 1.15 nowy przykład zostałby zaklasyfikowany jako trójkąt, ponieważ większość najbardziej podobnych przykładów przynależy do tej klasy.



Rysunek 1.15. Uczenie z przykładów

Uczenie z modelu

Innym sposobem uogólniania wyników uzyskiwanych z danych uczących jest stworzenie modelu z tych przykładów i użycie go do **przewidywania (prognozowania, ang. prediction)**. Jest to **uczenie z modelu (ang. model-based learning)** (rysunek 1.16).



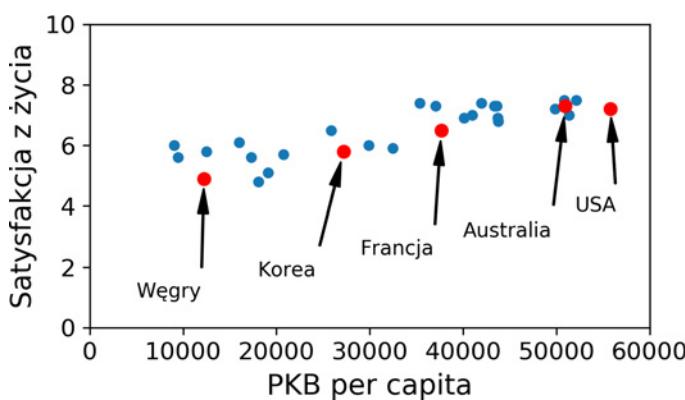
Rysunek 1.16. Uczenie z modelu

Załóżmy na przykład, że chcemy sprawdzić, czy pieniądze dają szczęście. W tym celu moglibyśmy pobrać dane *Better Life Index* ze strony Organizacji Współpracy Gospodarczej i Rozwoju (OECD, <http://stats.oecd.org/index.aspx?DataSetCode=BLI>), a także statystyki PKB per capita z witryny Międzynarodowego Funduszu Walutowego (IMF, <https://stats.oecd.org/index.aspx?DataSetCode=BLI>). Teraz wystarczyłoby połączyć obydwie tabele i posortować dane dotyczące PKB. Tabela 1.1 prezentuje fragment uzyskanych informacji.

Tabela 1.1. Czy pieniądze przynoszą szczęście?

Państwo	PKB per capita (w dolarach amerykańskich)	Satysfakcja z życia
Węgry	12 240	4,9
Korea	27 195	5,8
Francja	37 675	6,5
Australia	50 962	7,3
Stany Zjednoczone	55 805	7,2

Wygenerujmy teraz wykres danych dla tych krajów (rysunek 1.17).



Rysunek 1.17. Czy widzisz tu trend?

Możemy dostrzec tu pewien trend. Pomimo tego, że dane są **zaszumione** (tzn. częściowo losowe), wygląda na to, że poziom satysfakcji z życia wzrasta w sposób mniej więcej liniowy wraz ze wzrostem produktu krajowego brutto na osobę. Postanawiasz zatem stworzyć model satysfakcji z życia jako funkcji liniowej wobec parametru PKB per capita. Etap ten nazywamy **doborem modelu**: wybraliśmy **model liniowy** satysfakcji z życia wykorzystujący tylko jeden atrybut — PKB per capita (równanie 1.1).

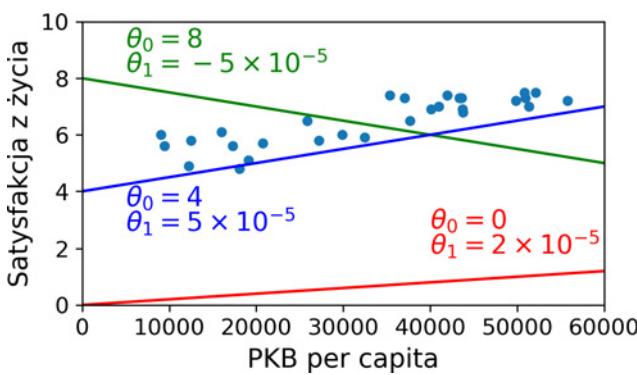
Równanie 1.1. Prosty model liniowy

$$\text{satysfakcja_z_życia} = \theta_0 + \theta_1 \times \text{PKB_per_capita}$$

Model ten zawiera dwa **parametry**, θ_0 i θ_1 ⁷. Poprzez modyfikowanie tych parametrów możesz za ich pomocą uzyskać dowolną funkcję liniową, co zostało ukazane na rysunku 1.18.

Zanim zaczniesz korzystać z modelu, musisz zdefiniować wartości jego parametrów θ_0 i θ_1 . Skąd mamy wiedzieć, które wartości nadają się najlepiej dla danego modelu? Aby móc odpowiedzieć na to pytanie, należy zdefiniować miarę efektywności (jakości dopasowania). Możesz wyznaczyć **funkcję użyteczności** (zwana także **funkcją dopasowania**), mówiącą nam, jak **dobry** jest dany model, lub **funkcję kosztu**, mającą przeciwnie zastosowanie. W zagadnieniach wykorzystujących regresję liniową

⁷ Zgodnie z konwencją grecką litera θ (teta) często symbolizuje parametry modelu.



Rysunek 1.18. Kilka wybranych modeli liniowych

zazwyczaj jest stosowana funkcja kosztu mierząca odległość pomiędzy przewidywaniami modelu liniowego a przykładami uczącymi; naszym zadaniem jest zminimalizowanie tego dystansu.

W tym właśnie miejscu przydaje się algorytm regresji liniowej: dostarczamy mu dane uczące, a on określa parametry najlepiej pasujące do danego modelu liniowego. Proces ten nosi nazwę uczenia (trenowania) modelu. W naszym przykładzie algorytm regresji liniowej wyznacza następujące optymalne wartości parametrów: $\theta_0 = 4,85$ i $\theta_1 = 4,91 \times 10^{-5}$.



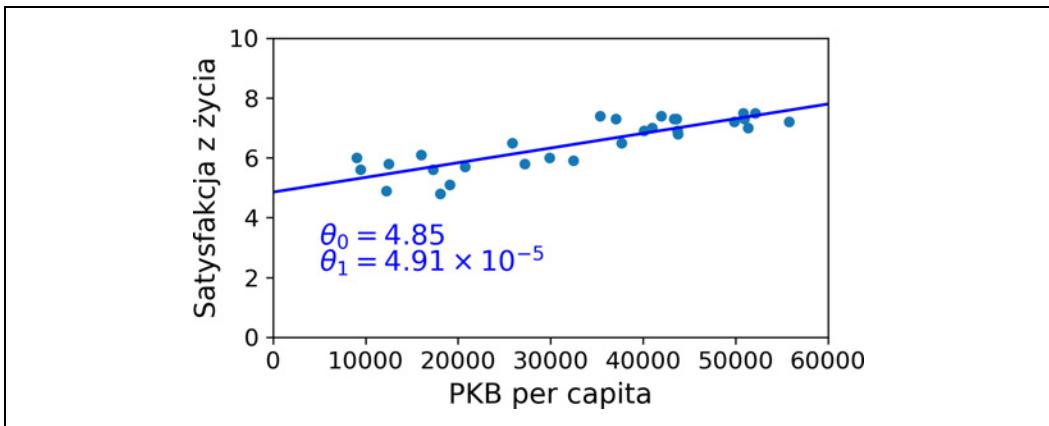
Aby nie było zbyt łatwo, ten sam wyraz „model” może dotyczyć **typu modelu** (np. model regresji liniowej), **w pełni określonej architektury modelu** (np. model regresji liniowej z jednym wejściem i jednym wyjściem) lub **ostatecznie wytrenowanego modelu** gotowego do uzyskiwania prognoz (np. model regresji liniowej z jednym wejściem i jednym wyjściem, z wyznaczonymi parametrami $\theta_0 = 4,85$ i $\theta_1 = 4,91 \times 10^{-5}$). Na dobór modelu składa się wyznaczenie typu modelu i sprecyzowanie jego architektury. Przez uczenie modelu rozumiemy uruchomienie algorytmu wyszukującego parametry modelu optymalnie dopasowane do danych uczących (oraz, przy odrobinie szczęścia, uzyskujące dobre prognozy na podstawie nowych danych).

Teraz nasz model jest maksymalnie dopasowany do danych uczących (jak na model liniowy), o czym możemy się przekonać na rysunku 1.19.

Możemy w końcu uruchomić model, aby przeprowadził prognozy. Założymy na przykład, że chcemy się dowiedzieć, jak szczęśliwi są Cypryjczycy. Informacji tej nie znajdziemy w bazie danych OECD. Ale możemy się posłużyć modelem, aby uzyskać wiarygodne przewidywania: sprawdzamy wartość PKB per capita dla Cypru (22 587 dolarów), a po uruchomieniu modelu dowiadujemy się, że współczynnik satysfakcji z życia oscyluje wokół wartości $4,85 + 22\ 587 \times 4,91 \times 10^{-5} = 5,96$.

Zaostrzę Ci apetyt, pokazując w przykładzie 1.1 kod Pythona służący do wczytywania danych, ich przygotowania⁸, utworzenia wykresu oraz wyuczenia modelu liniowego i prognozowania wyników⁹.

⁸ Definicja funkcji `prepare_country_stats()` nie została tu zaprezentowana (zajrzyj do notatnika Jupyter, jeżeli interesują Cię takie szczegóły). Jest to po prostu nudny kod wykorzystujący bibliotekę pandas, łączący dane dotyczące PKB i satysfakcji z życia w jeden obiekt.



Rysunek 1.19. Model liniowy najlepiej dopasowany do danych uczących

Przykład 1.1. Uczenie i uruchamianie modelu liniowego za pomocą biblioteki Scikit-Learn

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

# Wczytuje dane
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="bd")

# Przygotowuje dane
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["PKB per capita"]]
y = np.c_[country_stats["Satysfakcja z życia"]]

# Wizualizuje dane
country_stats.plot(kind='scatter', x="PKB per capita", y='Satysfakcja z życia')
plt.show()

# Dobiera model liniowy
model = sklearn.linear_model.LinearRegression()

# Trenuje model
model.fit(X, y)

# Prognozuje wyniki dla Cypru
X_new = [[22587]] # PKB per capita dla Cypru
print(model.predict(X_new)) # Generuje wynik [[ 5.96242338]]
```

⁹ Nie szkodzi, jeżeli nie do końca rozumiesz zamieszczony kod. W następnych rozdziałach zajmiemy się omówieniem biblioteki Scikit-Learn.

¹⁰ Definicja funkcji `prepare_country_stats()` nie została tu zaprezentowana (zajrzyj do notatnika Jupyter, jeżeli interesują Cię takie szczegóły). Jest to po prostu nudny kod wykorzystujący bibliotekę pandas, łączący dane dotyczące PKB i satysfakcji z życia w jeden obiekt.

¹¹ Nie szkodzi, jeżeli nie do końca rozumiesz zamieszczony kod. W następnych rozdziałach zajmiemy się omówieniem biblioteki Scikit-Learn.



Gdybyś użyła/użył algorytmu uczenia z przykładów, zauważałbyś/zauważałbyś, że wartość PKB najbardziej zbliżoną do Cypru ma Słowenia (20 732 dolary), a skoro dzięki danym OECD wiemy, że współczynnik satysfakcji z życia wynosi u Słowenów 5,7, moglibyśmy w drodze analogii stwierdzić, że jego wartość dla Cypryjczyków powinna być bardzo podobna. Po nieznacznym „oddaleniu skali” i przyjrzeniu się dwóm kolejnym państwom o zbliżonej wartości satysfakcji z życia okaże się, że są to Portugalia (5,1) i Hiszpania (6,5). Średnia tych trzech wartości wynosi 5,77, co stanowi wartość całkiem zbliżoną do wyliczonej przez nasz model. Wspomniany tu prosty algorytm nosi nazwę regresji **k-najbliższych sąsiadów** (ang. *k-nearest neighbors*); w tym przykładzie $k = 3$.

Aby zastąpić algorytm regresji liniowej algorytmem k-najbliższych sąsiadów, wystarczy podmienić dwa wiersze:

```
import sklearn.linear_model  
model = sklearn.linear_model.LinearRegression()  
  
na następujące:  
  
import sklearn.neighbors  
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

Jeśli wszystko zostało dobrze wykonane, Twój model powinien wyliczać dobre prognozy. W przeciwnym wypadku może być konieczne wykorzystanie większej liczby atrybutów (stopy zatrudnienia, zdrowia społeczeństwa, zanieczyszczenia powietrza itd.), zdobycie więcej danych uczących lub danych lepszej jakości, ewentualnie skorzystanie z wydajniejszego modelu (np. algorytmu regresji wielomianowej).

Podsumowując:

- Przeanalizowaliśmy dane.
- Wyбралиśmy model.
- Wytrenowaliśmy go na danych uczących (np. algorytm uczący wyszukał wartości parametrów pozwalających na zminimalizowanie funkcji kosztu).
- Na końcu wykorzystaliśmy wytrenowany model do prognozowania wyników dla nowych przypadków (jest to tzw. **wnioskowanie**) z nadzieją, że będzie on skutecznie generalizował zdobytą wiedzę.

Tak właśnie wygląda typowy projekt uczenia maszynowego. W rozdziale 2. doświadczysz tego na własnej skórze podczas przygotowywania projektu od podstaw.

Poruszyliśmy do tej pory wiele zagadnień: wiesz już, czym jest uczenie maszynowe, dlaczego okazuje się przydatne i jakie są podstawowe kategorie tej dziedziny, a także znasz już podstawowy cykl pracy nad projektem uczenia maszynowego. Spójrzmy teraz, co może pójść nieprawidłowo w procesie uczenia i uniemożliwić uzyskiwanie dokładnych prognoz.

Główne problemy uczenia maszynowego

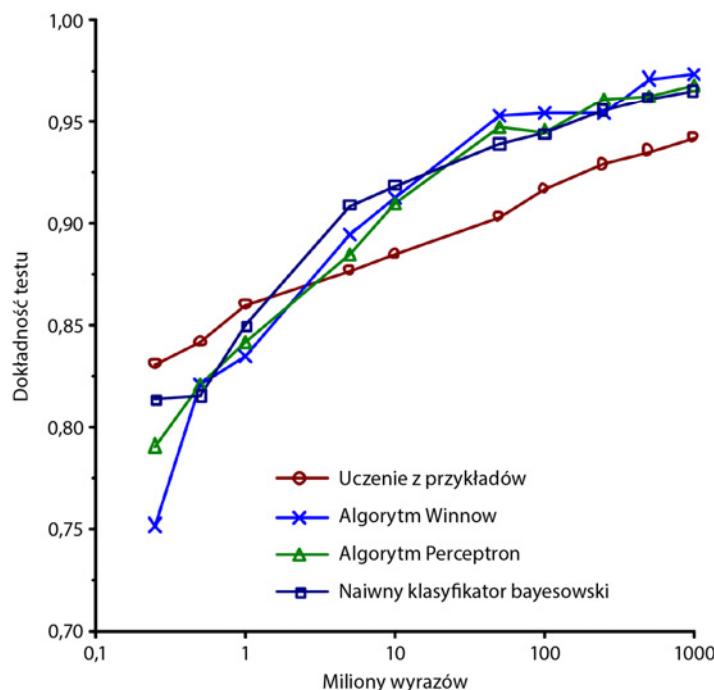
Skoro naszym głównym zadaniem jest dobór algorytmu uczącego i wyuczenie go za pomocą określonych danych, to dwoma największymi zagrożeniami są „zły algorytm” i „złe dane”. Zaczniemy od przyjrzenia się niewłaściwym danym.

Niedorzeczna efektywność danych

W słynnym dokumencie (<https://dl.acm.org/citation.cfm?id=1073017>) z 2001 roku badacze z firmy Microsoft, Michele Banko i Eric Brill, udowodnili, że różne algorytmy uczenia maszynowego, włącznie z najprostszymi, po przesłaniu im odpowiedniej ilości danych osiągają niemal identyczną wydajność wobec złożonych problemów ujednoznacznie w języku naturalnym¹² (rysunek 1.20).

Autorzy ujmują to następująco: „Wyniki te sugerują, że warto zastanowić się nad przeniesieniem środka ciężkości w kompromisie pomiędzy poświęcaniem czasu i pieniędzy na tworzenie algorytmów a przeznaniem zasobów na rozwój korpusu językowego”.

Koncepcja przewagi znaczenia danych nad algorytmami w rozwiązywaniu złożonych problemów została spopularyzowana przez Petera Nordviga i in. w dokumencie *The Unreasonable Effectiveness of Data* (<https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/35179.pdf>), opublikowanym w 2009 roku¹³. Należy jednak zauważyć, że ciągle są popularne niewielkie i średniej wielkości zbiory danych, a pozyskanie ich większej ilości często bywa bardzo trudne lub kosztowne, dlatego nie bagateliczujmy znaczenia algorytmów.



Rysunek 1.20. Porównanie znaczenia danych i algorytmów¹⁴

¹² Na przykład rozróżnianie znaczeń wyrazów „buk”, „Bug” i „bóg” w zależności od kontekstu.

¹³ Peter Norvig i in., *The Unreasonable Effectiveness of Data*, „IEEE Intelligent Systems” 24, no. 2 (2009), s. 8 – 12.

¹⁴ Rysunek zamieszczony za zgodą Michele Banko i Erica Brilla, *Scaling to Very Very Large Corpora for Natural Language Disambiguation*, „Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics” (2001), s. 26 – 33.

Niedobór danych uczących

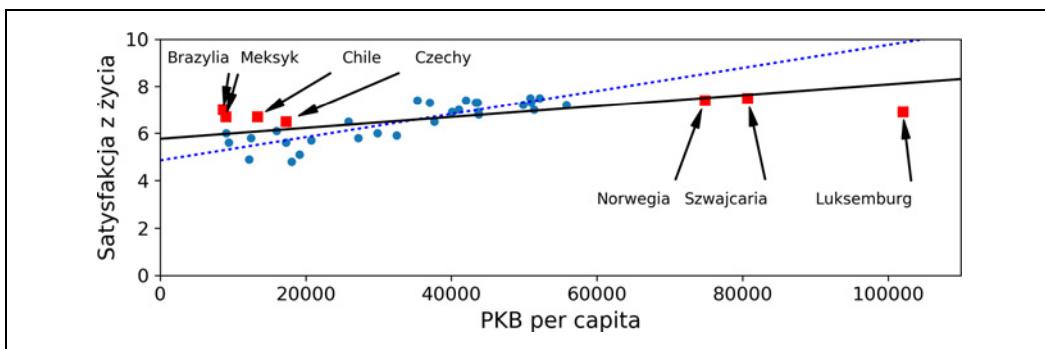
Aby nauczyć berbecia, czym jest jabłko, wystarczy wskazać mu ten owoc i powiedzieć „jabłko” (najprawdopodobniej trzeba to jeszcze kilkakrotnie powtórzyć). Dziecko teraz będzie w stanie rozpoznać jabłka w najróżniejszych kolorach i kształtach. Mały geniusz.

Niestety techniki uczenia maszynowego znajdują się pod tym względem daleko w tyle — żeby większość algorytmów działała prawidłowo, należy im zapewnić mnóstwo danych. Nawet w przypadku bardzo prostych problemów należy zazwyczaj dostarczyć tysiące przykładów, natomiast bardziej zaawansowane zagadnienia, takie jak rozpoznawanie mowy, wymagają nawet milionów próbek (chyba że jesteś w stanie wykorzystać fragmenty już istniejącego modelu).

Niereprezentatywne dane uczące

Aby proces uogólniania przebiegał skutecznie, dane uczące muszą być reprezentatywne wobec nowych danych, wobec których nastąpi generalizacja. Jest to prawdziwe zarówno dla uczenia z przykładów, jak i dla uczenia z modelu.

Na przykład zbiór krajów użyty przez nas do trenowania modelu liniowego nie był doskonale reprezentatywny, brakuje w nim bowiem kilku państw. Na rysunku 1.21 widzimy, jak będą wyglądać dane po dodaniu tych kilku brakujących krajów.



Rysunek 1.21. Bardziej reprezentatywny zbiór przykładów uczących

Model wyuczony za pomocą tych danych został ukazany w postaci linii ciągkiej, natomiast stary model reprezentuje linia przerywana. Jak widać, dodanie kilku krajów nie tylko znaczco modyfikuje model, ale pozwala nam również uzmysolić sobie, że prosty model liniowy raczej nigdy nie będzie bardzo dokładny. Wygląda na to, że mieszkańcy bardzo bogatych państw nie są szczęśliwi od obywateli średnio zamożnych krajów (wydaje się wręcz, że są bardziej nieszczęśliwi), a z kolei ludność wielu ubogich państw wydaje się znacznie szczęśliwsza od populacji bogatych krajów.

Za pomocą niereprezentatywnego zbioru danych wyuczylismy model, który raczej nie będzie generował dokładnych prognoz, zwłaszcza w przypadku krajów ubogich i bardzo zamożnych.

Jest niezwykle istotne, aby zbiór danych uczących był reprezentatywny wobec generalizowanych przypadków. Często jest to trudniej osiągnąć, niż się wydaje: jeżeli przykład będzie zbyt mały, musisz liczyć się z **zaszumieniem próbki** (ang. *sampling noise*; niereprezentatywne dane pojawiają się tu

w wyniku przypadku), ale nawet olbrzymie zbiory danych mogą nie być reprezentatywne, jeśli użyta zostanie niewłaściwa metoda próbkowania. W tym przypadku mamy do czynienia z **obciążeniem próbkowania** (ang. *sampling bias*).

Przykłady obciążenia próbkowania

Prawdopodobnie najsłynniejszy przykład wpływu obciążenia próbkowania można było zaobserwować podczas wyborów na prezydenta Stanów Zjednoczonych w 1936 roku, w których przeciwko Rooseveltowi stanął Landon: magazyn „Literary Digest” przeprowadził ankietę, wysyłając pocztą ulotki do około 10 milionów obywateli. Zostało odesłanych 2,4 miliona odpowiedzi, dzięki którym z dużą dozą pewności prognozowano, że Landon uzyska 57% głosów. Ostatecznie okazało się, że to Roosevelt zebrał 62% głosów. Problem polegał na użytej metodzie próbkowania:

- Po pierwsze, magazyn „Literary Digest” zebrał adresy osób, którym została wysłana ankieta, z książek telefonicznych, list abonentów i członków klubów itd. We wszystkich tych źródłach dominowały mająte osoby, które częściej głosowały na republikanów (czyli na Landona).
- Po drugie, odpowiedzi odesłano mniej niż 25% osób. Również ten czynnik odpowiada za obciążenie próbkowania, ponieważ nie są brane pod uwagę osoby apolityczne, przeciwnicy magazynu „Literary Digest” i inne kluczowe grupy. Jest to specjalny rodzaj obciążenia próbkowania, zwany **błędem braku odpowiedzi** (ang. *nonresponse bias*).

Weźmy pod uwagę inny przykład: powiedzmy, że chcesz stworzyć system rozpoznający teledyski z muzyką funkową. Jednym ze sposobów przygotowania zbioru danych uczących jest wyszukanie wyników „funk music” w serwisie YouTube i skorzystanie z nich. Zakładamy jednak w tym przypadku, że silnik wyszukiwania zwróci zestaw teledysków stanowiący przekrój wszystkich piosenek funkowych zamieszczonych w serwisie. W rzeczywistości wyniki wyszukiwania są często tendencjonalnie nastawione wobec najpopularniejszych artystów (natomiast jeśli mieszkasz w Brazylii, zostanie wyświetlonych wiele teledysków z nurtu „funk carioca”, który w ogóle nie brzmi jak James Brown). Z drugiej strony, jak inaczej można pozyskać duży zbiór danych uczących?

Dane kiepskiej jakości

Jest oczywiste, że jeśli dane uczące zawierają mnóstwo błędów, elementów odstających i szumu (np. z powodu niskiej jakości pomiarów), to systemowi będzie znacznie trudniej wykryć wzorce, przez co nie osiągnie optymalnej wydajności. Często warto poświęcić czas na oczyszczenie takich danych. Prawda jest taka, że większość analityków danych poświęca na tę czynność wiele czasu. Oto dwa przykłady, w których wymagane byłoby oczyszczenie danych:

- Jeżeli niektóre elementy wyraźnie odstają od reszty przykładów, wystarczy je po prostu odrzucić albo spróbować własnoręcznie poprawić błędy.
- Jeśli niektórym przykładom brakuje kilku cech (np. 5% klientów nie podało wieku), musisz zadecydować, czy należy takie cechy całkowicie ignorować, ignorować te przykłady, wypełnić brakujące wartości (np. korzystając z mediany wieku), wytrenować jeden model przy użyciu tej cechy, a drugi bez niej.

Nieistotne cechy

Zgodnie ze słynnym powiedzeniem: z gipsu tortu nie ulepisz. Twój system będzie w stanie uczyć się jedynie za pomocą danych zawierających wystarczającą liczbę istotnych cech i niezaśmieconych nadmiarem cech nieistotnych. Elementem krytycznym dla sukcesu w projekcie uczenia maszynowego jest wybór dobrego zbioru cech uczących. Proces ten, zwany **inżynierią cech** (ang. *feature engineering*), składa się z następujących etapów:

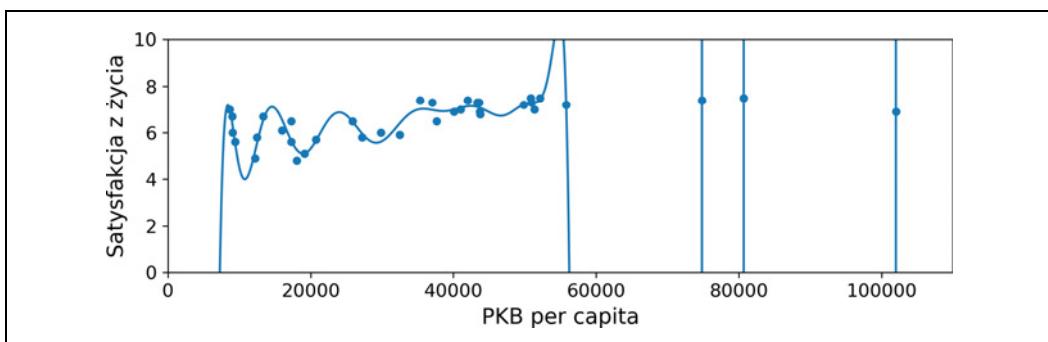
- **dobór cechy** (ang. *feature selection*) — dobór najprzydatniejszych spośród dostępnych cech;
- **odkrywanie cechy** (ang. *feature extraction*) — łączenie ze sobą istniejących cech w celu uzyskania przydatniejszej cechy (jak już wiemy, pomóc nam w tym może algorytm redukcji wymiarowości);
- uzyskiwanie nowych cech z nowych danych.

Skoro już wiemy, jak wyglądają złe dane, przeanalizujmy kwestię nieprawidłowych algorytmów.

Przetrenowanie danych uczących

Załóżmy, że na zagranicznej wycieczce okradł Cię taksówkarz. Być może na Twoje usta ciśnie się stwierdzenie, że wszyscy taksówkarze w danym kraju są złodziejami. Ludzie często mają tendencję do nadmiernego generalizowania i, niestety, maszyny równie łatwo wpadają w tę pułapkę, jeśli nie zachowamy ostrożności. Zjawisko to w terminologii uczenia maszynowego nosi nazwę **przetrenowania** albo **nadmiernego dopasowania** (ang. *overfitting*). Termin ten oznacza, że model sprawdza się w przypadku danych uczących, ale sam proces uogólniania nie sprawuje się zbyt dobrze.

Na rysunku 1.22 widzimy przykład modelu wielomianowego wysokiego stopnia satysfakcji z życia, który bardzo dobrze opisuje dane uczące. Nawet jeśli sprawuje się on znacznie lepiej wobec danych uczących niż zwykły model liniowy, to czy moglibyśmy zawierzyć jego prognozom?



Rysunek 1.22. Przetrenowanie danych uczących

Złożone modele, jak głębokie sieci neuronowe, są w stanie wykrywać subtelne wzorce w danych, ale jeśli zbiór danych uczących jest zaszumiony lub zbyt mały (ryzyko zaszumienia próbki), to model ten prawdopodobnie będzie wykrywał wzorce nie w użytecznych danych, lecz w szumie. Oczywiście szablonów tych nie da się generalizować na nowe próbki. Powiedzmy, że dostarczamy modelowi satysfakcji z życia wiele dodatkowych atrybutów, w tym takich nieprzydatnych jak nazwy państw. W takiej sytuacji złożony model może wykrywać takie wzorce jak np. wskaźnik satysfakcji z życia

przekraczający wartość 7 w krajach mających w nazwie literę „w”: Nowa Zelandia (7,3), Norwegia (7,4), Szwecja (7,2) czy Szwajcaria (7,5). Czy masz pewność, że ta „reguła liter w” dotyczy również Rwandy lub Zimbabwe? Oczywiście wzorzec ten wystąpił w danych uczących zupełnie przypadkowo, ale model nie jest w stanie stwierdzić, czy taki szablon jest rzeczywisty, czy stanowi wynik zaszumienia danych.



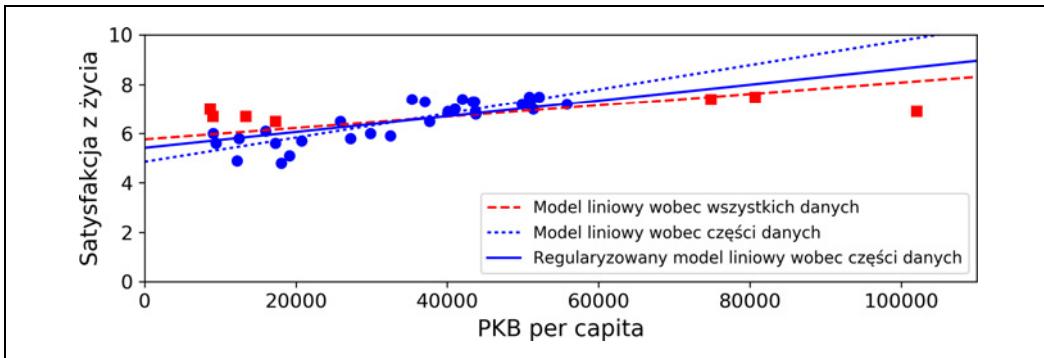
Zjawisko przetrenowania występuje, gdy model jest zbyt skomplikowany w porównaniu do ilości lub zaszumienia danych uczących. W takich przypadkach możliwe są następujące rozwiązania:

- Upraszczać model poprzez wybór modelu zawierającego mniej parametrów (np. modelu liniowego w miejsce modelu wielomianowego), zmniejszenie liczby atrubutów w danych uczących lub ograniczenie modelu.
- Zdobądź większą ilość danych uczących.
- Zmniejszaj zaszumienie danych uczących (np. poprzez usunięcie błędnych danych lub elementów odstających).

Ograniczenie modelu w celu jego uproszczenia i zmniejszenia ryzyka przetrenowania nosi nazwę **regularyzacji** (ang. *regularization*). Przykładowo zdefiniowany przez nas wcześniej model liniowy zawiera dwa parametry: θ_0 i θ_1 . Algorytm uczący uzyskuje w ten sposób dwa **stopnie swobody**, za pomocą których możemy dostosowywać model do danych uczących: mamy wpływ zarówno na wysokość (θ_0), jak i nachylenie (θ_1) prostej. Gdybyśmy wymusili wartość $\theta_1 = 0$, algorytm zostałby ograniczony do jednego stopnia swobody i jego prawidłowe wytrenowanie stałoby się znacznie trudniejsze: moglibyśmy jedynie zmieniać położenie prostej wzdłuż osi y i starać się umieścić jej przebieg jak najbliżej punktów uczących, co w tym przypadku oznaczałoby określenie wartości średniej. Zaiste, bardzo prosty model! Jeżeli pozwolimy algorytmowi na nieznaczne zmiany wartości parametru θ_1 , to w istocie algorytm będzie znajdował się gdzieś pomiędzy jednym a dwoma stopniami swobody. Uzyskamy w ten sposób model prostszy od mającego dwa stopnie swobody, ale bardziej złożony od jednostopniowego. Musimy wyszukiwać odpowiednią równowagę między perfekcyjnym dopasowywaniem danych uczących a zachowaniem odpowiedniej prostoty modelu pozwalającej generalizować wyniki.

Rysunek 1.23 prezentuje trzy modele. Linia kropkowana symbolizuje nasz oryginalny model, wy trenowany na danych z krajów reprezentowanych w postaci kółek (bez danych z krajów oznaczonych kwadratami). Linia kreskowana stanowi drugi model, wy trenowany za pomocą danych z wszystkich krajów (kółek i kwadratów). Linia ciągła natomiast reprezentuje model wy trenowany za pomocą takich samych danych jak w pierwszym przypadku, ale przy wprowadzonej regularyzacji. Widzimy, że w wyniku regularyzacji funkcja ma trochę mniejsze nachylenie: model ten jest nieco mniej dopasowany do danych uczących (kółek) w stosunku do pierwszego modelu, ale za to sprawuje się lepiej w procesie uogólniania nowych, nieznanych wcześniej przykładów (kwadratów).

Stopień regularyzacji przeprowadzanej na etapie nauki możemy kontrolować za pomocą **hiperparametrów** (ang. *hyperparameters*). Najprościej mówiąc, hiperparametrami nazywamy parametry algorytmu uczącego (nie całego modelu). Nie są one modyfikowane przez sam algorytm uczący — należy je wyznaczyć tuż przed rozpoczęciem procesu uczenia i w jego trakcie ich wartości pozostają



Rysunek 1.23. Regularyzacja zmniejsza ryzyko przetrenowania

niezmienne. Jeśli wyznaczysz bardzo dużą wartość hiperparametru regularyzacji, uzyskana funkcja będzie niemal stała (będzie miała niemal zerowe nachylenie); algorytm uczący prawie na pewno nie ulegnie przetrenowaniu, ale jednocześnie trudniej będzie mu znaleźć prawidłowe rozwiązanie. Strojenie hiperparametrów stanowi istotną część tworzenia systemu uczenia maszynowego (w następnym rozdziale przekonasz się o tym na własnej skórze).

Niedotrenowanie danych uczących

Jak łatwo się domyślić, **niedotrenowanie** (ang. *underfitting*) stanowi przeciwieństwo przetrenowania: występuje ono wtedy, gdy model jest zbyt prosty, aby wyuczyć się struktur danych uczących. Na przykład liniowy model satysfakcji z życia jest podatny na niedotrenowanie; rzeczywistość jest po prostu bardziej skomplikowana od modelu, zatem jego prognozy nie będą dokładne, nawet w przypadku danych uczących.

Oto główne sposoby rozwiązania tego problemu:

- Wybieraj potężniejszy model wykorzystujący większą liczbę parametrów.
- Dołączaj większą liczbę cech do algorytmu uczącego (inżyniera cech).
- Zmniejsz ograniczenia modelu (np. redukuj hiperparametr regularyzacji).

Podsumowanie

Wiesz coraz więcej na temat uczenia maszynowego. Wspomniałem jednak o tak wielu pojęciach, że możesz się pogubić, dlatego zatrzymajmy się na chwilę i sprawdźmy, co już wiesz:

- Uczenie maszynowe polega na ciągłym usprawnianiu wykonywania zadań przez urządzenie poprzez jego uczenie się z danych, a nie jawne dopisywanie nowych reguł do kodu.
- Istnieje wiele różnych metod uczenia maszynowego: nadzorowane, nienadzorowane, wsadowe, przyrostowe, z przykładów lub z modelu.
- W projekcie uczenia maszynowego gromadzimy dane w zbiorze uczącym, który następnie dostarczamy algorytmowi uczącemu. Jeśli wykorzystujemy technikę uczenia z modelu, pewne parametry zostają dostrojone do zbioru danych uczących (np. w celu uzyskiwania dokładnych prognoz wobec samych danych uczących), dzięki czemu model ten może również dobrze przewidywać

wyniki wobec nowych przykładów. Z kolei w metodach uczenia z przykładów próbki zostają po prostu wyuczone „na pamięć”, a algorytm przeprowadza uogólnianie wyników na nowe przykłady za pomocą miary podobieństwa poprzez porównywanie ich z przykładami wyuczonymi.

- System nie będzie dobrze spełniał swojego zadania, jeżeli zbiór danych testowych jest zbyt mały, same dane są niereprezentatywne, zaszumione lub zaśmiecone nieistotnymi cechami (z gipsu tortu nie ulepisz). Model nie może być również zbyt prosty (niedotrenowanie) ani zbyt skomplikowany (przetrenowanie).

Musimy omówić jeszcze jedno istotne zagadnienie: po wyuczeniu modelu nie wystarczy sama nadzieja na skuteczne uogólnianie nowych próbek. Chcesz ocenić wydajność modelu i w razie potrzeby dostroić go do swoich potrzeb. Dowiesz się teraz, jak tego dokonać.

Testowanie i ocenianie

Jednym sposobem, by sprawdzić, jak dobrze model generalizuje wyniki, jest wypróbowanie go na nowych danych. Możemy na przykład zaimplementować go w środowisku produkcyjnym i monitorować jego wydajność. Jest to skuteczne rozwiązanie, ale jeśli model okaże się beznadziejny, użytkownicy będą narzekać — a tego lepiej unikać.

Lepszym pomysłem jest podział danych na dwa zestawy: **zbiór uczący** (ang. *training set*) i **zbiór testowy** (ang. *test set*). Zgodnie z tymi nazwami model trenujemy za pomocą zbioru uczącego, a sprawdzamy przy użyciu zbioru testowego. Współczynnik błędu uzyskiwany dla nowych przykładów nosi nazwę **błędu uogólniania** (lub **błędu generalizacji**), a dzięki zbiorowi testowemu możemy oszacować jego wartość. Parametr ten mówi nam również, jak model będzie się spisywał wobec nieznanych danych.

Jeśli wartość błędu uczenia jest niewielka (tzn. model rzadko się myli wobec zbioru uczącego), ale błąd uogólniania jest duży, oznacza to, że model jest przetrenowany.



Zazwyczaj na zbiór uczący składa się 80% danych, a pozostałe 20% *przechowujemy* w celu testowania. Zależy to jednak od rozmiaru zestawu danych: jeżeli składa się on z 10 milionów przykładów, to pozostawienie 1% oznacza zestaw danych testowych o rozmiarze 100 000 przykładów, co prawdopodobnie w zupełności gwarantuje dobre oszacowanie błędu uogólniania.

Strojenie hiperparametrów i dobór modelu

Ocena modelu nie jest wcale taka trudna: wystarczy użyć zestawu testowego. Założymy jednak, że wahamy się przy wyborze jednego z dwóch modeli (np. modelu liniowego i modelu wielomianowego). W jaki sposób mamy zdecydować? Jednym z rozwiązań jest wyuczenie obydwu modeli i sprawdzenie, jak sobie radzą z uogólnianiem wobec zbioru testowego.

Przymijmy teraz, że model liniowy radzi sobie lepiej z uogólnianiem, ale chcesz wprowadzić regularyzację w celu uniknięcia przetrenowania. Rodzi się w tym momencie pytanie: w jaki sposób dobrać wartość hiperparametru regularyzacji? Możesz na przykład wytrenować 100 różnych modeli, z których każdy będzie miał inną wartość tego hiperparametru. Założymy, że znajdziesz optymalną wartość

tego hiperparametru, dającą model o najniższym błędzie generalizacji (np. rzędu 5%). Wprowadzasz ten model do środowiska produkcyjnego, ale niestety sprawuje się on poniżej oczekiwania i generuje 15% błędów. Co się właściwie stało?

Problem polega na tym, że wielokrotnie mierzyłaś/mierzyleś błąd uogólniania wobec zbioru testowego i dostosowałaś/dostosowałeś go *do tego określonego zestawu danych*. Oznacza to, że teraz prawdopodobnie model ten nie będzie sobie radził zbyt dobrze z nowymi próbками.

Powszechnie stosowanym rozwiązyaniem tego problemu jest tzw. **sprawdzian na odłożonych danych** (ang. *holdout validation*): wystarczy odłożyć część zestawu danych uczących w celu zweryfikowania kilku różnych modeli i wybrania najlepszego z nich. Taki nowy, odłożony zestaw nazywamy **zbiorem walidacyjnym** (ang. *validation set*) albo **zbiorem rozwojowym** (ang. *development set* lub w skrócie *dev set*). Mówiąc dokładniej, trenujemy wiele modeli mających różne wartości hiperparametrów za pomocą zredukowanego zbioru uczącego (tzn. pełnego zestawu danych uczących minus zestaw walidacyjny) i dobieramy model najlepiej sprawujący się wobec zbioru walidacyjnego. Po zakończeniu sprawdzianu na odłożonych danych trenujemy najlepszy model na pełnym zestawie danych uczących (włącznie ze zbiorem walidacyjnym), co pozwala uzyskać model ostateczny. Na koniec wystarczy przeprowadzić ostatni sprawdzian wobec zbioru testowego, aby oszacować wartość błędu uogólniania.

Rozwiązywanie to zazwyczaj sprawdza się bardzo dobrze. Jeżeli jednak zestaw walidacyjny jest zbyt mały, to weryfikacja modelu może się okazać nieprecyzyjna: możesz przez pomyłkę wybrać nieoptimalny model. Z drugiej strony, jeżeli zbiór walidacyjny będzie zbyt duży, to pozostały zestaw danych uczących okaże się zbyt mały w stosunku do pełnego zbioru uczącego. Dlaczego stanowi to problem? Skoro model ostateczny będzie trenowany na pełnym zestawie danych uczących, to nie jest dobrze porównywać różne modele wyuczone na znacznie mniejszym zbiorze uczącym. Przypominamy, aby to wybór najszybszego sprintera to uczestnictwa w maratonie. Jednym ze sposobów uniknięcia tego problemu jest wielokrotne powtarzanie techniki zwanej **sprawdzianem krzyżowym** lub **krosvalidacją** (ang. *cross-validation*) za pomocą wielu małych zbiorów walidacyjnych. Każdy model zostaje raz oceniony na każdy zbiór walidacyjny po jego wyuczeniu za pomocą pozostałych danych. Poprzez uśrednienie wszystkich ocen modelu uzyskujesz znacznie dokładniejszą miarę jego wydajności. Rozwiążanie to ma jednak wadę: czas uczenia zostaje zwielokrotniony przez liczbę zbiorów walidacyjnych.

Niezgodność danych

W niektórych przypadkach łatwo jest uzyskać duże ilości danych uczących, prawdopodobnie nie będą one jednak idealnie odzwierciedlać danych wykorzystywanych w środowisku produkcyjnym. Założmy na przykład, że korzystasz z aplikacji mobilnej wykonującej zdjęcia kwiatów i automatycznie określającej ich gatunek. Możesz z łatwością pobrać miliony zdjęć kwiatów z internetu, nie będą one jednak stanowić doskonałego odwzorowania zdjęć wykonywanych przez tę aplikację. Może dysponujesz zaledwie 10 000 reprezentatywnych zdjęć (np. wykonanych za pomocą aplikacji). W takim przypadku należy koniecznie pamiętać o tym, aby zestawy walidacyjny i testowy zawierały jak największej reprezentatywnych danych, których można będzie spodziewać się w środowisku produkcyjnym, dlatego powinny składać się wyłącznie ze zdjęć wykonanych za pomocą naszej aplikacji: możesz je przetasować i umieścić połowę w zbiorze walidacyjnym, a drugą w zbiorze testowym (i zagwarantować, że w obydwu zestawach nie pojawią się żadne duplikaty lub prawie duplikaty). Jeżeli jednak po wyuczeniu modelu za pomocą zdjęć pobranych z internetu okaże się, że skuteczność modelu

względem zestawu walidacyjnego będzie niezadowalająca, nie będziesz wiedzieć, czy wynika to z przetrenowania modelu na danych uczących czy z niezgodności zdjęć pobranych z sieci i wykonanych za pomocą aplikacji. Jednym z rozwiązań jest odłożenie części danych uczących (zdjęć pobranych z internetu) w kolejnym zbiorze, który Andrew Ng określił jako **zestaw ucząco-rozwojowy** (ang. *train-dev-set*). Po wytrenowaniu modelu (za pomocą zestawu uczącego, a nie ucząco-rozwojowego) możesz ocenić jego wydajność na zbiorze ucząco-rozwojowym. Jeżeli uzyskane wyniki są dobre, to model nie uległ przetrenowaniu. Jeżeli okazuje się niezbyt skuteczny względem zestawu walidacyjnego, to przyczyną problemu musi być niezgodność danych. Możesz spróbować poradzić sobie z tym problemem, przetwarzając wstępnie zdjęcia pochodzące z internetu tak, aby przypominały dane uzyskane za pomocą aplikacji mobilnej, a następnie ponownie trenując model. I odwrotnie, jeśli model nie radzi sobie ze zbiorem ucząco-rozwojowym, to nastąpiło przetrenowanie tego modelu, dlatego należy go uprościć albo regularyzować, uzyskać więcej danych uczących i je oczyścić.

Twierdzenie o nieistnieniu darmowych obiadów

Model stanowi uproszczoną postać obserwacji. Uproszczenia wynikają z konieczności unikania niepotrzebnych szczegółów, które nie ułatwiają procesu uogólniania. Podczas dobierania i odrzucania danych musimy przyjmować **założenia**. Na przykład w modelu liniowym zakładamy, że dane są, nomen omen, liniowe, a odległość pomiędzy przykładami i wykresem funkcji stanowi wyłącznie szum, który możemy bezpiecznie zignorować.

W słynnej publikacji z 1996 roku¹⁵ (<https://www.mitpressjournals.org/doi/abs/10.1162/neco.1996.8.7.1341>) David Wolpert udowodnił, że jeśli nie przyjmiemy jakichkolwiek założeń dotyczących danych, to okaże się, że żaden model nie będzie lepszy od pozostałych. Jest to tak zwane **twierdzenie o nieistnieniu darmowych obiadów** (ang. *No Free Lunch Theorem* — NFL). Dla pewnych zbiorów danych najlepiej nadaje się model liniowy, natomiast dla innych — sieci neuronowe. Nie istnieje żaden model, który z założenia będzie działał lepiej (stąd nazwa twierdzenia). Jedynie poprzez ocenę działania każdego modelu możemy przekonać się, który z nich będzie sprawiał się najlepiej. Jest to niewykonalne, dlatego w praktyce przyjmujemy rozsądne założenia dotyczące danych i oceniamy działanie tylko kilku rozsądnie dobranych modeli. Na przykład wobec prostych zadań możemy ocenić działanie modeli liniowych różniących się stopniem regularyzacji, a bardziej skomplikowane problemy możemy przetestować przy użyciu sieci neuronowych.

Ćwiczenia

W tym rozdziale omówiłem część najważniejszych koncepcji opisujących dziedzinę uczenia maszynowego. W kolejnych rozdziałach skoncentrujemy się na szczegółach i pisaniu właściwego kodu, zanim jednak przejdziesz dalej, upewnij się, że jesteś w stanie odpowiedzieć na następujące pytania i polecenia:

1. Podaj definicję uczenia maszynowego.
2. Wymień cztery rodzaje problemów, z których rozwiązaniem najlepiej sobie radzi proces uczenia maszynowego.

¹⁵ David Wolpert, *The Lack of A Priori Distinctions Between Learning Algorithms*, „Neural Computation” 8, nr 7 (1996), s. 1341 – 1390.

- 3.** Czym jest oznakowany zbiór danych uczących?
- 4.** Jakie są dwa najczęstsze zastosowania uczenia nadzorowanego?
- 5.** Wymień cztery najpowszechniejsze zastosowania uczenia nienadzorowanego.
- 6.** Jakiego rodzaju algorytmu uczenia maszynowego użyłabyś/użyłbyś w robocie przeznaczonym do poruszania się po nieznanym terenie?
- 7.** Jakiego rodzaju algorytmu uczenia maszynowego użyłabyś/użyłbyś do rozdzielenia klientów na kilka różnych grup?
- 8.** Czy problem wykrywania spamu stanowi część mechanizmu uczenia nadzorowanego czy nienadzorowanego?
- 9.** Czym jest system uczenia przyrostowego?
- 10.** Czym jest uczenie pozakorowe?
- 11.** W jakim algorytmie uczenia maszynowego jest wymagana miara podobieństwa do uzyskiwania prognoz?
- 12.** Wyjaśnij różnicę pomiędzy parametrem modelu a hiperparametrem algorytmu uczącego.
- 13.** Czego poszukują algorytmy uczenia z modelu? Z jakiej strategii najczęściej korzystają? W jaki sposób przeprowadzają prognozy?
- 14.** Wymień cztery główne problemy związane z uczeniem maszynowym.
- 15.** Z czym mamy do czynienia, jeżeli model sprawuje się znakomicie wobec danych uczących, ale nie radzi sobie z uogólnianiem wobec nowych próbek? Wymień trzy możliwe rozwiązania.
- 16.** Czym jest zbiór testowy i dlaczego powinniśmy z niego korzystać?
- 17.** Do czego służy zbiór walidacyjny?
- 18.** Czym jest zestaw ucząco-rozwojowy? Kiedy jest potrzebny? W jaki sposób korzystamy z niego?
- 19.** Co nam grozi w przypadku strojenia hiperparametru wobec zbioru testowego?

Odpowiedzi znajdziesz w dodatku A.

Nasz pierwszy projekt uczenia maszynowego

W tym rozdziale stworzymy od początku do końca przykładowy projekt uczenia maszynowego — będziemy udawać, że jesteśmy świeżo zatrudnionymi analitykami danych w agencji handlu nieruchomości¹. Aby nam się udało, musimy wykonać następujące czynności:

1. Przeanalizować całokształt czekającego nas zadania.
2. Uzyskać dane.
3. Odkryć i zwizualizować dane w celu rozpoznania wzorców i dodatkowych informacji.
4. Przygotować dane pod względem algorytmów uczenia maszynowego.
5. Wybrać i wyuczyć model.
6. Dostroić model.
7. Zaprezentować rozwiązanie.
8. Uruchomić, monitorować i utrzymywać system.

Praca z rzeczywistymi danymi

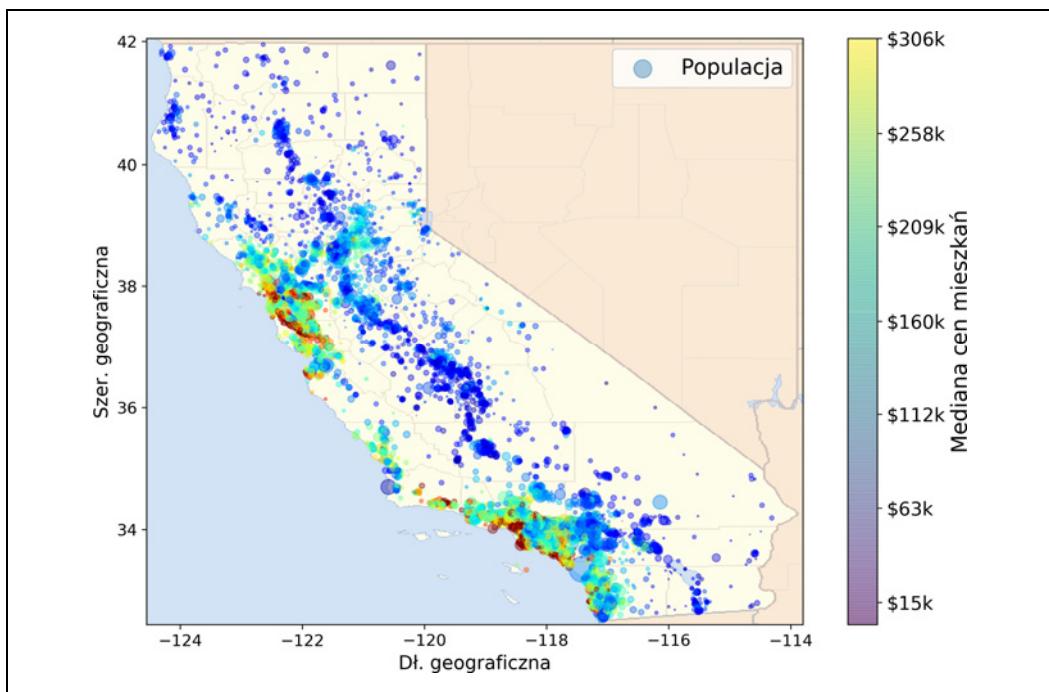
Podczas poznawania zagadnień uczenia maszynowego najlepiej jest eksperymentować na rzeczywistych, a nie sztucznych danych. Na szczęście są dostępne tysiące otwartych zbiorów danych, opisujące chyba wszelkie możliwe dziedziny życia. Poniżej wymieniam kilka miejsc, w których możesz szukać interesujących Cię danych:

- popularne, otwarte repozytoria danych:
 - repozytorium uczenia maszynowego Uniwersytetu Kalifornijskiego w Irvine (<http://archive.ics.uci.edu/ml/index.php>),
 - zbiory danych Kaggle (<https://www.kaggle.com/datasets>),
 - zbiory danych w usłudze Google AWS (<https://registry.opendata.aws/>),

¹ Projekt ten jest fikcyjny; jego zadaniem jest ukazanie poszczególnych etapów przygotowywania projektu uczenia maszynowego, a nie przekazywanie wiedzy na temat obrotu nieruchomościami.

- metaportale (zawierają one listy otwartych repozytoriów danych):
 - Data Portals (<http://dataportals.org/>),
 - OpenDataMonitor (<http://opendatamonitor.eu/>),
 - Quandl (<http://quandl.com/>),
- inne serwisy zawierające listy popularnych otwartych repozytoriów danych:
 - lista zbiorów danych dotyczących uczenia maszynowego na Wikipedii (https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research),
 - Quora.com (<https://www.quora.com/Where-can-I-find-large-datasets-open-to-the-public>),
 - temat na forum Reddit (<https://www.reddit.com/r/datasets/>).

Na użytku tego rozdziału wykorzystamy zestaw danych California Housing Prices (w wolnym tłumaczeniu „ceny domów w Kalifornii”), stanowiący część repozytorium StatLib² (rysunek 2.1). Dane te zostały opracowane na podstawie spisu ludności Kalifornii z 1990 roku. Nie są one już zbyt aktualne (w tamtym czasie można było nabyć dom w Bay Area za całkiem rozsądne pieniądze), ale pod wieloma względami mają one charakter edukacyjny, dlatego będziemy udawać, że są całkiem świeże. W celach dydaktycznych dodałem również atrybut kategorialny i usunąłem kilka cech.



Rysunek 2.1. Ceny domów w Kalifornii

² Pierwotnie zbiór tych danych pojawił się w artykule R. Kelley'a Pace'a i Ronald'a Barry'ego *Sparse Spatial Autoregressions*, „Statistics & Probability Letters 33”, nr 3 (1997), s. 291 – 297.

Przeanalizuj całokształt projektu

Witaj w Korporacji Inteligentne Nieruchomości! Twoim pierwszym zadaniem jest wykorzystanie danych spisu ludności Kalifornii do stworzenia modelu cen mieszkań w tym stanie. Dane te zawierają takie informacje jak wielkość populacji, mediana dochodów, mediana cen mieszkań — wszystko rozzielone pomiędzy poszczególne grupy bloków Kalifornii. Grupami bloków nazywamy najmniejsze jednostki terytorialne, dla których są publikowane próbki danych ze spisu ludności (zazwyczaj na typową grupę bloków składa się od 600 do 3000 osób). Dla uproszczenia będziemy nazywać je „dystryktami”.

Twój model powinien uczyć się z tych danych i przy użyciu pozostałych metryk być w stanie przewidywać medianę cen mieszkań w dowolnym dystrykcie.



Skoro jesteśmy poukładanymi analitykami danych, naszą pierwszą czynnością powinno być przygotowanie listy kontrolnej projektu. Na początek możesz skorzystać z listy umieszczonej w dodatku B; powinna się ona nadawać do większości projektów uczenia maszynowego, nie zapomnij jednak dostosować jej do własnych potrzeb. W niniejszym rozdziale zajmiemy się wieloma elementami umieszczonymi na tej liście, ale też pominiemy niektóre, gdyż albo nie wymagają objaśnienia, albo zostaną omówione w następnych rozdziałach.

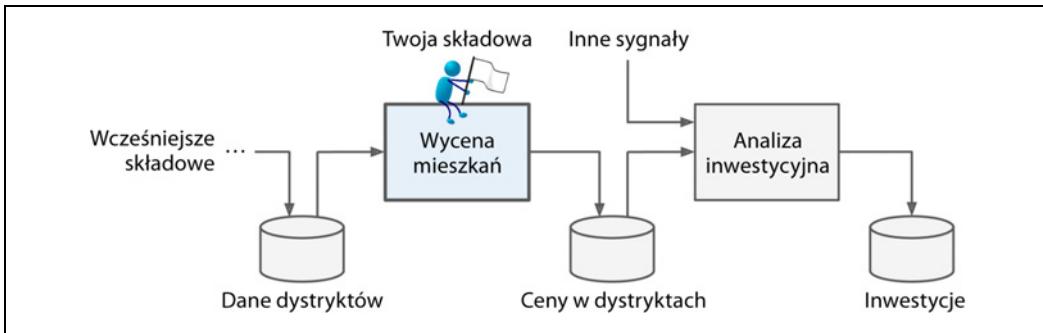
Określ zakres problemu

Pierwszym pytaniem, jakie należałoby zadać przełożonemu, byłoby: „Jaki jest dokładny cel biznesowy?”; utworzenie modelu prawdopodobnie nie jest ostatecznym celem samym w sobie. W jaki sposób firma zamierza skorzystać i zarobić na tym modelu? Znajomość celu jest istotna, ponieważ pomaga określić zakres problemu, dobrać odpowiednie algorytmy oraz miarę wydajności służącą do oceny wydajności modelu, a także pozwala oszacować wysiłek potrzebny do zoptymalizowania działania modelu.

Twój szef odpowiada, że wynik tego modelu (prognoza mediany cen mieszkań w danym dystrykcie) zostanie przekazany do innego systemu uczenia maszynowego (rysunek 2.2) wraz z wieloma innymi sygnałami³. Dzięki takiemu przepływowemu systemowi możliwe staje się określenie, czy warto inwestować na danym obszarze. Wszystko musi być dopięte na ostatni guzik, gdyż uzyskiwane rezultaty będą miały bezpośredni wpływ na obroty.

Następnie należałoby zapytać szefa, jak wygląda obecne rozwiązanie (jeśli w ogóle jakieś istnieje). Często możesz uzyskać w ten sposób dostęp do referencyjnej wydajności, a także podpowiedzi ułatwiające rozwiązanie problemu. Twój szef odpowiada, że obecnie ceny domów są szacowane ręcznie przez zespół ekspertów: gromadzą oni aktualne informacje o dystryktach, a gdy nie są w stanie wyliczyć mediany cen mieszkań, korzystają ze skomplikowanych reguł.

³ Zgodnie z teorią informacji, którą Claude Shannon, zdefiniował w firmie Bell Labs w celu poprawy jakości telekomunikacji, **sygnałem** nazywamy informację przekazywaną systemowi uczenia maszynowego. Zgodnie z tą teorią zależy nam na uzyskaniu wysokiej wartości stosunku sygnału do szumu.



Rysunek 2.2. Potok uczenia maszynowego używany w branży nieruchomości

Jest to bardzo kosztowna i czasochłonna czynność, a uzyskane tą metodą oszacowania nie są wcale takie dokładne; często w sytuacjach, gdy analitykom udaje się wyliczyć medianę cen mieszkań, okazuje się, że pomyliły się w oszacowaniach nawet o ponad 20%. Dlatego zarząd firmy uważa, że warto byłoby wyuczyć model przewidujący medianę cen mieszkań w danym dystrykcie za pomocą innych danych opisujących dany dystrykt. Znakomitym źródłem potrzebnych informacji okazuje się spis ludności, ponieważ możemy w nim znaleźć medianę cen mieszkań wyliczoną dla tysięcy dystryktów, a także wiele innych danych.

Potoki

Potokiem (ang. *pipeline*) danych nazywamy sekwencję **składowych** przetwarzanych danych. Potoki są bardzo popularne w systemach uczenia maszynowego, ponieważ manipulujemy w nich olbrzymią ilością danych oraz przeprowadzamy na nich wiele przekształceń.

Składowe są zazwyczaj przetwarzane asynchronicznie. Każda składowa pobiera znaczną ilość danych, przetwarza je i umieszcza wyniki w innym magazynie informacji. Po pewnym czasie te przetworzone wyniki są pobierane przez następną składową potoku, w której zostają znowu przetworzone i umieszczone w kolejnym magazynie danych. Każda składowa jest w znacznej mierze uniezależniona od pozostałych; jedynym interfejsem pomiędzy poszczególnymi składowymi jest magazyn danych. W ten sposób uzyskujemy prosty do zrozumienia system (za pomocą wykresów graficznych), a poszczególne zespoły mogą skoncentrować się na własnych składowych. Poza tym awaria danej składowej nie wpływa na pracę następnych składowych (przynajmniej przez pewien czas), gdyż mogą one korzystać z ostatniego prawidłowego wyniku wygenerowanego przez uszkodzoną składową. Dzięki temu taka architektura jest dość odporna.

Z drugiej strony, przy braku odpowiednich mechanizmów monitorowania uszkodzona składowa może przez dłuższy czas niezauważona dostarczać nieprawidłowe wyniki. Dane stają się nieaktualne i spada ogólna wydajność systemu.

Po uzyskaniu tych informacji możesz zająć się projektowaniem systemu. Najpierw zdefiniujmy problem: rozwiążemy go za pomocą uczenia nienadzorowanego, nadzorowanego czy może przez wzmacnianie? Jest to zadanie klasyfikacji, regresji czy jeszcze jakieś inne? Powinniśmy korzystać z technik uczenia przyrostowego czy wsadowego? Zanim przejdziemy dalej, przerwij na chwilę czytanie i postaraj się samodzielnie odpowiedzieć na te pytania.

Masz już odpowiedzi? Zastanówmy się: z pewnością mamy tu do czynienia z klasycznym zadaniem uczenia nadzorowanego, ponieważ będziemy korzystać z **oznakowanych** przykładów uczących (każdy

przykład ma zdefiniowany od razu oczekiwany wynik, np. medianę cen mieszkań w dystrykcie). Jest to również klasyczne zadanie regresyjne, ponieważ musimy przewidzieć jakąś wartość. Mówiąc dokładniej, stajemy przed problemem **regresji wielorakiej** (ang. *multiple regression*), gdyż do prognozowania wyniku nasz system wykorzysta wiele cech (populację dystryktu, medianę dochodów itd.). Możemy także traktować ten problem jako zadanie **regresji jednoczynnikowej** (ang. *univariate regression*), ponieważ dla każdego dystryktu staramy się przewidzieć tylko pojedynczą wartość. Gdybyśmy próbowali prognozować wiele wartości dla każdego dystryktu, mielibyśmy do czynienia z **regresją wieloczynnikową** (ang. *multivariate regression*). Na koniec zwróćmy uwagę, że dane nie będą dostarczane w ciągły sposób, nie będziemy musieli dynamicznie dostosowywać systemu do zmieniających się danych, a same dane są wystarczająco małe, aby zmieścić się w pamięci, dlatego powinna nam wystarczyć zwykła, wsadowa metoda uczenia.



Gdyby dane miały jednak ogromne rozmiary, mogłabyś/móglibyś rozdzielić czynności uczenia wsadowego pomiędzy kilka serwerów (przy użyciu techniki MapReduce) albo zastąpić je po prostu mechanizmem uczenia przyrostowego.

Wybierz metrykę wydajności

Kolejnym etapem jest dobór metryki wydajności. W przypadku zagadnień regresyjnych klasyczną miarą wydajności jest **pierwiastek błędu średniokwadratowego** (ang. *Root Mean Square Error — RMSE*). Dowiadujemy się dzięki niemu, w jakim stopniu model myli się w przewidywaniach — wraz ze wzrostem wartości błędu rośnie również waga tej metryki. Równanie 2.1 przedstawia wzór matematyczny używany do wyliczania błędu RMSE.

Równanie 2.1. Pierwiastek błędu średniokwadratowego (RMSE)

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

Notacje

Powyższy wzór zawiera kilka najpowszechniejszych notacji używanych w uczeniu maszynowym, z których będziemy korzystać w dalszej części książki:

- Wartość m określa liczbę elementów zbioru uczącego, wobec których będziemy mierzyć błąd RMSE,
 - przykładowo, jeśli chcesz obliczyć błąd RMSE dla zbioru walidacyjnego 2000 dystryktów, to $m = 2000$.
- Wartość $\mathbf{x}^{(i)}$ stanowi wektor wartości wszystkich cech (z pominięciem etykiet) i -tego przykładu, natomiast $y^{(i)}$ stanowi etykietę tejże próbki (czyli pożądaną wartość wyniku dla danego przykładu),
 - na przykład, jeśli pierwszy dystrykt w zbiorze danych znajduje się na długości geograficznej $-118,29^\circ\text{W}$ i szerokości geograficznej $33,91^\circ\text{N}$, a do tego wiemy, że jego populacja wynosi 1416 mieszkańców przy średnim dochodzie 38 372 dolarów, natomiast mediana ceny mieszkań na tym obszarze wynosi 156 400 dolarów (na razie ignorujemy pozostałe cechy), to możemy zapisać te dane w następujący sposób:

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

i
 $y^{(1)} = 156400.$

- Wartość X oznacza macierz zawierającą wartości wszystkich cech (z pominięciem etykiet) wszystkich przykładów składających się na zbiór danych. Dla każdego przykładu przeznaczony jest jeden wiersz, natomiast i -ty wiersz stanowi transpozycję wektora $\mathbf{x}^{(i)}$; zwróć uwagę na zapis $(\mathbf{x}^{(i)})^T$ ⁴,
 - na przykład zgodnie z podanym powyżej opisem pierwszego dystryktu macierz X będzie wyglądała następująco:

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

- Parametr h jest funkcją prognozującą naszego systemu (bywa ona zwana także **hipotezą**). Po wprowadzeniu wartości wektora cech $\mathbf{x}^{(i)}$ do systemu zostaje wygenerowana prognozowana wartość $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$ dla tego przykładu (symbol \hat{y} nazywamy „igrekiem z daszkiem”),
 - przykładowo, jeśli Twój system wyliczy, że mediana cen mieszkań w pierwszym dystrykcie wynosi 158 400 dolarów, to $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158400$. Wartość błędu predykcyjnego dla tego dystryktu to $\hat{y}^{(1)} - y^{(1)} = 2000$.
- Zapis $RMSE(X, h)$ oznacza funkcję kosztu mierzoną dla zbioru uczącego za pomocą hipotezy h .

Wartości skalarne (np. m lub $y^{(i)}$), a także nazwy funkcji (np. h) zapisujemy małymi literami i kursywą, symbole pogrubione (np. $\mathbf{x}^{(i)}$) oznaczają wektory, natomiast duże, pogrubione litery (np. X) są zarezerwowane dla macierzy.

Pomimo że preferowaną metryką wydajności w zadaniach regresyjnych jest pomiar błędu RMSE, w pewnych sytuacjach lepiej sprawdza się inna funkcja. Założymy na przykład, że w zestawie danych znajduje się wiele dystryktów odstających od reszty. W takim przypadku warto byłoby wyliczyć **średni absolutny błąd** (ang. *Mean Absolute Error* — MAE; zwany jest także średnim odchyleniem bezwzględnym, ang. *Average Absolute Deviation*) (równanie 2.2).

Równanie 2.2. Średni absolutny błąd (MAE)

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

⁴ Przypominam, że operator transpozycji przekształca wiersze na kolumny i odwrotnie.

Obydwie miary (RMSE i MAE) pozwalają na obliczanie odległości pomiędzy dwoma wektorami: wektorem prognoz i wektorem wartości docelowych. Dostępne są różne miary odległości (tzw. **normy**):

- Obliczenie pierwiastka błędu średniokwadratowego (RMSE) wiąże się z **normą euklidesową** — stanowi ona doskonale nam znaną notację odległości. Zwana jest ona także **normą ℓ_2** i jest zapisywana w postaci wyrażenia $\|\cdot\|_2$ (lub po prostu $\|\cdot\|$).
- Obliczenie średniego absolutnego błędu (MAE) wiąże się z **normą ℓ_1** , zapisywana jako wyrażenie $\|\cdot\|_1$. Jest ona nazywana również **normą taksówkową**, **miejską** oraz **Manhattan**, ponieważ mierzy ona odległość pomiędzy dwoma punktami w mieście, jedynie poruszając się wzduż ortogonalnych bloków miasta.
- W ogólniejszym ujęciu **normę ℓ_k** wektora v zawierającego n elementów definiujemy następująco: $\|v\|_k = \left(|v_0|^k + |v_1|^k + \dots + |v_n|^k \right)^{\frac{1}{k}}$. Norma ℓ_0 podaje liczbę niezerowych elementów w danym wektorze, natomiast ℓ_∞ wyznacza w nim maksymalną wartość bezwzględną.
- Im wyższy indeks normy, tym bardziej skupia się ona na dużych wartościach i ignoruje małe wartości. Z tego właśnie powodu metryka RMSE jest bardziej wyczulona na elementy odstające niż miara MAE. Jednakże jeśli liczba tych elementów wykładniczo maleje (np. w przypadku krzywej dzwonowej), pomiary RMSE znakomicie się sprawdzają i generalnie jest zalecane używanie tej miary.

Sprawdź założenia

Przed przystąpieniem do tworzenia modelu zawsze warto sporządzić listę dotychczasowych założeń i je zweryfikować (przez Ciebie lub kogoś innego) — może Ci to pomóc już na wczesnym etapie w wychwyceniu jakichś problemów. Na przykład prognozowane przez Twój system ceny w poszczególnych dystryktach będą przekazywane do następnej składowej potoku, a Ty zakładasz, że będą one tam używane w zdefiniowanym przez nas formacie. A jeśli okaże się, że ceny te będą kategoryzowane (np. „tanie”, „średnie” i „drogie”) i dopiero tego typu wartości będą wykorzystywane? W takim wypadku uzyskanie dokładnej ceny mija się zupełnie z celem; Twój system musi jedynie prawidłowo przewidywać kategorie. Skoro tak, zagadnienie, którym się zajmujesz, powinno zostać oznaczone jako problem klasyfikacyjny, a nie regresyjny. Raczej nie chciałbyś/chciałbyś tego odkryć dopiero po kilkumiesięcznej pracy nad systemem regresyjnym.

Na szczęście po rozmowie z szefostwem i pozostałymi zespołami masz pewność, że potrzebne będą rzeczywiste ceny, a nie kategorie. Znakomicie! Wszystko przygotowane, masz zielone światło i możesz zabrać się za pisanie kodu!

Zdobądź dane

Czas pobrudzić sobie ręce. Bez wahania uruchom laptopa i w miarę zapoznawania się z treścią książki samodzielnie sprawdzaj omawiane przykłady, zawarte w notatnikach Jupyter. Wszelkie potrzebne materiały znajdziesz pod adresem <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Stwórz przestrzeń roboczą

W pierwszej kolejności należy zainstalować środowisko Python. Prawdopodobnie masz już ten etap za sobą. Jeśli nie, wszelkie potrzebne pliki znajdziesz na stronie <https://www.python.org/>⁵.

Teraz musimy stworzyć katalog roboczy, w którym będziemy przechowywać kod projektu oraz zbiory danych. Otwórz terminal i wpisz następujące polecenia (po znaku zachęty \$):

```
$ export ML_PATH="$HOME/um" # Jeśli chcesz, możesz zmienić nazwę katalogu  
$ mkdir -p $ML_PATH
```

Potrzebnych nam będzie kilka modułów środowiska Python: Jupyter, NumPy, pandas, Matplotlib i Scikit-Learn. Jeśli już masz je zainstalowane, możesz spokojnie przejść do punktu „Pobierz dane”. W przeciwnym razie istnieje wiele różnych sposobów ich instalacji (wraz z niezbędnymi zależnościami). Możesz użyć systemowego menedżera pakietów (np. *apt-get* w dystrybucji Ubuntu albo *MacPorts* lub *Homebrew* w systemie MacOS), zainstalować interfejs naukowy platformy Python (np. *Anaconda*) i skorzystać z jego zasobów albo po prostu wykorzystać wewnętrzny menedżer pakietów Python (*pip*), domyślnie instalowany (począwszy od wersji Python 2.7.9)⁶ wraz z plikami binarnymi omawianego języka. Za pomocą poniższej komendy możesz sprawdzić, czy menedżer *pip* jest zainstalowany na Twoim systemie:

```
$ python3 -m pip --version  
pip 19.3.1 from [...]/lib/python3.7/site-packages (python 3.7)
```

Aby system obsługiwał możliwość instalowania modułów binarnych, musisz mieć zainstalowaną najnowszą wersję menedżera *pip*. W celu zaktualizowania modułu *pip* wpisujemy następujące polecenie (dokładna wersja modułu może być inna)⁷:

```
$ python3 -m pip install --user -U pip  
Collecting pip  
[...]  
Successfully installed pip-19.3.1
```

Możesz teraz zainstalować wszystkie wymagane moduły wraz z zależnościami przy użyciu następującej prostej komendy *pip* (jeżeli nie korzystasz ze środowiska wirtualnego, musisz dodać opcję *--user* albo uprawnienia administratora):

```
$ python3 -m pip install -U jupyter matplotlib numpy pandas scipy scikit-learn  
Collecting jupyter  
Downloading https://[...]/jupyter-1.0.0-py2.py3-none-any.whl  
Collecting matplotlib  
[...]
```

⁵ Zalecam korzystanie z najnowszego wydania środowiska Python 3. Platforma Python 2.7+ powinna również działać bez zarzutu, ale pamiętaj, że jest już przestarzała. Wszystkie główne biblioteki naukowe porzucają obsługę wersji 2. Pythona, dlatego należy przenieść się jak najszybciej do środowiska Python 3.

⁶ Zaprezentuję proces instalacji za pomocą menedżera *pip* w konsoli systemu Linux lub macOS. Może pojawić się potrzeba zmodyfikowania tych poleceń na innych systemach operacyjnych. W przypadku użytkowników systemów z rodziną Windows zalecam zainstalowanie platformy *Anaconda*.

⁷ Jeżeli chcesz zaktualizować moduł *pip* dla wszystkich użytkowników komputera, musisz usunąć opcję *--user* i wykonać komendę z poziomu administratora (np. umieszcając na początku wiersza komendę *sudo* w systemie macOS lub Linux).

Środowisko izolowane

Jeśli chcesz pracować w izolowanym środowisku (do czego gorąco namawiam, gdyż możesz pracować nad różnymi projektami bez ryzyka konfliktu pomiędzy różnymi wersjami bibliotek), zainstaluj moduł `virtualenv`⁸ za pomocą następującego polecenia (również w tym przypadku, jeżeli chcesz, aby środowisko wirtualne zostało zainstalowane dla wszystkich użytkowników komputera, usuń człon `--user` i uruchom komendę przy użyciu uprawnień administratora):

```
$ python3 -m pip install --user -U virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv-16.7.7
```

Możesz teraz stworzyć izolowane środowisko Pythona:

```
$ cd $ML_PATH
$ python3 -m virtualenv moje_srod
Using base prefix '[...]'
New python executable in [...]/um/moje_srod/bin/python3
Also creating executable in [...]/um/moje_srod/bin/python
Installing setuptools, pip, wheel...done.
```

Teraz za każdym razem, gdy zechcesz uruchomić to środowisko, wystarczy otworzyć terminal i wpisać poniższe polecenia:

```
$ cd $ML_PATH
$ source moje_srod/bin/activate # W Linuksie lub macOS
$ .\moje_srod\Scripts\activate # W Windowsie
```

Aby wyłączyć to środowisko, wpisz polecenie **deactivate**. Przy włączonym środowisku izolowanym będą w nim instalowane wszelkie pakiety dostarczane poprzez menedżer `pip`; dostęp do nich będzie miała wyłącznie platforma Python (jeżeli chcesz uzyskać dostęp również do pakietów systemowych, stwórz środowisko izolowane przy użyciu opcji `--system-site-packages`).Więcej informacji znajdziesz w dokumentacji modułu `virtualenv`.

Jeżeli stworzyłaś/stworzyłeś środowisko wirtualne, musisz zarejestrować je w środowisku Jupyter i nadać mu nazwę:

```
$ python3 -m ipykernel install --user --name=python3
```

Włączmy teraz aplikację Jupyter:

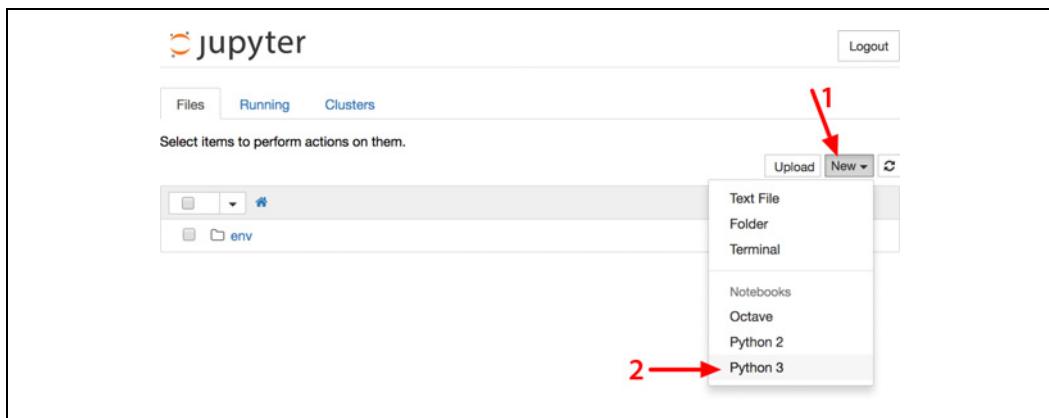
```
$ jupyter notebook
[...] Serving notebooks from local directory: [...]/um
[...] The Jupyter Notebook is running at:
http://localhost:8888/?token=60995e108e44ac8d8865a[...]
[...] or http://127.0.0.1:8889/?token=60995e108e44ac8d8865a[...]
[...] Use Control-C to stop this server and shut down all kernels [...]
```

Serwer Jupyter już działa w terminalu i nasłuchiwa portu 8888. Możesz zająrzyć na ten serwer, wpisując w przeglądarce adres `http://localhost:8888/` (zazwyczaj następuje to automatycznie w momencie

⁸ Do alternatywnych narzędzi zaliczamy `venv` (bardzo przypomina `virtualenv` i stanowi część biblioteki standardowej), `virtualenvwrapper` (dodaje pewne funkcje do modułu `virtualenv`), `pyenv` (umożliwia łatwe przełączanie pomiędzy wersjami Pythona), a także `pipenv` (znakomite narzędzie autora m.in. popularnej biblioteki `requests`, stanowiące dodatek do modułów `pip` i `virtualenv`).

uruchomienia serwera). Twoim oczom powinien ukazać się pusty katalog roboczy (zawierający仅仅 folder *env*, jeśli zainstalowałaś/zainstalowałeś również środowisko *virtualenv*).

Stworzymy teraz nowy notatnik Jupyter, klikając przycisk *New* i wybierając właściwą wersję środowiska Python⁹ (rysunek 2.3). W ten sposób zostaje utworzony nowy notatnik, o nazwie *Untitled.ipynb*, w obszarze roboczym, następuje uruchomienie jądra Jupyter Python obsługującego tenże notatnik, a następnie notatnik ten zostaje otwarty w nowej zakładce. Zaczniemy od zmiany nazwy notatnika na *Mieszkania* (w rzeczywistości zostanie zmieniona nazwa pliku na *Mieszkania.ipynb*) — wystarczy kliknąć napis *Untitled* i wprowadzić własną nazwę.



Rysunek 2.3. Twój obszar roboczy w aplikacji Jupyter

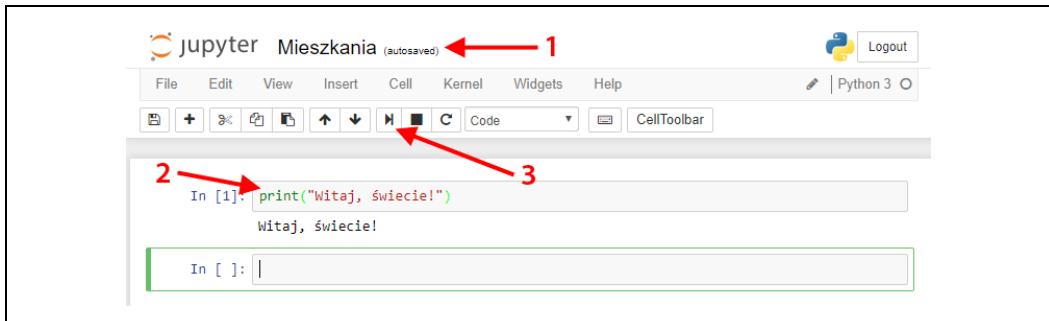
Notatnik składa się z komórek. W każdej komórce umieszczamy albo kod wykonywalny, albo sformatowany tekst. Obecnie w naszym notatniku znajduje się tylko jedna, pusta komórka, oznaczona jako *In [1]:*. Spróbuj wpisać w niej komendę `print("Witaj, świecie!")` i kliknij przycisk odtwarzania (ikona oznaczona liczbą 3 na rysunku 2.4) lub użyj kombinacji klawiszy *Shift+Enter*. Zawartość tej komórki zostanie przesłana do jądra środowiska Python, gdzie nastąpi jej przetwarzanie, po czym ujrzymy jej wynik na ekranie. Rezultat zostaje umieszczony pod daną komórką, a ponieważ dotarłaś/dotarłeś do końca notatnika, automatycznie nastąpi utworzenie nowej komórki. Aby poznać podstawy obsługi notatników Jupyter, zapoznaj się z samouczkiem *User Interface Tour*, dostępnym w menu pomocy (*Help*) aplikacji Jupyter.

Pobierz dane

W typowym środowisku dane są najczęściej umieszczone w relacyjnej bazie danych (lub innym po-wszechnie używanym magazynie danych) i rozrzucone pomiędzy wiele tabel/dokumentów/plików. Aby uzyskać do nich dostęp, należy najpierw uzyskać autoryzację do ich przeglądania lub jakieś inne poświadczenie¹⁰, a następnie zaznajomić się z używanym schematem danych. W naszym projekcie

⁹ Zwrót uwagę, że aplikacja Jupyter obsługuje różne wersje języka Python, a nawet wiele innych języków, takich jak R czy Octave.

¹⁰ Musisz również brać pod uwagę ograniczenia prawne, np. prywatne pola, których nie należy nigdy kopować do niezbyt bezpiecznych magazynów danych.



Rysunek 2.4. Witaj, świecie!

jest jednak znacznie prościej: pobierzemy jedynie archiwum (*housing.tgz*) zawierające plik CSV (ang. *comma-separated values* — wartości rozdzielone przecinkami) o nazwie *housing.csv*, w którym mieszkały się wszystkie interesujące nas dane.

Możesz pobrać to archiwum z poziomu przeglądarki i rozpakować je za pomocą komendy tar -xzf *housing.tgz*, ale lepiej stworzyć w tym celu niewielką funkcję. Funkcja pobierająca dane przydaje się zwłaszcza w sytuacjach, gdy są one regularnie aktualizowane: możesz stworzyć małe skrypty uruchamiane za każdym razem, gdy będziesz potrzebować najświeższych danych (ewentualnie możesz zaplanować wykonywanie tej czynności w regularnych odstępach czasu). Automatyzacja tego procesu ułatwia również instalację zbioru danych na wielu komputerach.

Funkcja pobierająca dane przedstawia się następująco¹²:

```
import os
import tarfile
import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
HOUSING_PATH = os.path.join("zestawy danych", "mieszkania")
HOUSING_URL = DOWNLOAD_ROOT + "zestawy danych/mieszkania/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
    os.makedirs(housing_path, exist_ok=True)
    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()
```

Teraz po wywołaniu funkcji `fetch_housing_data()` zostanie utworzony katalog *zestawy danych/mieszkania* w Twojej przestrzeni roboczej, do którego będzie pobrane archiwum *housing.tgz*, po czym zostanie wypakowany z niego plik *housing.csv*.

¹¹ Musisz również brać pod uwagę ograniczenia prawne, np. prywatne pola, których nie należy nigdy kopować do niezbyt bezpiecznych magazynów danych.

¹² W prawdziwym projekcie umieścilibyśmy ten kod w pliku Python, na razie jednak wystarczy wstawić go w notatniku Jupyter.

Wczytajmy teraz te dane za pomocą modułu pandas. Także i w tym przypadku stworzymy niewielką funkcję służącą do ich odczytania.

```
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)
```

Powyzsza funkcja zwraca obiekt *DataFrame* modułu pandas przechowujący wszystkie wczytane dane.

Rzut oka na strukturę danych

Przyjrzyjmy się pierwszym pięciu wierszom naszych danych za pomocą metody `head()` (rysunek 2.5)¹³.

	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

Rysunek 2.5. Pięć pierwszych wierszy w naszym zbiorze danych

Każdy wiersz reprezentuje jeden dystrykt. Dostępnych jest 10 atrybutów, z czego sześć widzimy na rysunku 2.5: `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` i `ocean_proximity` (w języku polskim są to, odpowiednio: *Dł. geograficzna*, *Szer. geograficzna*, *Mediana wieku mieszkańców*, *Całk. liczba pokoi*, *Całk. liczba sypialni*, *Populacja*, *Rodziny*, *Medianana dochodów*, *Medianana cen mieszkań* i *Odległość do oceanu*).

Dzięki metodzie `info()` możemy zapoznać się z krótkim opisem danych, zwłaszcza z całkowitą liczbą wierszy, typem każdego atrybutu oraz liczbą wartości niezerowych (rysunek 2.6).

Na nasz zbiór danych składa się 20 640 przykładów, co oznacza, że jest on dość mały jak na standardy uczenia maszynowego, ale nadaje się idealnie dla początkujących analityków. Zwróć uwagę, że atrybut `total_bedrooms` zawiera zaledwie 20 433 wartości niezerowe, co oznacza, że cecha ta nie została zdefiniowana dla 207 dystryktów. Zajmiemy się tym później.

¹³ W celach poglądowych pozostawiam w tym rozdziale nieprzetłumaczone nazwy atrybutów, dzięki czemu po przepisaniu kodu z książki uzyskasz takie same wyniki, jak na rysunkach, natomiast w towarzyszącym notatniku Jupyter znajdziesz wersję tego projektu częściowo przełożoną (tam, gdzie to konieczne) na język polski — *przyp. tłum.*

```
In [6]: housing.info()

<class 'pandas.core.frame.DataFrame'>
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
```

Rysunek 2.6. Informacje o zbiorze danych

Z wyjątkiem pola `ocean_proximity` wszystkie pozostałe atrybuty mają wartości numeryczne. Typem wartości w polu `ocean_proximity` jest `object`, co oznacza, że można tu przechowywać dowolny obiekt języka Python. Skoro jednak wczytaliśmy dane z pliku CSV, to wiemy, że musi to być wartość tekstowa. Po przyjrzeniu się pierwszym pięciu wierszom zbioru danych okazuje się, że wartości w kolumnie `ocean_proximity` są powtarzalne, dzięki czemu możemy wywnioskować, że mamy najprawdopodobniej do czynienia z atrybutem kategorialnym. Możemy sprawdzić, jakie kategorie są dostępne oraz jaki jest rozkład poszczególnych dystryktów do każdej z nich; posłużymy się w tym celu metodą `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
```

Przyjrzyjmy się innym polom. Metoda `describe()` generuje podsumowanie atrybutów numerycznych (rysunek 2.7).

```
In [8]: housing.describe()

Out[8]:
```

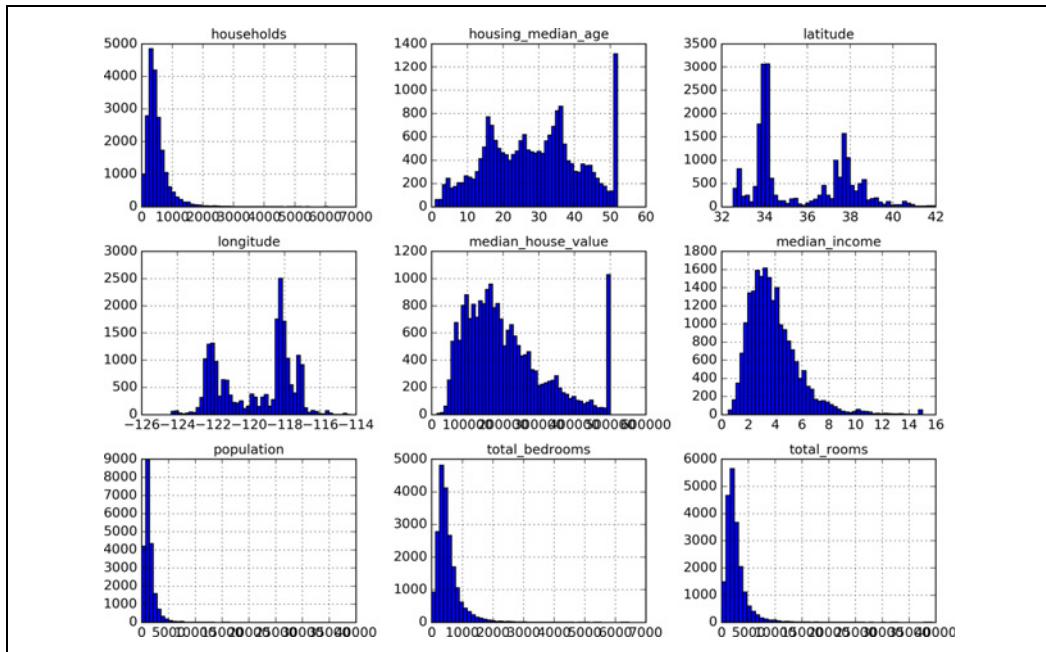
	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.0000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Rysunek 2.7. Podsumowanie wszystkich atrybutów numerycznych

Wartości count (liczba), mean (średnia), min i max raczej nie wymagają wyjaśnienia. Zwróć uwagę, że wartości zerowe są ignorowane (z tego powodu liczba elementów total_bedrooms wynosi tu 20 433, a nie 20 640). Wiersz std zawiera **odchylenie standardowe** (ang. *standard deviation*), określające „rozrzut” wartości¹⁴. Wartości 25%, 50% i 75% ukazują odpowiadające im **percentyle**: percentyl wskazuje wartość, poniżej której znajduje się określony odsetek obserwowanych przykładów. Przykładowo, 25% dystryktów ma wartość atrybutu housing_median_age mniejszą od 18, z kolei 50% próbek nie przekracza wartości 29, a 75% procent nie osiągnęło wieku 37 lat. Parametry te są często nazywane, kolejno: 25. percentilem (lub pierwszym **kwartylem**), medianą oraz 75. percentilem (lub trzecim kwartylem).

Kolejnym szybkim sposobem przyjrzenia się analizowanym danym jest wygenerowanie histogramu dla każdego atrybutu numerycznego. Histogram przedstawia liczbę przykładów (w pionowej osi) znajdujących się w określonym przedziale wartości (oś pozioma). Możesz stworzyć histogram jednego atrybutu lub za pomocą metody hist() dokonać tego dla całego zbioru danych, co spowoduje narysowanie histogramu dla każdego atrybutu numerycznego (rysunek 2.8 i poniższy listing):

```
%matplotlib inline # Wyłącznie w notatniku Jupyter
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Rysunek 2.8. Histogram każdego atrybutu numerycznego

¹⁴ Odchylenie standardowe jest zwyczajowo oznaczane greczką literą σ (sigma) i w ujęciu matematycznym stanowi pierwiastek kwadratowy z **wariancji**, którą z kolei definiujemy jako średnią arytmetyczną kwadratów odchyleń od wartości średnich przykładów. Gdy wykres danej cechy przyjmuje dzwonowy kształt **rozkładu normalnego** (zwanego również **rozkładem Gaussa**), co jest bardzo często spotykane, znajduje zastosowanie tzw. **reguła 68-95 - 99,7%**: mniej więcej 68% próbek znajduje się w odległości 1σ od wartości oczekiwanej, 95% w odległości 2σ , a 99,7% — w odległości 3σ .



Metoda `hist()` korzysta z modułu Matplotlib, który z kolei jest uzależniony od platformy graficznej zdefiniowanej przez użytkownika. Zatem zanim zostanie narysowany jakikolwiek wykres, musisz wybrać platformę używaną przez moduł Matplotlib. Najprostszym rozwiązaniem jest wprowadzenie magicznego polecenia aplikacji Jupyter: `%matplotlib inline`. W ten sposób moduł Matplotlib zostaje skonfigurowany do korzystania z „zaplecza” notatników Jupyter. Oznacza to, że wykresy są generowane w samym notatniku. Zwróć uwagę, że wywołanie funkcji `show()` jest całkowicie dozwolone w notatnikach Jupyter, gdyż wykres zostanie tak czy inaczej narysowany po uruchomieniu odpowiedniej komórki.

Możemy wyczytać z tych histogramów kilka informacji:

1. Po pierwsze, atrybut mediany dochodów nie przypomina danych podawanych w dolarach amerykańskich (USD). Po konsultacji z zespołem odpowiedzialnym za zebranie tych danych wiesz już, że dane te zostały przeskalowane i ograniczone do maksymalnej wartości 15 (w rzeczywistości 15,0001) dla wyższej mediany dochodów oraz do minimalnej wartości 0,5 (w rzeczywistości 0,4999) dla niższej mediany dochodów. Liczby reprezentują w przybliżeniu dziesiątki tysięcy dolarów (np. wartość 3 oznacza w rzeczywistości ok. 30 000 dolarów). Praca ze wstępnie przetworzonymi danymi stanowi normę w uczeniu maszynowym i niekoniecznie musi stanowić problem, należy jednak spróbować zrozumieć, jakim operacjom zostały one poddane.
2. Ograniczeniu uległy również wartości median wieku oraz cen mieszkań. Ta druga informacja może nas szczególnie zmartwić, ponieważ stanowi ona nasz docelowy atrybut (etykietę). Algoritmy uczenia maszynowego mogą uznać, że ceny domów nigdy nie przekraczają górnej, ograniczonej wartości. Musisz skontaktować się z zespołem klienckim (zespołem wykorzystującym uzyskane przez Ciebie wyniki) i dowiedzieć się, czy ograniczenie to będzie stanowiło problem. Jeżeli powiedzą Ci, że potrzebne są im precyzyjne prognozy nawet powyżej wartości 500 000 dolarów, to pozostają Ci dwie możliwości:
 - a. uzyskać prawidłowe etykiety dla dystryktów mających obcięte górne wartości cen;
 - b. usunąć te dystrykty z zestawu uczącego (a także testowego, ponieważ system nie powinien być karany, jeżeli będzie przewidywał wartości przekraczające 500 000 dolarów).
3. Każdy z tych atrybutów jest ukazany w odmiennych skalach, nieraz znacznie zróżnicowanych. Zajmiemy się tym zagadnieniem w dalszej części rozdziału, podczas omawiania skalowania cech.
4. Wiele histogramów cechuje się **rozkładem długogonowym** (ang. *heavy-tailed*): rozciągają się one znacznie bardziej po prawej stronie mediany niż po lewej. Utrudnia to nieco niektórym algorytmom uczenia maszynowego rozpoznawanie wzorców. Spróbujemy później przekształcić te atrybuty w taki sposób, aby ich rozkład był bardziej dzwonowy.

Mam nadzieję, że już wiesz co nieco na temat danych, którymi będziemy się zajmować.



Zaczekaj! Zanim znowu zajrzesz do danych, musisz stworzyć z nich podzbior testowy, odłożyć go i nigdy więcej do niego nie zerkać.

Stwórz zbiór testowy

Być może dziwnym pomysłem wydaje się dobrowolne odłożenie części danych na tym etapie. Przecież ledwie zajrzaliśmy do nich, aby poznać ich strukturę, i z pewnością moglibyśmy wyciągnąć na ich temat znacznie więcej wniosków przed wybraniem jakiegoś algorytmu, prawda? Owszem, z tym że Twój mózg zawiera niesamowity układ rozpoznawania wzorców, co oznacza, że jest znacznie narażony na przetrenowanie: jeśli przyjrzyisz się zbiorowi testowemu, możesz dostrzec jakiś pozornie interesujący wzorzec, który sprawi, że wybierzesz określony model uczenia maszynowego. Po oszacowaniu błędu uogólniania za pomocą tego zbioru uzyskane wyniki okażą się nadmiernie optymistyczne i uruchomisz system, którego wydajność będzie mniejsza od zakładanej. Zjawisko to jest nazywane **obciążeniem związanym z podglądamieniem danych** (ang. *data snooping bias*).

Teoretycznie tworzenie zbioru testowego nie powinno być zbyt kłopotliwe: wystarczy wybrać losowo część przykładów (najczęściej 20% całego zbioru lub mniej, jeżeli zestaw danych jest bardzo duży) i je odłożyć.

```
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]
```

Teraz możesz użyć powyższej funkcji w następujący sposób¹⁵:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

To rozwiązanie działa, ale nie jest idealne: przy następnym uruchomieniu programu zostanie wygenerowany zupełnie inny zbiór testowy! Po pewnym czasie Ty (albo algorytm uczenia maszynowego) zobaczyesz cały zbiór danych uczących, a tego właśnie chcemy uniknąć.

Jednym z rozwiązań jest zapisanie zestawu testowego po jego pierwszym utworzeniu i wczytywanie go przy kolejnych okazjach. Możesz również skorzystać z zarodka liczb losowych (np. `random.seed(42)`)¹⁶ przed wywołaniem funkcji `np.random.permutation()` i uzyskiwać za każdym razem te same pseudolosowe indeksy.

Obydwa te rozwiązania jednak zawiodą po dostarczeniu nowej porcji zaktualizowanych danych. Aby dysponować stabilnym podziałem na przykłady uczące/testowe nawet po zaktualizowaniu zestawu danych, możemy dobierać przykłady stanowiące część zbioru testowego za pomocą ich identyfika-

¹⁵ W niniejszej ksiązce listingi składające się zarówno z kodu, jak i wyników (tak jak w omawianym przypadku) są dla zwiększenia czytelności formatowane na wzór interpretera Pythona: wiersze kodu mają na początku oznaczenie `>>>` (lub ... w przypadku wcięcia), natomiast rezultaty nie zawierają żadnego przedrostka.

¹⁶ Ludzie często korzystają z zarodka o wartości 42. Liczba ta nie jest w żaden sposób wyjątkowa, stanowi jedynie odpowiedź na odwieczne pytanie o źródło życia, wszechświata i wszystkiego.

torów (przy założeniu, że próbki te zawierają niepowtarzalne i niezmienne identyfikatory). Możesz na przykład obliczyć hasz każdego identyfikatora i umieścić ten przykład w zestawie danych testowych, jeżeli wartość hasza jest mniejsza lub równa 20% maksymalnej wartości hasza. Dzięki temu jesteśmy w stanie zagwarantować stabilność zestawu testowego za każdym razem, gdy uruchamiamy program, nawet po odświeżeniu zbioru danych. Nowy zbiór testowy będzie zawierał 20% świeżych przykładów, ale nie pojawi się w nim żaden przykład występujący wcześniej w zbiorze uczącym. Oto przykładowa implementacja tego mechanizmu:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

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

Niestety zbiór danych *Housing* nie zawiera kolumny z identyfikatorami. Najprostszym rozwiązaniem okazuje się wykorzystanie w tym celu indeksu wiersza.

```
housing_with_id = housing.reset_index() # Dodaje kolumnę 'index'
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
```

Jeśli używasz indeksu wiersza jako niepowtarzalnego identyfikatora, musisz się upewnić, że nowe informacje będą umieszczane na końcu zestawu danych i że żaden wiersz nie zostanie nigdy usunięty. Jeśli nie jesteś w stanie tego zagwarantować, możesz spróbować wykorzystać najstabilniejsze cechy do stworzenia niepowtarzalnego identyfikatora. Przykładowo, współrzędne geograficzne dystryktów raczej nie zmieniają się przez kilka najbliższych milionów lat, dlatego możesz wygenerować z nich identyfikatory w zaprezentowany poniżej sposób¹⁷:

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

Moduł Scikit-Learn zawiera kilka różnych funkcji rozdzielających zbiory danych na wiele podzbiorów. Najprostszą z nich jest `train_test_split()`, której działanie w znacznym stopniu przypomina zdefiniowaną przed chwilą funkcję `split_train_test()`, zawiera jednak ona pewne dodatkowe elementy. Po pierwsze mamy tu do czynienia z parametrem `random_state`, pozwalającym wybrać zarodek liczb losowych. Po drugie natomiast możemy przekazywać wiele zbiorów danych mających taką samą liczbę wierszy — będą one rozdzielane pomiędzy takie same indeksy (jest to bardzo przydatne rozwiązanie w przypadku, gdy np. etykiety znajdują się w osobnym obiekcie *DataFrame*):

```
from sklearn.model_selection import train_test_split

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

¹⁷ W rzeczywistości podane współrzędne geograficzne nie są zbyt precyzyjne i w związku z tym wiele dystryktów otrzyma taką samą wartość identyfikatora oraz znajdzie się w tym samym zbiorze (uczącym bądź testowym). Ku naszemu utrapieniu stanowi to jedno ze źródeł obciążenia próbkowania.

Do tej pory rozważaliśmy metody całkowicie losowego próbkowania. Zazwyczaj takie rozwiązanie spisuje się dobrze w przypadku odpowiednio dużych zbiorów danych (zwłaszcza w odniesieniu do liczby atrybutów), w przeciwnym razie jednak ryzykujemy wprowadzenie dużego obciążenia próbkowania. Gdy urząd statystyczny zamierza przeprowadzić ankietę na tysiącu osób, nie są one wybierane losowo z książki telefonicznej. Muszą zostać tak dobrane, aby stanowiły reprezentatywny przykład całej populacji. Przykładowo, populacja Stanów Zjednoczonych składa się w 51,3% z kobiet i 48,7% z mężczyzn, zatem prawidłowo przeprowadzona ankjeta powinna uwzględniać te proporcje: 513 kobiet i 487 mężczyzn. Jest to tzw. **losowanie warstwowe** (ang. *stratified sampling*): populację zostaje rozdzielona na jednorodne podgrupy zwane **warstwami** (ang. *strata*, l. poj. *stratum*) i z każdej z nich jest dobierana odpowiednia liczba przykładów zapewniająca prawidłowe odzwierciedlenie stanu populacji. Gdyby ankieterzy używali metod czysto losowego doboru przykładów, istniałoby 12-procentowe prawdopodobieństwo uzyskania wypaczonego zestawu testowego zawierającego mniej niż 49% lub ponad 54% kobiet. W każdym przypadku wyniki ankiety cechowałyby się znacznym obciążeniem próbkowania.

Załóżmy, że po rozmowie z ekspertami okazuje się, że bardzo ważnym atrybutem pomagającym w prognozowaniu mediany cen mieszkań jest mediana dochodów. Możesz chcieć zagwarantować, żeby zbiór testowy wiernie reprezentował różne kategorie dochodów dla całego zbioru danych. Mediana dochodów stanowi atrybut ciągłych wartości numerycznych, dlatego najpierw należy stworzyć atrybuty kategorii dochodów. Przyjrzymy się uważniej histogramowi mediany dochodów na rysunku 2.8: większość wartości mieści się w przedziale 1,5 do 6 (tzn. 15 000 – 60 000 dolarów), jednak niektóre znacznie przekraczają 6. Ważne jest, aby każda warstwa zestawu danych zawierała wystarczającą liczbę przykładów, gdyż w przeciwnym wypadku oszacowanie znaczenia danej warstwy może być nieadekwatne do rzeczywistości. Oznacza to, że nie możemy tworzyć zbyt wielu warstw i każda z nich powinna być wystarczająco duża. Poniższy kod wykorzystuje funkcję `pd.cut()` do utworzenia atrybutu kategorii dochodów składającego się z pięciu kategorii (oznaczonych cyframi od 1 do 5): zakres pierwszej kategorii wynosi od 0 do 1,5 (tzn. poniżej 15 000 dolarów), drugiej kategorii od 1,5 do 3 itd.:

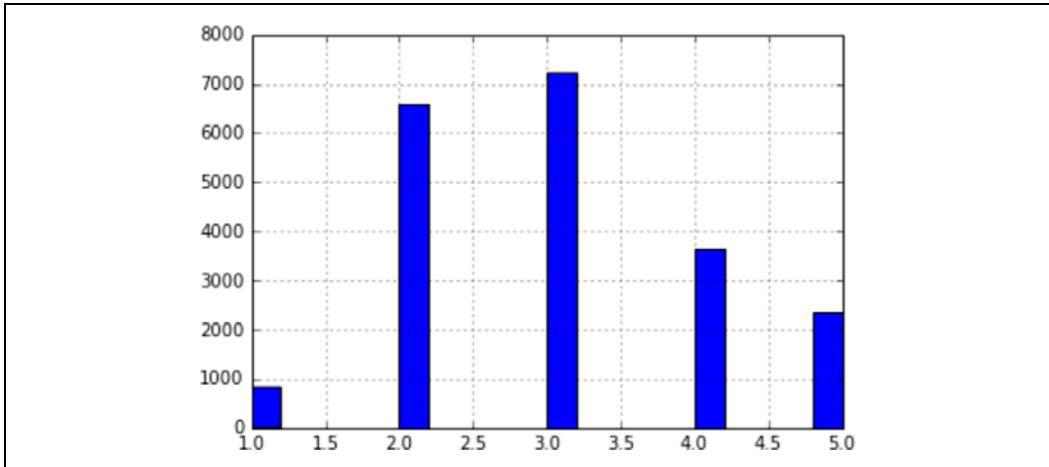
```
housing["income_cat"] = pd.cut(housing["median_income"],
                                bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                labels=[1, 2, 3, 4, 5])
```

Uzyskane kategorie zostały pokazane na rysunku 2.9:

```
housing["income_cat"].hist()
```

Teraz możemy przeprowadzić próbkowanie warstwowe na podstawie kategorii dochodów. Użyjemy w tym celu klasy `StratifiedShuffleSplit` modułu 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]
```



Rysunek 2.9. Histogram kategorii dochodów

Sprawdźmy, czy mechanizm ten działa zgodnie z naszymi założeniami. Najpierw możemy zobaczyć proporcje kategorii dochodów w zestawie testowym:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3      0.350533
2      0.318798
4      0.176357
5      0.114583
1      0.039729
Name: income_cat, dtype: float64
```

Za pomocą podobnego kodu możemy mierzyć proporcje kategorii przychodów w pełnym zestawie danych. Na rysunku 2.10 porównujemy te proporcje w całym zbiorze danych, w zbiorze testowym wygenerowanym za pomocą losowania warstwowego oraz w zestawie testowym utworzonym w całkowicie losowy sposób. Jak widać, zbiór testowy wygenerowany przy użyciu losowania warstwowego ma proporcje niemal identyczne, jak uzyskane w pełnym zbiorze danych, podczas gdy wyniki uzyskane w całkowicie losowym zestawie danych wskazują wypaczenie proporcji.

	L. warstwowe	Losowe	Łącznie	Błąd - losowe (%)	Błąd - l. warstwowe (%)
1.0	0.039729	0.040213	0.039826	0.973236	-0.243309
2.0	0.318798	0.324370	0.318847	1.732260	-0.015195
3.0	0.350533	0.358527	0.350581	2.266446	-0.013820
4.0	0.176357	0.167393	0.176308	-5.056334	0.027480
5.0	0.114583	0.109496	0.114438	-4.318374	0.127011

Rysunek 2.10. Porównanie obciążenia próbkowania dla losowania warstwowego i losowego próbkowania

Teraz powinniśmy usunąć atrybut `income_cat`, dzięki czemu dane zostaną przywrócone do pierwotnego stanu:

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

Nie bez powodu poświęciliśmy tyle czasu na zagadnienie tworzenia zbioru testowego: jest to często bagatelowany, ale jeden z najważniejszych elementów uczenia maszynowego. Do tego wiele z poruszonych tu koncepcji przyda nam się w dalszej części książki, podczas omawiania sprawdzianu krzyzowego. Przejdźmy teraz do następnego etapu: eksplorowania danych.

Odkrywaj i wizualizuj dane, aby zdobywać nowe informacje

Na razie zerknęliśmy jedynie na dane w celu ustalenia rodzaju informacji, z jakimi będziemy pracować. Teraz naszym celem jest zagłębienie się w te dane.

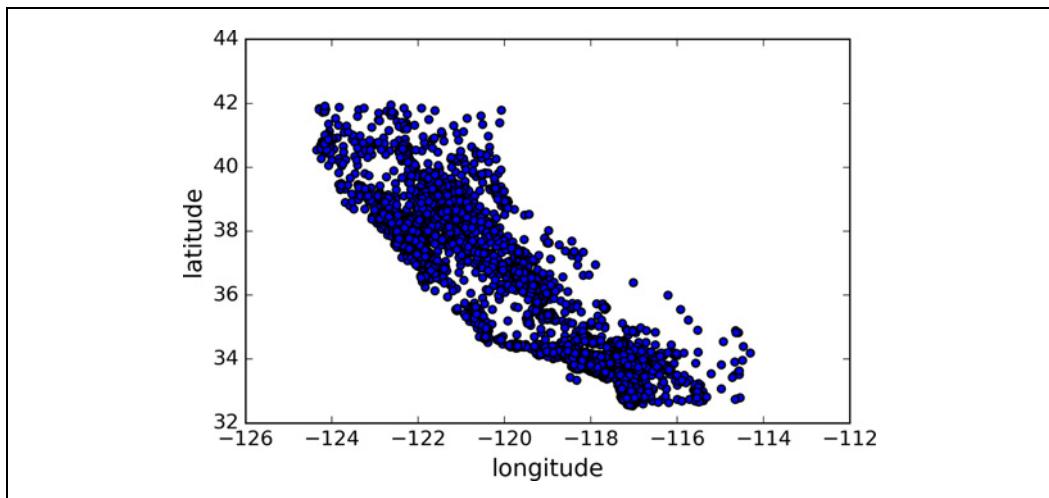
Upewnijmy się najpierw, że odłożyliśmy zbiór testowy i że zajmujemy się wyłącznie zestawem uczącym. Poza tym jeśli zbiór uczący ma bardzo duże rozmiary, warto wydzielić z niego zbiór eksploracyjny, aby przyśpieszyć i ułatwić sobie manipulowanie danymi. W naszym przypadku zestaw uczący jest na tyle mały, że możemy korzystać ze wszystkich danych. Stwórzmy jego kopię, aby móc się nim bawić bez strachu, że go w jakiś sposób uszkodzimy:

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

Wizualizowanie danych geograficznych

Skoro mamy do czynienia z danymi geograficznymi (długością i szerokością geograficzną), warto stworzyć wykres punktowy wszystkich dystryktów, aby zobaczyć ich rozmieszczenie w układzie współrzędnych (rysunek 2.11):

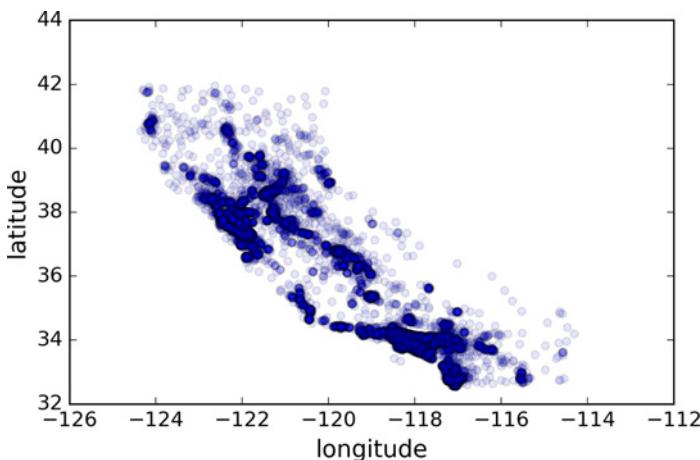
```
housing.plot(kind="scatter", x="longitude", y="latitude")
```



Rysunek 2.11. Geograficzny wykres punktowy danych

Kształt tego wykresu rzeczywiście przypomina Kalifornię, jednak poza tym ciężko doszukać się jakiegokolwiek wzoru. Po wyznaczeniu wartości 0,1 parametru alpha będzie nam o wiele łatwiej zwizualizować miejsca, w których występuje duże zagnieszczenie punktów danych (rysunek 2.12):

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



Rysunek 2.12. Lepsza wizualizacja, ukazująca obszary o dużym zagęszczeniu punktów danych

Teraz jest znacznie lepiej: wyraźnie widać obszary o dużym zagęszczeniu dystryktów, mianowicie rejon Bay Area oraz okolice miast Los Angeles i San Diego, a także długi pas w Dolinie Kalifornijskiej, zwłaszcza wokół miast Sacramento i Fresno.

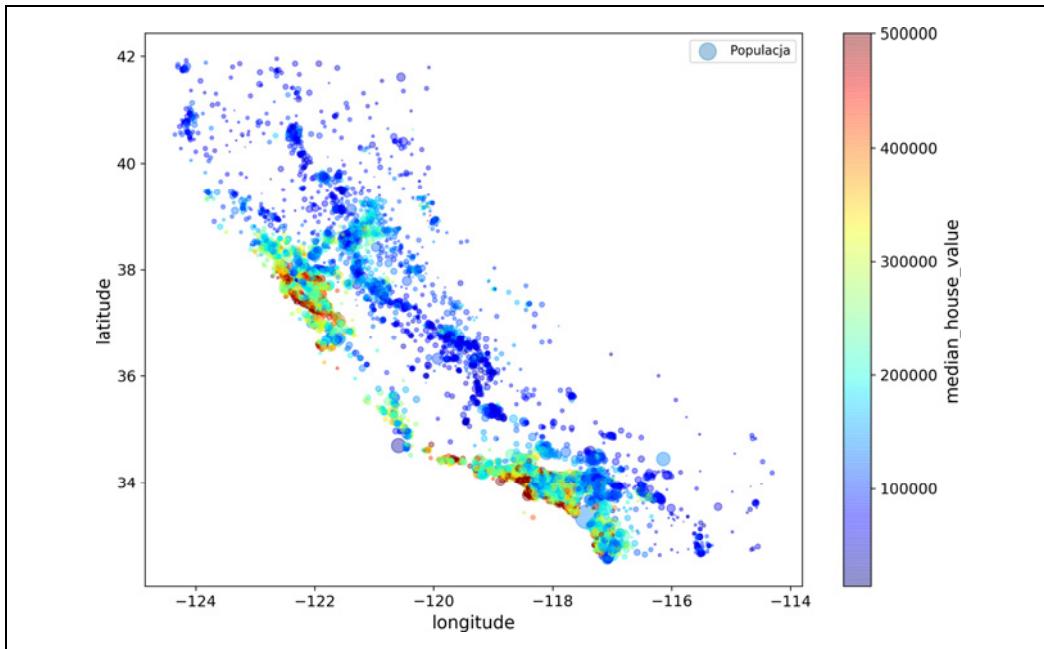
Nasze mózgi doskonale radzą sobie z dostrzeganiem wzorów na danych przedstawionych w postaci graficznej, ale czasami trzeba sobie pomagać, modyfikując niektóre parametry wizualizacji.

Przyjrzyjmy się teraz sytuacji z cenami mieszkań (rysunek 2.13). Promień każdego widocznego kółka symbolizuje populację dystryktu (opcja s), z kolei kolorami oznaczamy ceny mieszkań (opcja c). Skorzystamy z domyślnej mapy kolorów (opcja cmap) o nazwie jet, której zakres mieści się od niebieskiego (małe ceny) do czerwonego (duże ceny)¹⁸:

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

Możemy z tego wykresu wyczytać, że ceny mieszkań są w dużej mierze uzależnione od położenia geograficznego (np. od odległości do oceanu) i zaludnienia, o czym już zapewne wiesz. Algorytm analizy skupień powinien pomóc w wykryciu największego skupiska, a także dodać nowe cechy, pozwalające określić odległość do centrów tego skupienia. Również przydatna może okazać się odległość do oceanu, chociaż w północnej Kalifornii w rejonie wybrzeża ceny domów nie są zbyt wysokie, zatem ta reguła nie jest taka prosta.

¹⁸ Jeżeli czytasz tę książkę w wersji czarno-białej, zaznacz czerwonym pisakiem większość wybrzeża od regionu Bay Area aż do San Diego. Możesz również na żółto otoczyć okolice Sacramento.



Rysunek 2.13. Ceny mieszkań w Kalifornii: na czerwono zostały zaznaczone wysokie ceny mieszkań, na niebiesko niskie ceny, a duże kółka symbolizują obszary o dużym zaludnieniu

Poszukiwanie korelacji

Nasz zbiór danych nie jest zbyt duży, dlatego możemy z łatwością wyliczyć **współczynnik korelacji liniowej** (zwany również **współczynnikiem korelacji Pearsona**) pomiędzy każdą parą wartości za pomocą metody `corr()`:

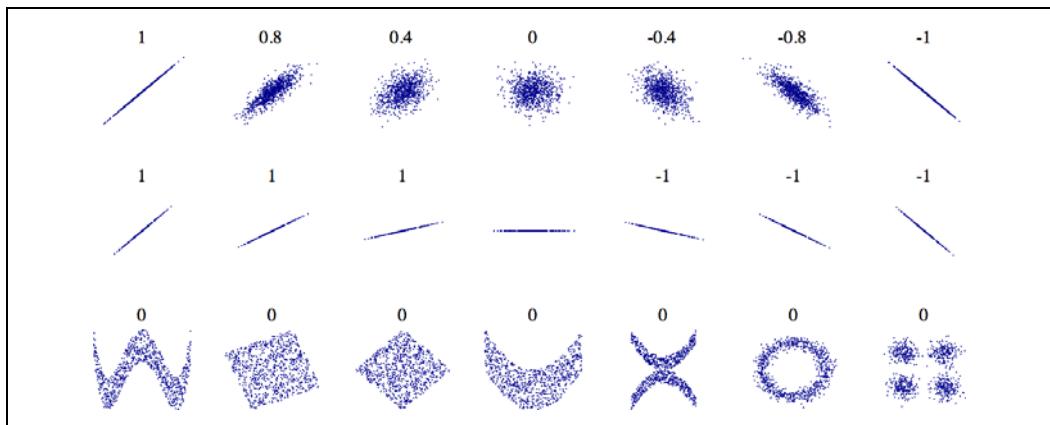
```
corr_matrix = housing.corr()
```

Sprawdźmy teraz stopień korelacji każdego atrybutu z medianą cen mieszkań:

```
>>> 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
```

Wartości współczynnika korelacji mieścią się w zakresie pomiędzy -1 a 1. Wartości zbliżone do 1 wskazują silną korelację dodatnią; na przykład mediana cen mieszkań zazwyczaj rośnie wraz ze wzrostem mediany dochodów. Z kolei wartości zbliżone do -1 mówią nam, że istnieje silna korelacja ujemna; widzimy niewielką korelację ujemną pomiędzy szerokością geograficzną a medianą cen mieszkań

(przykładowo ceny nieznacznie spadają, im bardziej kierujemy się na północ). Natomiast wartości blikskie zera oznaczają brak korelacji liniowej. Na rysunku 2.14 widzimy różne wykresy, a także współczynniki korelacji pomiędzy ich osiami odciętych i rzędnych.



Rysunek 2.14. Współczynnik korelacji liniowej dla różnych zbiorów danych
(źródło: Wikipedia; rysunek stanowi własność publiczną)

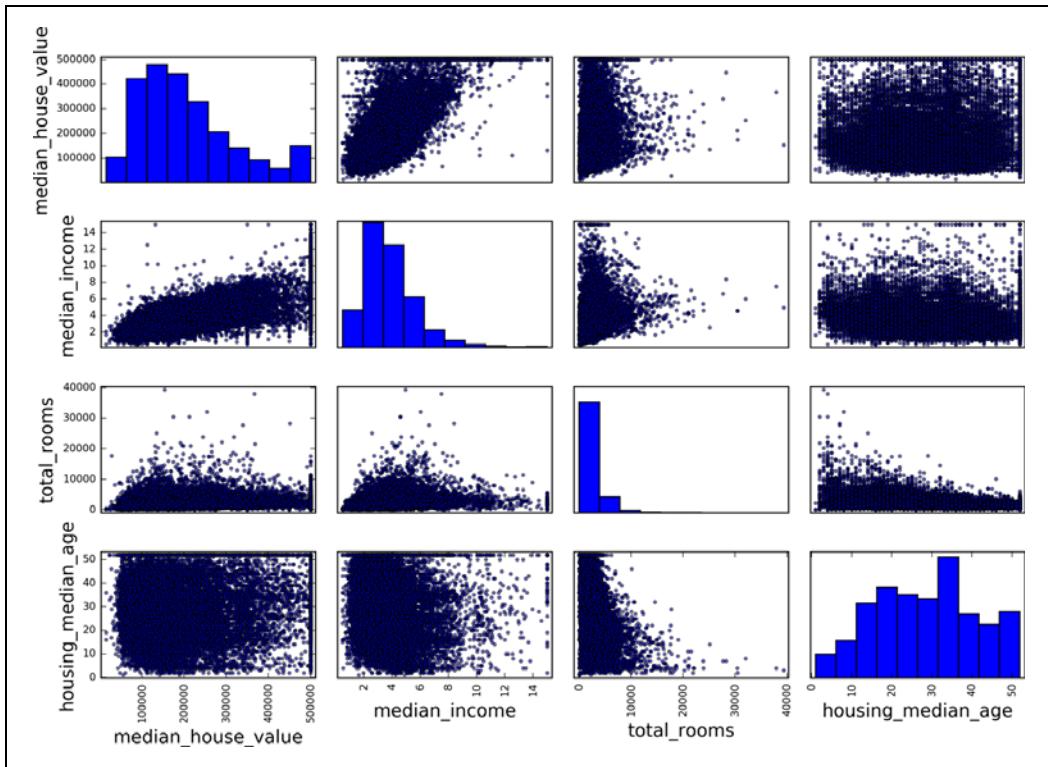


Współczynnik korelacji mierzy jedynie zależność liniową („jeśli wartość x rośnie, to wartość y zazwyczaj rośnie/spada”). Może ona zupełnie nie uwzględniać zależności nieliniowej (np. „jeśli wartość x zbliża się do zera, to wartość y zazwyczaj rośnie”). Zwróć uwagę, że wszystkie wykresy w dolnym rzędzie na rysunku 2.14 mają współczynnik korelacji równy 0 pomimo faktu, że ich osie są w jakiś sposób od siebie zależne: widzimy tu przykład zależności nieliniowej. Ponadto w drugim rzędzie są ukazane przykłady wykresów, w których współczynnik korelacji wynosi 1 lub -1; jak widać, nie ma to żadnego związku z nachyleniem prostej. Przykładowo, Twój wzrost w centymetrach ma współczynnik korelacji liniowej równy 1 dla Twojego wzrostu podanego w metrach lub nanometrach.

Innym sposobem sprawdzenia korelacji pomiędzy atrybutami jest użycie funkcji `scatter_matrix` stanowiącej część modułu pandas; generuje ona wykres każdego atrybutu numerycznego wobec pozostałych atrybutów numerycznych. Obecnie mamy do dyspozycji 11 atrybutów numerycznych, dlatego uzyskalibyśmy w sumie $11^2 = 121$ wykresów, które łącznie nie zmieściłyby się na jednej stronie, dlatego skoncentrujmy się na kilku najbardziej obiecujących atrybutach, które wydają się skorelowane w największym stopniu z medianą cen mieszkań (rysunek 2.15):

```
from pandas.plotting import scatter_matrix

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



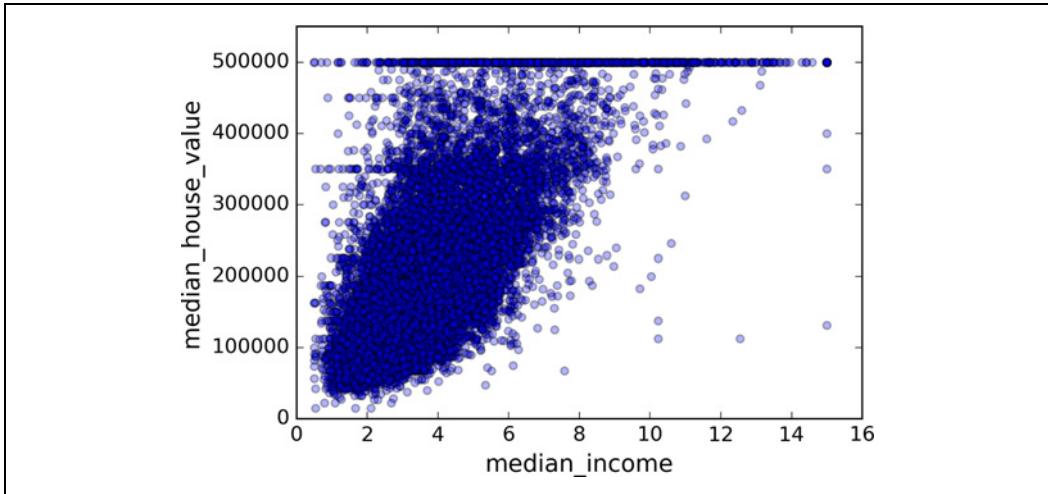
Rysunek 2.15. Zaprezentowany wykres rozproszenia pokazuje każdy atrybut numeryczny w stosunku do pozostałych atrybutów numerycznych, a także histogram poszczególnych atrybutów

Główna przekątna (biegnąca od lewego górnego rogu) byłaby wypełniona prostymi, gdyby moduł pandas tworzył wykresy każdej zmiennej wobec samej siebie, co nie byłoby zbyt przydatne. Dlatego zamiast tego widzimy histogram każdego atrybutu (są dostępne również inne opcje; są one dobrze opisane w dokumentacji modułu pandas).

Najbardziej obiecującym atrybutem służącym do prognozowania mediany cen mieszkań jest mediana dochodów, dlatego przyjrzyjmy się uważniej ich wykresowi korelacji (rysunek 2.16):

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

Z tego wykresu dowiadujemy się kilku rzeczy. Po pierwsze, korelacja pomiędzy tymi atrybutami jest naprawdę silna; możemy bez trudu dostrzec tendencję wzrostową, a poszczególne punkty znajdują się dość blisko siebie. Po drugie, wspomniane już ograniczenie ceny jest wyraźnie widoczne jako pozioma linia w punkcie 500 000 dolarów. Możemy jednak zauważać mniej oczywiste proste: poziomą w punkcie 450 000 dolarów, kolejną w okolicy 350 000 dolarów, być może jeszcze jedną mnóstwo więcej w punkcie 280 000 dolarów, a pod nią kilka innych. Warto byłoby spróbować usunąć z danych odpowiedzialne za te dystrykty, aby uniemożliwić algorytmom naukę odtwarzania takich dziwactw.



Rysunek 2.16. Wykres mediany cen mieszkań w funkcji mediany dochodów

Eksperymentowanie z kombinacjami atrybutów

Mam nadzieję, że dzięki informacjom zawartym w poprzednich punktach poznalaś/poznałeś już kilka sposobów eksplorowania danych i wydobywania z nich dodatkowej wiedzy. Wykryliśmy kilka artefaktów, które warto byłoby usunąć przed przesłaniem danych do algorytmu uczenia maszynowego, a do tego zauważliśmy kilka interesujących korelacji pomiędzy atrybutami, zwłaszcza w związku z naszym docelowym atrybutem. Zwróciliśmy także uwagę, że niektóre atrybuty cechują się rozkładem długooogonowym, dlatego wypadałoby je w jakiś sposób przekształcić (np. poprzez wyliczenie ich logarytmu). Oczywiście każdy projekt jest inny, ale ogólne koncepcje są dość podobne.

Ostatnią czynnością, jaką należałoby wykonać przed przygotowaniem danych do użytku algorytmów uczenia maszynowego, jest wypróbowanie różnych kombinacji atrybutów. Przykładowo całkowita liczba pomieszczeń w dystrykcie nie jest zbyt wartościowym atrybutem, jeśli nie znamy liczbę przebywających tam rodzin. W rzeczywistości interesuje nas liczba pokojów przypadających na rodzinę. Również całkowita liczba sypialni sama w sobie nic nam nie mówi: prawdopodobnie należałoby ją porównać z całkowitą liczbą pomieszczeń. Inną ciekawą kombinację atrybutów jest określenie zależności pomiędzy populacją a liczbą rodzin. Stwórzmy teraz wspomniane pary atrybutów:

```
housing["Pokoje_na_rodzinę"] = housing["total_rooms"]/housing["households"]
housing["Sypialnie_na_pokoje"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["Populacja_na_rodzinę"] = housing["population"]/housing["households"]
```

A teraz przyjrzyjmy się uzyskanej macierzy korelacji:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix[ "median_house_value" ].sort_values(ascending=False)
median_house_value 1.000000
median_income 0.687160
Pokoje_na_rodzinę 0.146285
total_rooms 0.135097
housing_median_age 0.114110
households 0.064506
```

```
total_bedrooms 0.047689  
Populacja_na_rodzinę -0.021985  
population -0.026920  
longitude -0.047432  
latitude -0.142724  
Sypialnie_na_pokoje -0.259984  
Name: median_house_value, dtype: float64
```

Całkiem nieźle! Nowy atrybut `Sypialnie_na_pokoje` jest znacznie bardziej skorelowany z medianą cen mieszkań niż całkowita liczba pomieszczeń lub sypialni. Najwidoczniej mieszkania o mniejszym współczynniku liczby sypialni do liczby pomieszczeń okazują się droższe. Liczba pokojów przypadająca na rodzinę również dostarcza nam więcej informacji niż całkowita liczba pomieszczeń w dystrykcie — jest dość oczywiste, że wraz z powierzchnią domu rośnie jego cena.

Ten etap eksploracji danych nie musi być bardzo gruntowny; mamy dzięki niemu dobrze wejść w projekt i szybko uzyskać informacje pomagające w stworzeniu pierwszego odpowiednio dobrego prototypu. Jest to jednak wielokrotnie powtarzany proces: po stworzeniu i uruchomieniu projektu możesz przeanalizować uzyskiwane za jego pomocą wyniki i korzystając ze zdobytych w ten sposób informacji, powrócić do etapu eksploracji danych.

Przygotuj dane pod algorytmy uczenia maszynowego

Czas przygotować dane dla algorytmów uczenia maszynowego. Mamy kilka powodów, aby nie zajmować się tym własnoręcznie, lecz napisać funkcje odpowiedzialne za ten proces:

- W ten sposób będziemy w stanie z łatwością odtwarzać te transformacje na dowolnym zbiorze danych (np. przy okazji całkowicie świeżego zbioru danych).
- Możemy stopniowo powiększać bibliotekę funkcji przekształcających, które możemy wykorzystywać w kolejnych projektach.
- Możemy używać tych funkcji w już działającym systemie, aby przekształcać nowe dane przed przesaniem ich do algorytmów uczenia maszynowego.
- Będziemy w stanie z łatwością wypróbowywać różne rodzaje przekształceń i sprawdzać, która kombinacja transformacji sprawuje się najlepiej.

Najpierw jednak powróćmy do pierwotnego zestawu uczącego (poprzez ponowne skopiowanie obiektu `strat_train_set`). Rozdzielimy także czynniki prognostyczne od etykiet, ponieważ nie chcemy dokonywać takich samych przekształceń na obydwu typach danych (zwróć uwagę, że funkcja `drop()` tworzy kopię danych i nie wpływa na zbiór `strat_training_set`):

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

Oczyszczanie danych

Większość algorytmów uczenia maszynowego nie może działać, jeśli brakuje jakichś cech, dlatego stworzymy kilka funkcji zajmujących się tym problemem. Jak wiemy, w atrybucie `total_bedrooms` brakuje kilku wartości, należy więc to zmienić. Mamy trzy możliwości:

1. pozbyć się dystryktów zawierających brakujące dane;
2. pozbyć się całego atrybutu;
3. uzupełnić dane określoną wartością (zero, średnia, mediana itd.).

Możemy tego łatwo dokonać za pomocą metod `dropna()`, `drop()` i `fillna()`, stanowiących część klasy `DataFrame`:

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

Jeśli wybierzesz trzecie rozwiązanie, oblicz wartość mediany dla zestawu uczącego i wstaw ją następnie w miejsce brakujących wartości. Nie zapomnij zapisać wartości tejże mediany. Będzie Ci ona później potrzebna do zastąpienia brakujących wartości w zbiorze testowym na etapie oceny działania systemu; posłuży ona również do zastępowania brakujących danych w nowych przykładach.

Moduł Scikit-Learn zawiera przydatną klasę zajmującą się brakującymi wartościami: `SimpleImputer`. Korzystamy z niej w następujący sposób: najpierw tworzymy wystąpienie klasy `SimpleImputer`, w którym zaznaczamy, że chcemy zastąpić brakujące wartości każdego atrybutu medianą tego atrybutu:

```
from sklearn.impute import SimpleImputer  
  
imputer = SimpleImputer(strategy="median")
```

Mediana może być wyliczana jedynie z wartości numerycznych, musisz zatem stworzyć kopię zbioru danych niezawierającą atrybutu `ocean_proximity`:

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

Możemy teraz dopasować wystąpienie klasy `Imputer` do danych uczących za pomocą metody `fit()`:

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

Klasa `imputer` po prostu obliczyła medianę atrybutu i zachowała wyniki w zmiennej `statistics`. Jedynie w atrybucie `total_bedrooms` brakuje niektórych wartości, nie wiemy jednak, czy po uruchomieniu ostatecznej wersji systemu będą dostępne wszystkie wartości w nowych zestawach danych, dlatego najbezpieczniejszym rozwiązaniem jest zastosowanie klasy `imputer` wobec wszystkich atrybutów numerycznych:

```
>>> 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])
```

Możemy teraz użyć „wyuczonej” klasy `imputer` do przekształcenia zbioru uczącego poprzez zastąpienie brakujących wartości obliczonymi medianami:

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

W wyniku transformacji uzyskujemy klasyczną tablicę NumPy zawierającą przekształcone cechy. Bardzo łatwo jest je umieścić z powrotem w obiekcie `DataFrame`:

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

Konstrukcja modułu Scikit-Learn

Moduł Scikit-Learn wyróżnia się wyjątkowo dobrze przemyślaną konstrukcją. Oto główne założenia projektowe¹⁹ (<https://arxiv.org/abs/1309.0238>):

- **Jednolitość** — wszystkie obiekty korzystają z jednolitego, prostego interfejsu:
 - **Estymatory (funkcje oszacowujące)**. Każdy obiekt zdolny do szacowania pewnych parametrów na podstawie zbioru danych jest zwany **estymatorem** (np. klasa `imputer` jest funkcją oszacowującą). Sama operacja szacowania jest wykonywana przez metodę `fit()`, zaś jej jedynym parametrem jest zbiór danych (lub dwa zestawy danych w przypadku algorytmów uczenia nadzorowanego; drugi zbiór zawiera etykiety). Wszelkie inne parametry wpływające na przebieg operacji szacowania są uznawane za hiperparametry (np. parametr `strategy` klasy `imputer`) — muszą być one wyznaczane jako zmienne wystąpienia (generalnie w postaci parametru konstruktora).
 - **Transformatory (funkcje transformujące lub przekształcające)**. Niektóre estymatory (jak na przykład klasa `imputer`) są w stanie również przekształcać zbiór danych; są one zwane **transformatorami**. Także w tym przypadku ich interfejs nie jest skomplikowany: proces transformacji jest przeprowadzany za pomocą metody `transform()`, a jego parametr stanowi modyfikowany zbiór danych. W rezultacie zostaje zwrotny przekształcony zbiór danych. Operacja transformacji zależy przede wszystkim od wyuczonych parametrów (tak jak w przypadku klasy `imputer`). Wszystkie transformatory zawierają również metodę złożoną `fit_transform()`, która jest równoznaczna z wywołaniem metody `fit()`, a po niej `transform()` (czasami jednak ta metoda złożona jest zoptymalizowana i działa znacznie szybciej od jej elementów składowych).
 - **Predyktory (funkcje prognostyczne)**. Pewne estymatory są w stanie przewidywać wyniki na podstawie zbioru danych; są to tak zwane **predyktory**. Na przykład model `LinearRegression` z poprzedniego rozdziału stanowi predyktor: przewidzieliśmy za jego pomocą satysfakcję z życia, znając wartość PKB per capita danego kraju. Funkcja prognostyczna zawiera metodę `predict()` przyjmującą nowe zbiory danych i zwracającą zestaw powiązanych z nimi prognoz. Dodatkowo występuje tu także metoda `score()`, mierząca jakość prognoz na podstawie zbioru testowego (oraz powiązanych etykiet w przypadku algorytmów uczenia nadzorowanego)²⁰.
- **Inspekcja** — wszystkie hiperparametry estymatorów są bezpośrednio dostępne poprzez publiczne zmienne wystąpień (np. `imputer.strategy`); wszystkie wyuczone parametry funkcji oszacowującej są dostępne poprzez tego typu zmienne oznaczone na końcu podkreśnikiem (np. `imputer.statistics_`).
- **Nierozprzestrzenianie klas**. Zbiory danych nie są reprezentowane w postaci własnoręcznie przygotowanych klas, lecz jako macierze NumPy lub macierze rzadkie SciPy. Z kolei hiperparametry to standardowe ciągi znaków lub wartości liczbowe języka Python.
- **Kompozycja**. Istniejące elementy składowe są używane tak często, jak to możliwe. Przykładowo, łatwo stworzyć estymator `Pipeline` z samodzielnie dobranej sekwencji funkcji transformujących zakończonych ostatnią funkcją oszacowującą, o czym przekonamy się już niebawem.
- **Rozsądne wartości domyślne**. Moduł Scikit-Learn zawiera przemyślane wartości domyślne dla większości parametrów, dzięki czemu możemy z łatwością tworzyć bazowy system roboczy.

¹⁹ Więcej informacji na temat założeń projektowych znajdziesz w dokumencie *API design for machine learning software: experiences from the scikit-learn project*, Lars Buitinck i in., arXiv preprint arXiv: 1309.0238: 2013.

²⁰ Niektóre funkcje prognostyczne zawierają również metody mierzące pewność wyliczonych prognoz.

Obsługa tekstu i atrybutów kategorialnych

Do tej pory zajmowaliśmy się wyłącznie atrybutami numerycznymi, przejdźmy jednak teraz do atrybutów tekstowych. W używanym przez nas zestawie danych występuje tylko jeden atrybut tego typu: `ocean_proximity`. Przyjrzyjmy się wartościom jego pierwszych dziesięciu przykładów:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
   ocean_proximity
0            <1H OCEAN
1            <1H OCEAN
2           NEAR OCEAN
3             INLAND
4            <1H OCEAN
5             INLAND
6            <1H OCEAN
7             INLAND
8            <1H OCEAN
9            <1H OCEAN
10           <1H OCEAN
```

Nie jest to typowy tekst: istnieje tu ograniczona liczba wartości, z których każda reprezentuje atrybut kategorialny. Większość algorytmów uczenia maszynowego lepiej sobie radzi z liczbami, przekształćmy więc te kategorie z tekstu na wartości numeryczne. Możemy w tym celu użyć klasy `OrdinalEncoder`²¹:

```
>>> from sklearn.preprocessing import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Możemy uzyskać listę kategorii za pomocą wystąpienia zmiennej `categories_`. Jest to lista zawierająca jednowymiarową tablicę kategorii dla każdego atrybutu kategorialnego (w tym przypadku mamy do czynienia z listą zawierającą jedną tablicę, ponieważ dysponujemy jednym atrybutem kategorialnym):

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

Istnieje pewien problem z tym rozwiązańiem: algorytmy uczenia maszynowego będą uznawały, że dwie zbliżone wartości będą bardziej do siebie podobne niż do dalszych wartości. W pewnych przypadkach nie stanowi to problemu (np. przy uporządkowanych kategoriach, takich jak „zły”, „przeciętny”, „dobry”, i „znakomity”), ale oczywiście nie dotyczy to kolumny `ocean_proximity`

²¹ Klasa ta jest dostępna od wersji 0.20 modułu Scikit-Learn. Jeżeli korzystasz z wcześniejszej wersji, rozważ jego aktualizację lub użycie metody `Series.factorize()` z modułu pandas.

(np. kategorie 0 i 4 są wyraźnie bardziej do siebie podobne niż 0 i 1). Powszechnie stosowanym rozwiązaniem jest stworzenie jednego binarnego atrybutu dla każdej kategorii: jeden atrybut ma wartość 1, gdy kategorią jest <1H OCEAN (w przeciwnym wypadku otrzymuje wartość 0), inny atrybut uzyskuje wartość 1 dla kategorii INLAND (i 0 dla pozostałych kategorii) itd. Jest to tzw. **kodowanie „gorącojedynkowe”** (ang. *one-hot encoding*), ponieważ tylko jeden atrybut będzie „gorący” (będzie miał wartość 1), podczas gdy pozostałe będą „zimne” (wartość 0). Nowe atrybuty są czasami nazywane **atrybutami sztucznymi** (ang. *dummy attributes*). Moduł Scikit-Learn zawiera koder OneHotEncoder, konwertujący kategoryalne wartości całkowite na wektory „gorącojedynkowe”²²:

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

Zauważmy, że w wyniku powyższej operacji uzyskujemy **macierz rzadką** (ang. *sparse matrix*) SciPy, a nie tablicę NumPy. Okazuje się to bardzo użyteczne w przypadku korzystania z atrybutów kategoryalnych zawierających tysiące kategorii. Kodowanie „gorącojedynkowe” generuje macierz składającą się z tysięcy kolumn, a cała macierz zawiera tylko jedną wartość 1 na każdy wiersz, pozostałe miejsca są wypełnione zerami. Przechowywanie tak wielu zer w pamięci byłoby marnotrawstwem, dlatego zapisana zostaje nie cała macierz rzadka, lecz położenie poszczególnych elementów niezerowych. Przeważnie możemy z niej korzystać jak ze zwykłej macierzy dwuwymiarowej²³, jeśli jednak naprawdę chcesz ją przekształcić w tablicę NumPy (gęstą), wystarczy wywołać metodę `toarray()`:

```
>>> housing_cat_1hot.toarray()  
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.]])
```

Możemy również uzyskać listę kategorii za pomocą wystąpienia zmiennej `categories_` kodera:

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



Jeżeli atrybut kategoryalny zawiera znaczną liczbę kategorii (np. kod kraju, zawód, gatunek), to w wyniku kodowania „gorącojedynkowego” będziemy otrzymywać znaczną liczbę cech wejściowych. Może to spowolnić proces uczenia i zmniejszyć skuteczność. W takim przypadku warto zastąpić wejściowe dane kategoryalne przydatnymi cechami numerycznymi powiązanymi z kategoriami, np. możesz zastąpić cechę `ocean_proximity` odlegością do oceanu (na drodze analogii kod kraju może zostać zastąpiony populacją państwa i PKB per capita). Ewentualnie możesz

²² Przed wydaniem wersji 0.20 modułu Scikit-Learn metoda ta była w stanie kodować wyłącznie kategoryalne wartości całkowite, jednak od wersji 0.20 obsługuje również inne typy danych wejściowych, w tym kategoryalne dane tekstowe.

²³ Szczegóły znajdziesz w dokumentacji modułu SciPy.

zamiast każdej kategorii wprowadzić poznawalny, małowymiarowy wektor zwany **wektorem właściwościowym** (ang. *embedding*). Reprezentacja każdej kategorii byłaby poznawana w trakcie uczenia. Jest to przykład **uczenia się reprezentacji** (ang. *representation learning*) (zob. rozdziały 13. i 17.).

Niestandardowe transformatory

Mimo że moduł Scikit-Learn zawiera wiele przydatnych funkcji przekształcających, często jesteśmy zmuszeni pisać własne transformatory wykonujące operacje niestandardowego oczyszczania danych lub łączenia określonych atrybutów. Chcemy, aby nasze funkcje transformujące działały płynnie z funkcjami modułu Scikit-Learn (np. z funkcją potokowania), skoro zaś moduł Scikit-Learn polega na technice inferencji typów (ang. *duck typing*), a nie dziedziczeniu, wystarczy stworzyć klasę i zaimplementować trzy metody: `fit()` (zwierającą wartość obiektu `self`), `transform()` oraz `fit_transform()`.

To ostatnie możemy uzyskać „za darmo”, dodając bazową klasę `TransformerMixin`. Jeśli dołączasz klasę bazową `BaseEstimator` (i unikasz zmiennych `*args` oraz `**kargs` w konstruktorze), uzyskujesz także dostęp do dwóch dodatkowych metod (`get_params()` i `set_params()`), które przydadzą się do automatycznego strojenia hiperparametrów. Na przykład, poniżej przedstawiamy małutką klasę transformującą dodającą omówione wcześniej połączone atrybuty:

```
from sklearn.base import BaseEstimator, TransformerMixin

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

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # Żadnych zmiennych *args ani **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # Nie robi nic innego
    def transform(self, X):
        Pokoje_na_rodzinę = X[:, rooms_ix] / X[:, households_ix]
        Populacja_na_rodzinę = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            Sypialnie_na_pokoje = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, Pokoje_na_rodzinę, Populacja_na_rodzinę,
                        Sypialnie_na_pokoje]
        else:
            return np.c_[X, Pokoje_na_rodzinę, Populacja_na_rodzinę]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

W tym przykładzie transformator zawiera jeden hiperparametr, `add_bedrooms_per_room`, którego domyślną wartością jest `True` (warto dawać rozsądne wartości domyślne). Dzięki temu hiperparametrowi z łatwością możesz sprawdzić, czy dodanie atrybutu `Sypialnie_na_pokoje` bardziej pomoże, czy zaszkodzi algorytmom uczenia maszynowego. Mówiąc bardziej ogólnie, za pomocą hiperparametru możesz bramkować każdy etap przygotowywania danych, którego skuteczności nie jesteś pewna/pewny. Im bardziej zautomatyzujesz etapy przygotowywania danych, tym więcej ich kombinacji możesz wypróbować, dzięki czemu znacznie zwiększasz szansę na odkrycie znakomitej kombinacji (i zaoszczędzenie czasu).

Skalowanie cech

Jednym z najważniejszych przekształceń dokonywanych na danych jest **skalowanie cech** (ang. *feature scaling*). Większość algorytmów uczenia maszynowego słabo sobie radzi z atrybutami numerycznymi znajdującymi się w różnych zakresach skali. Dotyczy to również naszego zbioru danych: całkowita liczba pomieszczeń mieści się w zakresie od 6 do 39 320, z kolei wartości mediany dochodów to zakres zaledwie od 0 do 15. Zauważmy, że generalnie nie jest wymagane skalowanie wartości docelowych.

Najczęściej są stosowane dwa rodzaje skalowania wszystkich atrybutów do jednego poziomu: **skalowanie min. – max.** (ang. *min-max scaling*) i **standaryzacja** (ang. *standarization*).

Skalowanie min. – max. (zwane przez wiele osób **normalizacją**) jest najprostszym procesem: wartości są tak skalowane, że mieszczą się w zakresie pomiędzy 0 i 1. Dokonujemy tego, odejmując od danej wartości minimalną i dzieląc otrzymany wynik przez różnicę wartości maksymalnej i minimalnej. W module Scikit-Learn służy do tego funkcja transformująca `MinMaxScaler`. Zawiera ona hiperparametr `feature_range`, pozwalający zmieniać zakres skali, jeśli z jakiegoś powodu nie odpowiada Ci domyślny zakres 0 – 1.

Mechanizm standaryzacji jest odmienny: najpierw odejmujemy od danej wartości średnią (czyli średnia w standaryzowanych próbkach zawsze wynosi 0), a następnie dzielimy ją przez odchylenie standardowe, dzięki czemu wynikowy rozkład ma wariancję jednostkową. W przeciwieństwie do skalowania min. – max. standaryzacja nie ogranicza skalowanych wartości do określonego zakresu, co w przypadku niektórych algorytmów stanowi pewien problem (np. sieci neuronowe często oczekują wartości wejściowych mieszczących się w zakresie 0 – 1). Z drugiej strony, standaryzacja jest znacznie mniej wrażliwa na elementy odstające. Założymy, na przykład, że przez pomyłkę mediana dochodów dla danego dystryktu jest równa 100. W przypadku skalowania min. – max. wszystkie pozostałe wartości z przedziału 0 – 15 zostałyby umieszczone w zakresie 0 – 0,15, natomiast nie miałyby to tak wielkiego wpływu na standaryzowane wyniki. Standaryzację w module Scikit-Learn uzyskujemy za pomocą transformatora `StandardScaler`.



Podobnie jak w przypadku wszystkich transformacji, należy dostosować funkcje skalujące wyłącznie do danych uczących, nie do całego zbioru danych (włącznie ze zbiorem testowym). Dopiero po wyuczeniu wobec danych uczących możesz użyć tych funkcji do przekształcania danych testowych (oraz nowych przykładów).

Potoki transformujące

Jak widać, należy przeprowadzać wiele operacji przekształcania we właściwej kolejności. Na szczęście możemy skorzystać z klasy `Pipeline`, pomagającej wyznaczyć odpowiednią sekwencję transformacji. Poniżej przedstawiamy niewielki potok stosowany wobec atrybutów numerycznych:

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

num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Konstruktor Pipeline przyjmuje listę par nazwa/estymator definiującą sekwencję operacji. Wszystkie elementy oprócz ostatniego estymatora muszą być funkcjami przekształcającymi (tj. muszą zawierać metodę `fit_transform()`). Nazwy mogą być dowolne (pod warunkiem że są niepowtarzalne i nie zawierają podwójnego podkreśnika: `__`); przydadzą się one później podczas strojenia hiperparametrów.

W momencie wywołania metody `fit()` następuje sekwencyjne wywołanie metody `fit_transform()` wobec wszystkich transformatorów, przekazanie wyniku każdego wywołania w postaci parametru do następnego wywołania aż do osiągnięcia ostatniego estymatora, dla którego zostaje wywołana metoda `fit()`.

Potok odsłania te same metody, co ostatni estymator. W tym przykładzie ostatnią funkcją prognozytorską jest `StandardScaler`, będąca w istocie transformatorem, dlatego potok zawiera metodę `transform()`, wykonującą wszystkie przekształcenia wobec sekwencji danych (a także oczywiście metodę `fit_transform()`, z której skorzystaliśmy).

Do tej pory zajmowaliśmy się oddzielnie kolumnami kategorialnymi i numerycznymi. Byłoby wygodniej, gdybyśmy dysponowali pojedynczym transformatorem przetwarzającym wszystkie kolumny i dobierający odpowiednie transformacje do poszczególnych typów kolumn. W tym celu zaprezentowano w wersji 0.20 modułu Scikit-Learn klasę `ColumnTransformer`, a dodatkowa znakomita informacja jest taka, że współpracuje ona z obiektami `DataFrame`. Wprowadźmy za jej pomocą wszystkie przekształcenia zestawu danych `Housing`:

```
from sklearn.compose import ColumnTransformer

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

full_pipeline = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", OneHotEncoder(), cat_attribs),
])

housing_prepared = full_pipeline.fit_transform(housing)
```

Najpierw importujemy klasę `ColumnTransformer`, następnie uzyskujemy listy nazw kolumn numerycznych i nazw kolumn kategorialnych, po czym konstruujemy obiekt `ColumnTransformer`. Konstruktor wymaga listy krotek, gdzie każda krotka zawiera nazwę²⁴, transformator, a także listę nazw (lub indeksów) kolumn, wobec których powinien zostać użyty transformator. W tym przykładzie decydujemy, że kolumny numeryczne powinny być przekształcane za pomocą uprzednio zdefiniowanego obiektu `num_pipeline`, natomiast do transformacji kolumn kategorialnych wyznaczamy `OneHotEncoder`. Na koniec stosujemy tę klasę `ColumnTransformer` na zestawie danych `Housing`: każdy transformator zostaje użyty wobec odpowiednich kolumn, a wyniki wzduż drugiej osi są łączone (transformatory muszą zwracać tę samą liczbę rzędów).

Zwróć uwagę, że obiekt `OneHotEncoder` zwraca macierz rzadką, natomiast `num_pipeline` gęstą. W przypadku występowania takiej mieszanki macierzy rzadkich i gęstych klasa `ColumnTransformer` oszacowuje gęstość macierzy końcowej (tzn. współczynnik występowania komórek niezerowych) i zwraca

²⁴ Podobnie jak w przypadku potoków nazwa może być dowolna, pod warunkiem że nie zawiera podwójnych podkreśników.

macierz rzadką, jeżeli gęstość jest mniejsza od ustalonego progu (domyślnie `sparse_threshold=0.3`). W tym przykładzie zostaje zwrócona macierz gęsta. I to tyle! Wykorzystujemy potok przetwarzania wstępniego przyjmujący pełny zestaw danych *Housing* i stosujący wobec każdej kolumny odpowiednie przekształcenia.



Zamiast korzystać z transformatora możesz wyznaczyć łańcuch znaków "drop", jeżeli chcesz, aby kolumny zostały usunięte, lub "passthrough", jeśli mają pozostać niezmienne. Domyślnie pozostałe kolumny (te, które nie zostały wymienione w liście) zostaną usunięte, ale możesz wyznaczyć parametr `remainder` w dowolnym transformatorze (lub łańcuchu znaków "passthrough"), jeśli chcesz, aby były one przetwarzane inaczej.

Jeżeli korzystasz z wersji 0.19 modułu Scikit-Learn lub wcześniejszej, możesz użyć innej biblioteki, np. `sklearn-pandas`, lub przygotować własny, niestandardowy transformator mający taką samą funkcjonalność jak `ColumnTransformer`. Ewentualnie masz również do dyspozycji klasę `FeatureUnion`, która może stosować odmienne transformatory i łączyć ich wyniki. Nie możesz jednak wyznaczać różnych kolumn dla każdego transformatora — transformatory są stosowane wobec wszystkich danych. Istnieje możliwość obejścia tego ograniczenia poprzez wykorzystanie niestandardowego transformatora przeznaczonego do dobioru kolumn (przykład znajdziesz w notatniku Jupyter).

Wybór i uczenie modelu

Nareszcie! Określiliśmy ramy problemu, zdobyliśmy i przeanalizowaliśmy dane, przetestowaliśmy zestawy uczący i testowy, a także napisaliśmy potoki służące do automatycznego oczyszczania i przygotowywania danych pod algorytmy uczenia maszynowego. Jesteśmy gotowi, aby wybrać i wytrenować model uczenia maszynowego.

Trenowanie i ocena modelu za pomocą zbioru uczącego

Mam dobre wieści do przekazania: dzięki wszystkim wcześniejszym czynnościom dalsze etapy będą znacznie prostsze, niż Ci się wydaje. Wyuczmy najpierw model regresji liniowej, podobnie jak dokonaliśmy tego w poprzednim rozdziale.

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

I już! Masz teraz do dyspozycji działający model regresji liniowej. Sprawdźmy kilka przykładów ze zbioru uczącego:

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

Model działa, chociaż prognozy nie są zbyt dokładne (np. pierwsza prognoza odstaje od etykiety niemal o 40%). Zmierzmy błąd RMSE modelu regresji dla całego zbioru uczącego za pomocą funkcji `mean_squared_error()`:

```
>>> 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
68628.19819848922
```

Lepsze to niż nic, ale wynik ten nie jest powalający: wartości atrybutu `Medianan cen mieszkań` dla większości dystryktów mieścią się w zakresie pomiędzy 120 000 a 265 000 dolarów, dlatego standar-dowy błąd predykcji rzędu 68 628 dolarów nie jest zbyt satysfakcyjny. Widzimy tu klasyczny przykład niedotrenowania modelu wobec danych uczących. Taka sytuacja oznacza, że cechy nie do-starczają odpowiedniej ilości informacji pozwalających na uzyskanie dobrych prognoz albo że model nie jest wystarczająco dobry. Jak wiemy z poprzedniego rozdziału, podstawowymi sposobami radzenia sobie z problemem niedotrenowania są wybór potężniejszego algorytmu, wprowadzenie lepszych cech lub zmniejszenie ograniczeń modelu. Nasz model nie jest regularyzowany, dlatego odpada ostatnia możliwość. Możemy spróbować dodać więcej cech (np. logarytm z populacji), najpierw jednak sprawdzmy bardziej skomplikowany model i zobaczymy, jak sobie poradzi.

Wy trenujemy model `DecisionTreeRegressor`. Jest to zaawansowany model potrafiący wyszukiwać w danych skomplikowane, nieliniowe zależności (w rozdziale 6. zajmiemy się szczegółowo tematem drzew decyzyjnych). Użyty kod powinien wyglądać już znajomo:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

Po wyuczeniu modelu sprawdzmy jego wydajność wobec zbioru uczącego:

```
>>> 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
```

Że jak? Żadnego błędu? Czyżby ten model był stu procentowo bezbłędny? Oczywiście jest o wiele bardziej prawdopodobne, że model po prostu został znacznie przetrenowany. Skąd mamy mieć pewność? Wiemy już, że nie chcemy używać zbioru uczącego, dopóki nie będziemy w pełni przekonani o skuteczności wybranego modelu, dlatego musimy wykorzystać część zbioru uczącego do treno-wania, a część do ocenienia modelu.

Dokładniejsze ocenianie za pomocą sprawdzianu krzyżowego

Jednym ze sposobów oceniania modelu drzewa decyzyjnego byłoby wykorzystanie funkcji `train_test_split()` do rozdzielenia zestawu uczącego na podzbiory trenujący i walidacyjny, następ-nie wyuczenie naszego modelu przy użyciu tego zbioru trenującego, po czym ewaluacja za pomocą zbioru walidacyjnego. Proces ten oznacza dodatkową pracę, ale nie jest zbyt skomplikowany i całkiem dobrze spełnia swoje zadanie.

Doskonałą alternatywą jest użycie funkcji k-krotnego sprawdzianu krzyżowego (*kroswalidacji*, ang. *k-fold cross-validation*). Za pomocą poniższego kodu zbiór uczący zostaje losowo rozdzielony na 10 oddzielnych **podzbiorów** (ang. *folds*), następnie przeprowadzane jest dziesięciokrotne trenowanie i ocenianie modelu drzewa decyzyjnego (za każdym razem zostaje wybrany inny podzbior do oceny wydajności modelu, a dziewięć pozostałych służy do uczenia). W konsekwencji użyjemy tablicę zawierającą 10 wyników ewaluacji:

```
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)
```



Funkcja sprawdzianu krzyżowego będąca częścią modułu Scikit-Learn oczekuje funkcji użyteczności (im większa wartość, tym lepiej), a nie funkcji kosztu (im mniejsza wartość, tym lepiej), dlatego funkcja zliczająca wynik stanowi w rzeczywistości przeciwieństwo błędu MSE (np. ujemną wartość), dlatego w powyższym kodzie obliczamy wartość `-scores` przed określeniem pierwiastka kwadratowego.

Spójrzmy na wyniki:

```
>>> def display_scores(scores):
...     print("Wyniki:", scores)
...     print("Średnia:", scores.mean())
...     print("Odchylenie standardowe:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Wyniki: [70194.33680785 66855.16363941 72432.58244769 70758.73896782
         71115.88230639 75585.14172901 70262.86139133 70273.6325285
         75366.87952553 71231.65726027]
Średnia 71407.68766037929
Odchylenie standardowe: 2439.4345041191004
```

Model drzewa decyzyjnego nie wygląda teraz już tak dobrze. W rzeczywistości wygląda na to, że sprawuje się nawet gorzej niż model regresji liniowej! Zwróć uwagę, że za pomocą sprawdzianu krzyżowego możemy nie tylko oszacować wydajność naszego modelu, lecz także zmierzyć precyzję oszacowań (tj. odchylenie standardowe). Wynik uzyskany dla modelu drzewa decyzyjnego wynosi w przybliżeniu 71 407, plus minus 2439. Nie uzyskalibyśmy takich informacji, gdybyśmy skorzystali wyłącznie z jednego zbioru walidacyjnego. Jednak ceną kroswalidacji jest konieczność kilkukrotnego uczenia modelu, dlatego rozwiążanie to nie zawsze jest dostępne.

Dla pewności przeprowadźmy sprawdzian krzyżowy dla modelu regresji liniowej:

```
>>> 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)
Wyniki: [66782.73843989 66960.118071 70347.95244419 74739.57052552
         68031.13388938 71193.84183426 64969.63056405 68281.61137997
         71552.91566558 67665.10082067]
Średnia: 69052.46136345083
Odchylenie standardowe: 2731.674001798348
```

Zgadza się: model drzewa decyzyjnego ulega tak silnemu przetrenowaniu, że osiąga gorsze wyniki od modelu regresji liniowej.

Sprawdźmy jeszcze jeden model: RandomForestRegressor. Jak się przekonamy w rozdziale 7., mechanizm działania modeli losowego lasu polega na uczeniu wielu drzew decyzyjnych za pomocą różnych podzbiorów cech, po czym następuje uśrednienie otrzymanych prognoz. Konstruowanie modelu na podwalinach wielu innych modeli nazywamy metodami **uczenia zespołowego** (ang. *ensemble learning*) — nieraz dzięki temu rozwiązaniu uzyskujemy jeszcze lepszą wydajność stosowanych algorytmów. Pominiemy większą część kodu, ponieważ w większości wygląda on tak samo, jak w przypadku poprzednich modeli:

```
>>> from sklearn.ensemble import RandomForestRegressor  
>>> forest_reg = RandomForestRegressor()  
>>> forest_reg.fit(housing_prepared, housing_labels)  
>>> [...]  
>>> forest_rmse  
18603.515021376355  
>>> display_scores(forest_rmse_scores)  
Wyniki: [49519.80364233 47461.9115823 50029.02762854 52325.28068953  
49308.39426421 53446.37892622 48634.8036574 47585.73832311  
53490.10699751 50021.5852922 ]  
Średnia: 50182.303100336096  
Odchylenie standardowe: 2097.0810550985693
```

O proszę, teraz jest znacznie lepiej: model losowego lasu prezentuje się bardzo obiecująco. Zauważmy jednak, że wynik dla zestawu uczącego jest ciągle mniejszy niż dla zbiorów walidacyjnych, co oznacza, że model ten w dalszym ciągu ulega przetrenowaniu. Mamy do dyspozycji następujące rozwiązania: uproszczenie modelu, jego ograniczenie (np. regularyzacja) lub pozyskanie znacznie większej liczby przykładów uczących. Zanim jednak wybierzemy model losowego lasu, powinniśmy wypróbować wiele innych modeli, reprezentujących różne rodzaje algorytmów uczenia maszynowego (np. kilka maszyn wektorów nośnych przy użyciu różnych jąder i być może sieć neuronową), bez spędzania dużej ilości czasu na strojeniu hiperparametrów. Naszym celem jest stworzenie krótkiej listy (od dwóch do pięciu pozycji) najbardziej obiecujących modeli.



Zapisuj każdy model, z którym eksperymentujesz, dzięki czemu będziesz w stanie szybko powracać do dowolnego modelu. Nie zapomnij zachowywać zarówno wartości hiperparametrów i wyuczonych parametrów, jak i wyników sprawdzianu krzyżowego, a być może nawet samych prognoz. W ten sposób możesz z łatwością porównywać wyniki poszczególnych rodzajów modeli i zwracane przez nie błędy. Możemy zapisywać modele stworzone w module Scikit-Learn za pomocą modułu `pickle` lub `joblib` — ten drugi jest skuteczniejszy w serializowaniu dużych macierzy NumPy (możesz zainstalować tę bibliotekę za pomocą modułu pip):

```
import joblib  
  
joblib.dump(my_model, "mój_model.pkl")  
# a później...  
my_model_loaded = joblib.load("mój_model.pkl")
```

Wyreguluj swój model

Załóżmy, że masz już sporządzoną listę obiecujących modeli. Musisz je teraz dostroić. Możemy tego dokonać na kilka sposobów.

Metoda przeszukiwania siatki

Jednym z rozwiązań jest własnoręczne dobieranie wartości hiperparametrów, dopóki nie uzyskasz ich znakomitej kombinacji. Jest to bardzo zmudne zajęcie i być może nie masz tyle czasu, aby sprawdzić wszystkie możliwości.

Zamiast tego możemy zlecić poszukiwania obiekowi GridSearchCV. Wystarczy podać interesujące nas hiperparametry oraz ich proponowane wartości, a wszystkie kombinacje zostaną ocenione za pomocą sprawdzianu krzyżowego. Na przykład poniższy kod poszukuje najlepszej kombinacji wartości hiperparametrów dla modelu 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',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```



Jeśli nie wiesz, jaką wartość wyznaczyć danemu hiperparametrowi, najprostszym rozwiązaniem jest wstawienie kolejnych wielokrotności liczby 10 (lub mniejszych wartości w przypadku bardziej szczegółowego przeszukiwania, co zostało zaprezentowane na przykładzie hiperparametru `n_estimators`).

Parametr `param_grid` najpierw oszacowuje wszystkie $3 \times 4 = 12$ kombinacji wartości hiperparametrów `n_estimators` i `max_features` zdefiniowanych w pierwszym obiekcie `dict` (na razie nie jest ważne, do czego służą te hiperparametry; wróćmy do nich w rozdziale 7.), a następnie $2 \times 3 = 6$ kombinacji wartości parametrów z drugiego obiektu `dict`, w tym przypadku jednak wyłączamy hiperparametr `bootstrap` (wartość `False`; jego domyślną wartością jest `True`).

Mechanizm przeszukiwania siatki sprawdzi $12+6 = 18$ kombinacji wartości hiperparametrów modelu `RandomForestRegressor`, a każdy model zostanie pięciokrotnie wyuczony (gdźże korzystamy z pięciokrotnego sprawdzianu krzyżowego). Innymi słowy, ostatecznie zostanie przeprowadzonych $18 \times 5 = 90$ przebiegów uczenia! Może to zająć trochę czasu, ale ostatecznie możesz uzyskać najlepszą kombinację hiperparametrów, np. taką:

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



Wartości 8 i 30 są największe spośród tych, które zostały przez nas wstawione do kodu, dlatego prawdopodobnie warto byłoby przeprowadzić kolejne przeszukiwanie siatki, tym razem z podanymi większymi wartościami — możemy dzięki temu uzyskać jeszcze lepsze wyniki.

Możemy również bezpośrednio uzyskać najlepszy estymator:

```
>>> 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=None,
oob_score=False, random_state=None, verbose=0, warm_start=False)
```



Jeżeli obiekt GridSearchCV zostanie zainicjowany z parametrem refit=True (jest to wartość domyślna), to po znalezieniu najlepszego estymatora za pomocą sprawdzianu krzyżowego ta funkcja oszacowująca będzie wykorzystana wobec całego zbioru uczącego. Zazwyczaj jest to dobre rozwiązanie, ponieważ im więcej danych zostanie użytych, tym bardziej wzrośnie wydajność modelu.

Oczywiście są również dostępne wyniki ewaluacji:

```
>>> cvres = grid_search.cv_results_
>>> for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
...     print(np.sqrt(-mean_score), params)
...
63669.05791727153 {'max_features': 2, 'n_estimators': 3}
55627.16171305252 {'max_features': 2, 'n_estimators': 10}
53384.57867637289 {'max_features': 2, 'n_estimators': 30}
60965.99185930139 {'max_features': 4, 'n_estimators': 3}
52740.98248528835 {'max_features': 4, 'n_estimators': 10}
50377.344409590376 {'max_features': 4, 'n_estimators': 30}
58663.84733372485 {'max_features': 6, 'n_estimators': 3}
52006.15355973719 {'max_features': 6, 'n_estimators': 10}
50146.465964159885 {'max_features': 6, 'n_estimators': 30}
57869.25504027614 {'max_features': 8, 'n_estimators': 3}
51711.09443660957 {'max_features': 8, 'n_estimators': 10}
49682.25345942335 {'max_features': 8, 'n_estimators': 30}
62895.088889905004 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54658.14484390074 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
59470.399594730654 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52725.01091081235 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
57490.612956065226 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51009.51445842374 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```



Pamiętaj, że możesz traktować niektóre etapy przygotowywania danych jako hiperparametry. Przykładowo, mechanizm przeszukiwania siatki automatycznie sprawdzi, czy należy dodać cechy, co do których nie masz pewności (np. hiperparametr add_bedrooms_per_room wyliczony za pomocą transformatora CombinedAttributesAdder). Może on w podobny sposób wyszukiwać najlepszą metodę radzenia sobie z elementami odstającymi, brakującymi cechami, doborem cech itd.

W tym przykładzie uzyskujemy najlepsze rozwiązanie, wyznaczając wartość 8 dla hiperparametru `max_features`, a 30 dla hiperparametru `n_estimators`. Błąd RMSE dla tej kombinacji wynosi 49 682, co stanowi nieco lepszy wynik niż w przypadku standardowych wartości tych hiperparametrów (50 182). Gratulacje! Właśnie skutecznie dostroiliśmy nasz najlepszy model!

Metoda losowego przeszukiwania

Mechanizm przeszukiwania siatki przydaje się wtedy, gdy chcemy sprawdzić względnie niewielką liczbę kombinacji (tak jak w powyższym przykładzie), jednak jeśli **przestrzeń przeszukiwania** hiperparametrów jest bardzo duża, lepiej skorzystać z obiektu `RandomizedSearchCV`. Klasa ta jest używana w bardzo podobny sposób, jak `GridSearchCV`, tutaj jednak nie są sprawdzane wszystkie możliwe kombinacje, lecz następuje ewaluacja określonej liczby losowych kombinacji poprzez dobór losowej wartości hiperparametru w każdym przebiegu. Rozwiązanie to cechują dwie zalety:

- Jeśli zezwolisz na, dajmy na to, 1000 przebiegów losowego przeszukiwania, zostanie sprawdzonych 1000 wartości dla każdego hiperparametru (nie jesteśmy ograniczeni wyłącznie do kilku wartości dla każdego hiperparametru, jak w metodzie przeszukiwania siatki).
- Zyskujesz większą kontrolę nad mocą obliczeniową, jaką chcesz przeznaczyć na wyszukiwanie wartości hiperparametrów — wystarczy regulować liczbę przebiegów.

Metody zespołowe

Innym sposobem strojenia modelu jest próba połączenia najlepiej spisujących się modeli. Grupa (lub „zespół”) zawsze ma większą wydajność od jej elementów składowych (tak jak w przypadku modelu losowego lasu składającego się z poszczególnych drzew decyzyjnych), zwłaszcza jeśli poszczególne modele popełniają odmienne rodzaje błędów. Przyjrzymy się temu zagadnieniu dokładniej w rozdziale 7.

Analizuj najlepsze modele i ich błędy

Często uzyskasz wiele informacji na temat problemu, sprawdzając najlepsze modele. Na przykład model `RandomForestRegressor` może wskazywać względną istotność każdego atrybutu w generowaniu dokładnych prognoz:

```
>>> 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])
```

Połączmy teraz te wyniki istotności z odpowiadającymi im nazwami atrybutów:

```
>>> extra_attribs = ["Pokoje_na_rodzinę", "Populacja_na_rodzinę", "Sypialnie_na_pokoje"]
>>> cat_encoder = full_pipeline.named_transformers_[“cat”]
>>> 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)
```

```
[ (0.3661589806181342, 'median_income'),
  (0.1647809935615905, 'INLAND'),
  (0.10879295677551573, 'Populacja_na_rodzinę'),
  (0.07334423551601242, 'longitude'),
  (0.0629090704826203, 'latitude'),
  (0.05641917918195401, 'Pokoje_na_rodzinę'),
  (0.05335107734767581, 'Sypialnie_na_pokoje'),
  (0.041143798478729635, 'housing_median_age'),
  (0.014874280890402767, 'population'),
  (0.014672685420543237, 'total_rooms'),
  (0.014257599323407807, 'households'),
  (0.014106483453584102, 'total_bedrooms'),
  (0.010311488326303787, '<1H OCEAN'),
  (0.002856474637320158, 'NEAR OCEAN'),
  (0.00196041559947807, 'NEAR BAY'),
  (6.028038672736599e-05, 'ISLAND')]
```

Dzięki uzyskanym informacjom możesz zrezygnować z niektórych mniej przydatnych cech (przykładowo, wygląda na to, że tylko kategoria ocean_proximity jest użyteczna, dlatego warto spróbować usunąć pozostałe).

Przyjrzyj się również błędem popełnianym przez Twój system, następnie postaraj się poznać ich przyczynę oraz znaleźć rozwiązanie (dodać cechy lub usunąć nieprzydatne, pozbyć się elementów odstających itd.).

Oceń system za pomocą zbioru testowego

Po etapie strojenia modeli w końcu uzyskasz system sprawujący się wystarczająco dobrze. Nadszedł czas, aby ocenić jego wydajność za pomocą zbioru danych testowych. Proces ten nie wyróżnia się niczym szczególnym; pobierz przykłady i etykiety z zestawu testowego, wykorzystaj potok full_pipeline do przekształcenia tych danych (wywołaj funkcję transform(), a nie fit_transform(): nie chcesz dopasowywać modelu do danych testowych!) i oceń ostateczny model za pomocą tego zbioru testowego.

```
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)
final_rmse = np.sqrt(final_mse) #=> otrzymujemy wynik 47730,2
```

W pewnych sytuacjach takie punktowe oszacowanie błędu uogólniania może okazać się niewystarczająco przekonujące: a jeżeli model jest zaledwie o 0,1% lepszy od obecnie używanego w środowisku produkcyjnym? Być może chcesz się dowiedzieć, jak precyzyjne jest to oszacowanie. W tym celu możesz obliczyć 95% **przedział ufności** (ang. *confidence interval*) dla błędu uogólniania za pomocą funkcji `scipy.stats.t.interval()`:

```
>>> from scipy import stats
>>> confidence = 0.95
>>> squared_errors = (final_predictions - y_test) ** 2
>>> np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
```

```
...  
...  
...  
loc=squared_errors.mean(),  
scale=stats.sem(squared_errors)))  
  
array([45685.10470776, 49691.25001878])
```

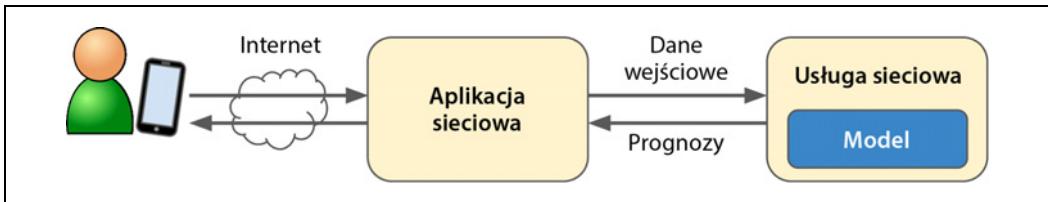
W przypadku intensywnego strojenia hiperparametrów będziemy zazwyczaj otrzymywać nieco gorsze wyniki niż zmierzone za pomocą sprawdzianu krzyżowego (gdyż system zostaje dostrojony do danych walidacyjnych i prawdopodobnie nie będzie sobie tak dobrze radził z nieznanymi próbkami). Akurat tak się nie dzieje w naszym przykładzie, ale jeśli Ci się to przytrafi, musisz powstrzymać chęć dalszego poprawiania hiperparametrów w celu uzyskania dobrych wyników dla zbioru testowego; wszelkie usprawnienia już raczej nie poprawią procesu uogólniania wobec nowych danych.

Teraz następuje faza przeduruchomieniowa projektu: musisz przedstawić rozwiązańe (zwracając szczególną uwagę na to, czego się nauczyłaś/nauczyłeś, co zadziałało, a co nie, jakie przyjęłaś/przyjęłeś założenia, a także zaznaczając ograniczenia systemu), wszystko udokumentować, a także stworzyć eleganckie prezentacje zawierające wyraźne wizualizacje wyników oraz łatwe do zapamiętania stwierdzenia (np. „mediana dochodów stanowi główny predyktor cen mieszkań”). W przykładzie z zestawem danych *Housing* ostateczna wydajność systemu okazuje się nie lepsza od oszacowań ekspertów, które często były nawet o 20% inne od rzeczywistych cen, mimo to jednak warto uruchomić taki model, choćby po to, aby zaoszczędzić czas ekspertów — mogą oni wtedy zająć się ciekawszymi lub bardziej produktywnymi zadaniami.

Uruchom, monitoruj i utrzymuj swój system

Znakomicie! Otrzymaliśmy pozwolenie na uruchomienie systemu! Musisz teraz przygotować nasze rozwiązanie do warunków produkcyjnych (tzn. dopieścić kod, napisać dokumentację i testy itd.). Nastepnym etapem jest wdrożenie modelu do środowiska produkcyjnego. Jednym ze sposobów jest zapisanie wyuczonego modelu Scikit-Learn (np. za pomocą modułu `joblib`), dołączenie gotowego potoku przetwarzania wstępnego i prognozowania, następnie wczytanie tego wytrenowanego modelu w środowisku produkcyjnym i wykorzystanie go do uzyskiwania predykcji poprzez wywołanie metody `predict()`. Na przykład być może dany model będzie wykorzystywany na stronie internetowej: użytkownik wpisze jakieś dane dotyczące nowego dystryktu, a następnie kliknie przycisk *Oszacuj cenę*. Do serwera sieciowego zostanie wysłane zapytanie zawierające dostarczone dane, skąd trafi do aplikacji sieciowej, a na koniec kod wywoła metodę `predict()` (model powinien zostać wczytany po uruchomieniu serwera, nie chcesz, żeby to się działało za każdym razem, gdy model zostaje użyty). Ewentualnie możesz umieścić model w wyspecjalizowanej usłudze sieciowej, do której aplikacja sieciowa będzie przesyłać zapytania poprzez REST API²⁵ (rysunek 2.17). W ten sposób ułatwiamy proces aktualizowania modelu do nowych wersji bez konieczności przerywania działania głównej aplikacji. Jednocześnie upraszczamy skalowanie, ponieważ możemy uruchamiać dowolną liczbę usług sieciowych oraz równoważyć obciążenie zapytań przesyłanych przez aplikację sieciową. Ponadto aplikacja sieciowa może umożliwiać korzystanie z innych języków, nie tylko Pythona.

²⁵ Mówiąc krótko, REST (lub RESTful) API to bazujący na protokole HTTP interfejs API zgodny z pewnymi konwencjami, takimi jak np. wykorzystywanie czasowników HTTP do odczytywania, aktualizowania, tworzenia lub usuwania zasobów (odpowiednio GET, POST, PUT i DELETE) oraz definiowanie danych wejściowych i wyjściowych w standardzie JSON.



Rysunek 2.17. Model wdrożony jako usługa sieciowa i wykorzystywany przez aplikację sieciową

Kolejną popularną strategią jest wdrożenie modelu w chmurze, np. serwisie Google Cloud AI Platform (znanym wcześniej jako Google Cloud ML Engine): wystarczy zapisać swój model za pomocą modułu joblib i przesłać go do Google Cloud Storage (GCS), następnie przejść do Google Cloud AI Platform i stworzyć nową wersję modelu, wyznaczając plik umieszczony w magazynie GCS. I to wszystko! Uzyskujesz dostęp do prostej usługi sieciowej, która przejmuje zadania równoważenia obciążenia i skalowania. Przyjmuje zapytania JSON zawierające dane wejściowe (np. dotyczące dystryktu) i zwraca odpowiedzi JSON przechowujące prognozy. Możesz następnie użyć takiej usługi sieciowej na swojej własnej stronie internetowej (lub w dowolnym innym środowisku produkcyjnym). Jak się przekonasz w rozdziale 19., wdrażanie modeli TensorFlow w usłudze AI Platform nie różni się zasadniczo od wdrażania modeli Scikit-Learn.

Jednak nasza praca nie kończy się na wdrażaniu. Musisz także przygotować kod odpowiedzialny za monitorowanie systemu, który będzie sprawdzał działanie w regularnych odstępach czasu i wysyłał powiadomienia w przypadku pojawienia się problemu. Może to być nagła przerwa w działaniu, wynikająca z awarii jakiegoś składnika infrastruktury, lecz również nieznaczny rozkład, niezauważony przez dłuższy czas. Jest to dość powszechny problem, gdyż modele mają tendencję do „rozkładu” wraz z ewoluowaniem danych: istotnie, świat ciągle ulega zmianom, dlatego jeżeli model został wyuczony za pomocą zeszłorocznych danych, może nie być dostosowany do dzisiejszych.



Nawet model wyuczony do klasyfikowania zdjęć psów i kotów może wymagać regularnego trenowania, nie z powodu nagłego zmutowania tych zwierząt, lecz ze względu na rozwój aparatów fotograficznych, a także formatów danych czy współczynników ostrości, jasności i rozmiaru. Ponadto w następnym roku może zapanować moda na inną rasę lub ludzie zaczną zakładać pupilkom kapelusze. Kto wie?

Musisz więc monitorować bieżącą wydajność modelu. Jak to jednak robić? To zależy. W niektórych przypadkach skuteczność modelu można wywnioskować bezpośrednio ze wskaźników cyklu pracy. Na przykład jeżeli Twój model stanowi część systemu rekomendacji i sugeruje produkty, które mogą zainteresować użytkowników, to z łatwością można monitorować liczbę sprzedanych zalecanych produktów. Jeżeli liczba ta zmala (w stosunku do produktów niezalecanych), to model staje się głównym podejrzany. Być może potok danych uległ załamaniu albo należy wyuczyć model za pomocą świeżych danych (wkrótce wrócę do tego tematu).

Nie zawsze jednak jest możliwa ocena wydajności modelu bez analizy wykonanej przez człowieka. Założymy na przykład, że wyuczyliśmy model klasyfikujący obrazy (zob. rozdział 3.) do wykrywania kilku usterek produktów na linii montażowej. W jaki sposób zostaniesz zaalarmowana/zaalarmowany o spadku skuteczności modelu, zanim tysiące wadliwych produktów trafi do klientów? Jednym ze sposobów jest przesłanie do ludzkich kontrolerów jakości części obrazów sklasyfikowanych

przez model (zwłaszcza tych, przy których nie miał dużej pewności). W zależności od zadania kontrolerzy mogą być ekspertami lub laikami, np. pracownikami źródeł społecznościovych (takich jak Amazon Mechanical Turk). W pewnych sytuacjach mogą być oni nawet samymi użytkownikami, odpowiadającymi na przykład za pomocą ankiet lub zmodyfikowanych testów captcha²⁶.

Tak czy inaczej musisz ustanowić system monitorujący (uwzględniający ludzkich kontrolerów kontrolujących pracujący model, ale niekoniecznie), a także wszelkie istotne procesy definiujące zachowanie w przypadku wystąpienia awarii i sposoby przygotowania się do nich. Niestety zadanie to jest pracochłonne. Tak naprawdę często wymaga poświęcenia więcej czasu niż w przypadku stworzenia i wytrenowania modelu.

Jeżeli dane ciągle ewoluują, musisz zaktualizować swoje zestawy danych i regularnie trenować model. W miarę możliwości zautomatyzuj cały proces. Oto kilka elementów, które można zautomatyzować:

- Regularne gromadzenie i oznaczanie świeżych danych (np. za pomocą ludzkich kontrolerów).
- Stworzenie skryptu automatycznie uczącego model i dostrajającego hiperparametry. Skrypt ten może być uruchamiany automatycznie, np. codziennie albo co tydzień, w zależności od zapotrzebowania.
- Napisanie kolejnego skryptu, który ocenia wydajność zarówno nowego, jak i starego modelu za pomocą zaktualizowanego zestawu testowego, a także wdraża model do środowiska produkcyjnego, jeżeli jego skuteczność nie zmalała (jeżeli zmalała, to musisz znaleźć przyczynę).

Musisz także ocenić jakość danych wejściowych. Czasami wydajność nieznacznie spada z powodu słabej jakości sygnału (np. pochodzącego z uszkodzonego czujnika przesyłającego losowe wartości lub niewielkiej dynamiki sygnałów wysyłanych przez zespół znajdujący się na wcześniejszym etapie potoku), jednak może minąć trochę czasu, zanim jakość systemu spadnie na tyle, żeby zostało wysłane powiadomienie. Jeśli będziesz obserwować dane wejściowe, możesz wykryć problem nieco wcześniej. Możesz przykładowo wprowadzić alert ostrzegający przed coraz większą liczbą danych wejściowych, w których brakuje jakieś cechy albo średnia lub odchylenie standardowe zbytnio odbiegają od zestawu testowego, albo w celu kategorialnej zaczynają pojawiać się nowe kategorie.

Na koniec nie zapomnij o przygotowaniu kopii zapasowych każdego modelu, a także o wprowadzeniu procesu i narzędzi umożliwiających szybkie przywrócenie modelu z kopii zapasowej na wypadek, gdyby nowy model z jakiejś przyczyny zaczął zachowywać się nieprawidłowo. Dzięki kopiom zapasowym możesz również porównywać modele nowe z wcześniejszymi. To samo dotyczy kopii zapasowych zestawów danych, gdyż dzięki temu można cofnąć się do wcześniejszego zestawu danych, gdyby nowy zestaw został uszkodzony (np. gdyby w dodawanych świeżych danych występowało zbyt dużo elementów odstających). Za pomocą kopii zestawów danych możesz także oceniać każdy model za pomocą dowolnej wersji zestawu danych.

Jak widać, na uczenie maszynowe składa się całkiem rozbudowana infrastruktura, dlatego nie zdziwi się, jeżeli stworzenie i wdrożenie Twojego pierwszego projekt UM będzie wymagało mnóstwa czasu i wysiłku. Na szczęście po przygotowaniu infrastruktury przejście od pomysłu do produkcji stanie się znacznie szybsze.

²⁶ Captcha to test sprawdzający, czy użytkownik jest robotem. Testy te są często używane jako tani sposób oznaczania danych uczących.



Możesz chcieć stworzyć kilka podzbiorów zestawu testowego, aby ocenić skuteczność modelu w określonych obszarach danych. Na przykład możesz stworzyć podzbiór zawierający wyłącznie najświeższe dane lub zestaw testowy przeznaczony dla określonych rodzajów danych wejściowych (np. dystrykty znajdujące się w głębi lądu i dystrykty miesiące się nad oceanem). W ten sposób łatwiej odkryjesz mocne i słabe strony modelu.

Teraz Twoja kolej!

Mam nadzieję, że dzięki niniejszemu rozdziałowi wiesz już mniej więcej, jak wygląda typowy projekt uczenia maszynowego, oraz że przydadzą Ci się omówione narzędzia, za pomocą których jesteś w stanie stworzyć wspaniały system. Jak widać, lwią część czasu pochłaniają etapy przygotowywania danych, tworzenia narzędzi monitorujących, konfigurowania potoków oceniania wydajności oraz automatyzowania regularnego uczenia modelu. Oczywiście algorytmy uczenia maszynowego są istotne, jednak bardziej opłaca się zaznajomić się z całym procesem przygotowywania projektu i dobrze opanować trzy lub cztery algorytmy, niż poświęcać cały czas na poznawanie zaawansowanych algorytmów.

Jeśli więc jeszczego nie zrobiłaś/zrobiliś, teraz nadeszła dobra pora na włączenie laptopa, wybranie interesującego Cię zbioru danych i przebrnięcie przez cały omawiany proces od początku do końca. Warto rozpocząć swoją przygodę od strony <https://www.kaggle.com/>. Znajdziesz tam odpowiedni zbiór danych dla siebie, wyraźny cel oraz osoby, z którymi możesz się dzielić doświadczeniami. Baw się dobrze!

Ćwiczenia

Poniższe ćwiczenia bazują na omówionym w tym rozdziale zestawie danych *Housing*.

1. Wypróbuj regresor maszyny wektorów nośnych (`sklearn.svm.SVR`) przy użyciu różnych hiperparametrów, takich jak `kernel="linear"` (oraz różnych wartości hiperparametru C) lub `kernel="rbf"` (oraz różnych wartości hiperparametrów C i γ). Na razie nie przejmuj się tym, że nie wiesz, do czego te hiperparametry służą. Jak się spisuje najlepszy predyktor maszyny wektorów nośnych?
2. Spróbuj zastąpić klasę `GridSearchCV` obiektem `RandomizedSearchCV`.
3. Spróbuj dodać w potoku przygotowawczym funkcję przekształcającą w taki sposób, aby były dobierane wyłącznie najistotniejsze atrybuty.
4. Spróbuj stworzyć pojedynczy potok przeprowadzający pełne przygotowywanie danych i generujący ostateczne prognozy.
5. Sprawdź automatycznie niektóre funkcje przygotowawcze za pomocą klasy `GridSearchCV`.

Rozwiązania tych ćwiczeń znajdziesz w notatniku Jupyter dostępnym pod adresem <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Klasyfikacja

W rozdziale 1. wspomniałem, że najpowszechniej stosowanymi metodami uczenia nadzorowanego są regresja (przewidywanie wartości) i klasyfikacja (przewidywanie klas). Rozdział 2. poświęciłem omówieniu mechanizmu regresji oraz prognozowania cen mieszkań za pomocą różnych algorytmów, takich jak regresja liniowa, drzewa decyzyjne czy losowy las (przyjrzymy się dokładniej w następnych rozdziałach). Teraz skoncentrujemy się na modelach klasyfikacji.

Zbiór danych MNIST

W niniejszym rozdziale będziemy korzystać ze zbioru danych MNIST, składającego się z 70 000 małych rysunków zawierających cyfry odręcznie zapisane przez uczniów szkół średnich i pracowników amerykańskiego Biura Spisu Ludności (ang. *US Census Bureau*). Każdy rysunek zawiera etykietę określającą zawartą na nim cyfrę. Omawiany zestaw danych został tak dogłębnie przeanalizowany, że w świecie uczenia maszynowego bywa określany mianem powitalnego zbioru danych dla nowych adeptów: po napisaniu nowego algorytmu klasyfikującego jest on najpierw sprawdzany właśnie na zbiorze MNIST, a każda osoba poznająca meandry uczenia maszynowego przedżej czy później musi na niego natrafić.

Moduł Scikit-Learn zawiera wiele funkcji ułatwiających pobieranie popularnych zestawów danych. Jak łatwo zgadnąć, MNIST stanowi jeden z takich zbiorów. Możemy go pobrać za pomocą poniższego kodu¹:

```
>>> from sklearn.datasets import fetch_openml  
>>> mnist = fetch_openml('mnist_784', version=1)  
>>> mnist.keys()  
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',  
          'categories', 'url'])
```

Zazwyczaj zbiory danych wczytywane przez moduł Scikit-Learn mają podobną strukturę składającą się z następujących elementów:

- klucza DESCR opisującego zbiór danych,
- klucza data zawierającego tablicę, w której każdy wiersz reprezentuje przykład, a kolumna — cechę,
- klucza target przechowującego tablicę etykiet.

¹ Domyslnie moduł Scikit-Learn przechowuje pobrane zbiory danych w katalogu *\$HOME/scikit_learn_data*.

Przyjrzyjmy się teraz tym tablicom:

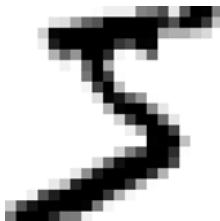
```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

W tym zbiorze danych występuje 70 000 obrazów, a każdy z nich jest opisany 784 cechami. Wynika to z faktu, że obraz ma rozmiar 28×28 pikseli i każda cecha opisuje natężenie szarości danego piksela i przyjmuje wartości od 0 (kolor biały) do 255 (kolor czarny). Sprawdźmy, jak wygląda przykładowa cyfra przechowywana w zestawie danych. Wystarczy w tym celu wybrać wektor cech danej próbki i przekształcić go w macierz o rozmiarze 28×28 , którą następnie możemy wyświetlić za pomocą funkcji `imshow()` modułu Matplotlib:

```
import matplotlib as mpl
import matplotlib.pyplot as plt

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = "binary")
plt.axis("off")
plt.show()
```



Obraz ten przypomina z kształtu cyfrę 5 i, zgodnie z etykietą, tak jest w istocie:

```
>>> y[0]
'5'
```

Zwróć uwagę, że etykieta jest łańcuchem znaków. Większość algorytmów uczenia maszynowego spodziewa się wartości numerycznych, dlatego przekształćmy y w liczbę całkowitą:

```
>>> y = y.astype(np.uint8)
```

Na rysunku 3.1 pokazuję kilka innych obrazów ze zbioru danych MNIST, dzięki czemu możesz sobie uświadomić złożoność zadania klasyfikującego.

Chwileczkę! Przed dokładniejszym przyjrzeniem się danym powinniśmy zawsze najpierw stworzyć zbiór testowy i odstawić go na bok. W rzeczywistości zestaw MNIST jest już podzielony na zbiory uczący (pierwsze 60 000 przykładów) i testowy (pozostałe 10 000 obrazów):

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



Rysunek 3.1. Cyfry ze zbioru danych MNIST

Zestaw danych uczących został już za nas przetasowany, co jest dobrym rozwiązaniem, gdyż gwarantuje, że podzbiory utworzone podczas sprawdzianu krzyżowego będą do siebie podobne (nie chcemy, aby w którymś podzbiorze zabrakło jakieś cyfry). Ponadto niektóre algorytmy są wrażliwe na kolejność próbek uczących i nie za dobrze sobie radzą, jeżeli przetwarzają wiele podobnych przykładów z rzędu. Unikniemy tego problemu dzięki przetasowaniu zbioru danych².

Uczenie klasyfikatora binarnego

Uprośćmy na razie problem i spróbujmy identyfikować tylko jedną cyfrę, np. 5. Ten „wykrywacz piątek” będzie stanowił przykład **klasyfikatora binarnego** (ang. *binary classifier*), zdolnego do rozpoznawania jedynie dwóch klas: piątek i niepiątek. Stwórzmy wektory docelowe dla tego zadania klasyfikującego:

```
y_train_5 = (y_train == 5) # Wartość True dla piątek, False dla wszystkich pozostałych cyfr  
y_test_5 = (y_test == 5)
```

Teraz wybierzmy i wytrenujmy jakiś klasyfikator. Warto rozpocząć od klasyfikatora **stochastycznego spadku wzduż gradientu** (ang. *Stochastic Gradient Descent* — SGD) — posłuży nam do tego klasa `SGDClassifier`. Algorytm ten cechuje się możliwością wydajnego przetwarzania bardzo dużych zestawów danych.

² W pewnych sytuacjach tasowanie danych może być złym pomysłem — np. w czasie pracy z danymi szeregu czasowego (takimi jak ceny akcji giełdowych czy warunki meteorologiczne). Poruszmy to zagadnienie w dalszej części książki.

wów danych. Wynika to częściowo z faktu, że klasyfikator SGD przetwarza poszczególne przykłady uczące niezależnie od siebie, po jednym naraz (z tego powodu nadaje się on również do **uczenia przyrostowego**), o czym przekonamy się już niebawem. Stwórzmy klasyfikator SGDClassifier i wytrenujmy go wobec całego zbioru uczącego:

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



Skuteczność klasyfikatora SGDClassifier zależy od losowości na etapie uczenia (stąd przyjmiotnik „stochastyczny”). Jeżeli chcesz, aby wyniki były powtarzalne, wyznacz wartość parametru random_state.

Możemy teraz posłużyć się tym algorytmem do wykrywania cyfry 5 w zbiorze danych:

```
>>> sgd_clf.predict([some_digit])  
array([True])
```

Klasyfikator zgaduje, że na wyznaczonym obrazie znajduje się cyfra 5 (wartość True). Wygląda na to, że się nie pomylił w tym konkretnym przypadku! Oceńmy teraz wydajność modelu.

Miary wydajności

Często ocena klasyfikatora stanowi większe wyzwanie od ewaluacji regresora, dlatego poświęcimy temu zagadnieniu dużą część niniejszego rozdziału. Istnieje wiele miar wydajności, dlatego zaparz sobie następną kawę i przygotuj się na kolejną porcję nowych pojęć i skrótów!

Pomiar dokładności za pomocą sprawdzianu krzyżowego

Dobrym sposobem oceny modelu jest zastosowanie użytej już przez nas w rozdziale 2. metody sprawdzianu krzyżowego.

Wykorzystajmy funkcję cross_val_score() do oceny naszego modelu SGDClassifier za pomocą metody sprawdzianu krzyżowego; wygenerujemy trzy podzbiory. Przypominam, że kroswalidacja k-krotna oznacza rozdzielenie zestawu uczącego na k podzbiorów (w tym przypadku k = 3), następnie wyuczenie modelu na k-1 podzbiorach, a na końcu przeprowadzenie prognoz i ich ocena wobec ostatniego podzbioru, a wszystko powtórzone k razy, tak że zostają wykorzystane wszystkie podzbiory (zostało to dokładniej wyjaśnione w rozdziale 2.):

```
>>> from sklearn.model_selection import cross_val_score  
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")  
array([0.96355, 0.93795, 0.95615])
```

O rety! **Dokładność** powyżej 93% (współczynnik prawidłowych prognoz) wobec wszystkich podzbioryów sprawdzianu krzyżowego? Zdumiewające, nieprawdaż? Zanim się zanadto podekscytujemy, przyjrzyjmy się bardzo prostemu klasyfikatorowi klasyfikującemu jedynie obrazy niebędące piątkami:

Implementacja sprawdzianu krzyżowego

Od czasu do czasu będziemy potrzebować większej kontroli nad metodą kroswalidacji, niż zapewnia nam domyślnie moduł Scikit-Learn. W takim przypadku możemy samodzielnie zaimplementować sprawdzian krzyżowy. Poniższy kod wykonuje mniej więcej te same operacje, co funkcja `cross_val_score()` i wyświetla te same wyniki:

```
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)) # Wyświetla wyniki 0.9502, 0.96565 i 0.96495
```

Klasa `StratifiedKFold` przeprowadza losowanie warstwowe (zostało ono omówione w rozdziale 2.), dzięki któremu otrzymujemy podzbiory zawierające reprezentacyjne populacje każdej z klas. W każdym przebiegu kod tworzy klon klasyfikatora, trenuje go za pomocą podzbiorów uczących i ocenia zdolność progностyczną na podzbiorze testowym. Następnie jest zliczana liczba prawidłowych prognoz i zostaje wyświetlony współczynnik prawidłowych prognoz.

```
from sklearn.base import BaseEstimator

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

Domyślasz się, jaka będzie dokładność tego modelu? Sprawdźmy:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

Zgadza się, uzyskujemy dokładność rzędu 90%! Wynika to jedynie z faktu, że tylko około 10% wszystkich obrazów stanowią piątki, dlatego gdybyśmy zawsze zgadywali, że dany obraz nie jest piątką, to mielibyśmy rację w ok. 90% przypadków. Nostradamus nie umywa się do nas.

Teraz już wiemy, dlaczego generalnie dokładność nie stanowi dobrej miary wydajności klasyfikatorów, zwłaszcza w przypadku **wypaczonych zbiorów danych** (ang. *skewed datasets*; są to takie zbiorы danych, w których niektóre klasy występują znacznie częściej od pozostałych).

Macierz pomyłek

Znacznie lepszym sposobem oceny wydajności klasyfikatora jest analiza **macierzy pomyłek** (ang. *confusion matrix*). Ogólna koncepcja polega tu na zliczaniu przypadków zaklasyfikowania próbek z klasy A jako przykładów należących do klasy B. Przykładowo aby dowiedzieć się, ile razy algorytm pomylił obrazy piątek z obrazami trójków, wystarczy spojrzeć na piąty rzad i trzecią kolumnę macierzy pomyłek.

W celu obliczenia macierzy pomyłek musimy najpierw uzyskać zbiór prognoz, które porównamy z rzeczywistymi wartościami docelowymi. Moglibyśmy prognozować wyniki dla zbioru testowego, jednak na razie nie będziemy go ruszać (pamiętaj, że chcemy wykorzystać zestaw testowy jedynie na samym końcu projektu, gdy nasz klasyfikator będzie już gotowy do pracy). Zamiast tego skorzystamy z funkcji `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)
```

Obydwie funkcje, `cross_val_score()` i `cross_val_predict()`, przeprowadzają k-krotny sprawdzian krzyżowy, jednak ta druga nie zwraca wyników ewaluacji, lecz prognozy uzyskane dla każdego podzbioru testowego. Oznacza to, że możemy otrzymać „czystą” prognozę dla każdej próbki stanowiącej część zbioru uczącego („czysta” w tym kontekście znaczy, że prognoza została uzyskana przez model, który wcześniej nie widział tego przykładu na etapie uczenia).

Teraz możemy uzyskać macierz pomyłek za pomocą funkcji `confusion_matrix()`. Wystarczy podać jej klasy docelowe (`y_train_5`) i przewidywane klasy (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix  
>>> confusion_matrix(y_train_5, y_train_pred)  
array([[53057, 1522],  
       [1325, 4096]])
```

Każdy rzad w macierzy pomyłek reprezentuje **rzeczywistą klasę**, natomiast kolumna symbolizuje **przewidywaną klasę**. Pierwszy rzad w naszej macierzy przechowuje obrazy niebędące piątkami (**klasa negatywna**): 53 053 próbki zostały prawidłowo sklasyfikowane jako niebędące piątkami (są to tak zwane przykłady **prawdziwie negatywne** — PN), natomiast pozostałe 1522 obrazy zostały nie-właściwie uznane za piątki (**falszywie pozytywne** — FP). W drugim rzędzie zostały umieszczone obrazy zaklasyfikowane jako piątki (**klasa pozytywna**): 1325 próbek jest nieprawidłowo sklasyfikowanych jako niebędące piątkami (**falszywie negatywne** — FN), zaś pozostałe 4096 prawidłowo zostało rozpoznanych jako piątki (**prawdziwie pozytywne** — PP). Doskonały klasyfikator uzyskiwałby wyłącznie przykłady prawdziwie pozytywne i prawdziwie negatywne, zatem niezerowe wartości w macierzy pomyłek mieściłyby się jedynie w głównej przekątnej (od lewego górnego do prawego dolnego rogu):

```
>>> y_train_perfect_predictions = y_train_5 # Udajemy, że uzyskaliśmy perfekcyjne rezultaty  
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)  
array([[54579, 0],  
       [0, 5421]])
```

Macierz pomyłek dostarcza wielu informacji, czasami jednak przydaje się bardziej zwięzły wskaźnik. Interesującym rozwiązaniem okazuje się dokładność pozytywnych prognoz — jest to tak zwana **precyzyja** klasyfikatora (równanie 3.1).

Równanie 3.1. Precyza

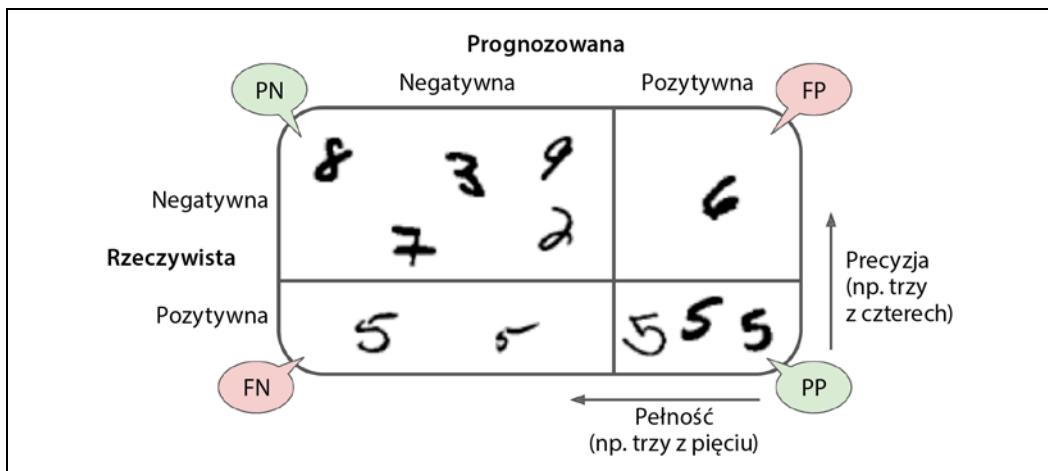
$$\text{Precyza} = \frac{PP}{PP + FP}$$

Najbardziej banalnym sposobem uzyskania doskonałej precyzji jest otrzymanie pojedynczej pozytywnej prognozy i upewnienie się, że jest prawidłowa ($\text{Precyza} = 1/1 = 100\%$). Nie byłoby to jednak zbyt przydatne, ponieważ klasyfikator ignorowałby wszystkie próbki oprócz jednej pozytywnej. Dlatego zazwyczaj precyza jest używana wraz z innym wskaźnikiem, zwanym **pełnością, czułością** lub **odsetkiem prawdziwie pozytywnych (OPP)**: jest to odsetek pozytywnych przykładów, które zostały prawidłowo rozpoznane przez klasyfikator (równanie 3.2).

Równanie 3.2. Pełność

$$\text{Pełność} = \frac{PP}{PP + FN}$$

Jeśli nie do końca rozumiesz mechanizm działania macierzy pomyłek, rysunek 3.2 powinien pomóc rozwiać wątpliwości.



Rysunek 3.2. Schemat macierzy pomyłek ukazujący przykłady prawdziwie negatywne (lewy górnny róg), fałszywie pozytywne (prawy górnny róg), fałszywie negatywne (lewy dolny róg) i prawdziwie pozytywne (prawy dolny róg)

Precyza i pełność

Moduł Scikit-Learn zawiera kilka funkcji obliczających wskaźniki klasyfikatorów, w tym również precyzę i pełność:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

Teraz nasz wykrywacz piątek nie jest już tak imponujący jak na etapie sprawdzania jego dokładności. Gdy uznaje, że rozpoznaje cyfrę 5, nie myli się jedynie w 72,9% przypadków, a do tego prawidłowo rozpoznaje jedynie 75,6% piątek.

Często wygodnie jest połączyć precyzję i pełność w jednym wskaźniku, zwanym **wynikiem F_1** , zwłaszcza jeżeli chcesz w prosty sposób porównać dwa klasyfikatory. Wynik F_1 stanowi **średnią harmoniczną** precyzji i pełności (równanie 3.3). Standardowa średnia traktuje wszystkie wartości jednakowo, natomiast średnia harmoniczna nadaje większą wagę małym wartościom. W rezultacie klasyfikator uzyska dużą wartość wyniku F_1 jedynie wtedy, gdy zarówno precyzja, jak i pełność będą miały dużą wartość.

Równanie 3.3. Wynik F_1

$$F_1 = \frac{2}{\frac{1}{\text{Precyzja}} + \frac{1}{\text{Pełność}}} = 2 \times \frac{\text{Precyzja} \times \text{Pełność}}{\text{Precyzja} + \text{Pełność}} = \frac{PP}{PP + \frac{FN + FP}{2}}$$

Aby obliczyć wynik F_1 , wystarczy wywołać funkcję `f1_score()`:

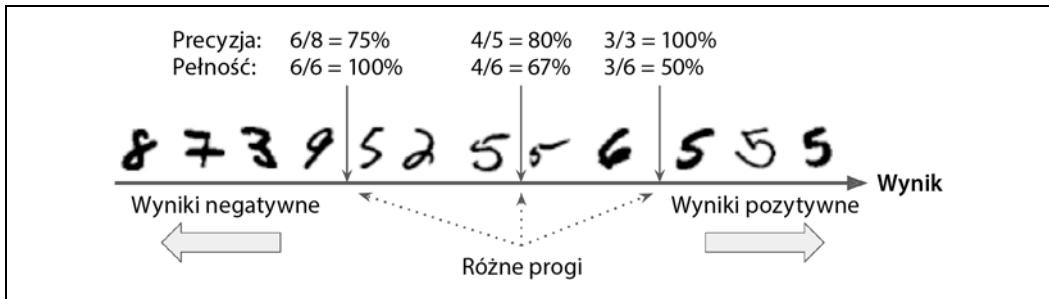
```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_train_5, y_train_pred)  
0.7420962043663375
```

Wynik F_1 faworyzuje klasyfikatory mające zbliżone wartości precyzji i pełności. Nie zawsze tego chcemy: w pewnych sytuacjach zależy nam bardziej na precyzji, a w innych na pełności. Przykładowo, po wyuczeniu klasyfikatora w określaniu filmów bezpiecznych dla dzieci, prawdopodobnie wolelibyśmy klasyfikator odrzucający wiele dobrych filmów (mała wartość pełności), ale zapamiętujący jedynie bezpieczne (duża precyzja); bardzo niekorzystna byłaby sytuacja, w której model cechuje się znacznie większą pełnością, ale jednocześnie dopuszcza kilka nieodpowiednich filmów (w takim przypadku należałoby nawet dodać do potoku element ludzki — osobę, która sprawdzałaby wyniki klasyfikatora). Z drugiej strony, założmy, że trenujemy klasyfikator rozpoznający złodziei z zapisów kamer: prawdopodobnie nic się zlego nie stanie, jeśli model będzie miał tylko 30% precyzji przy 99% pełności (jasne, ochroniarze wygenerują kilka fałszywych alarmów, ale niemal wszyscy przestępcy zostaną złapani).

Niestety nie możemy mieć wszystkiego naraz: wraz ze wzrostem precyzji maleje pełność i odwrotnie. Zależność ta jest nazywana **kompromisem pomiędzy precyzją a pełnością**.

Kompromis pomiędzy precyzją a pełnością

Aby zrozumieć ten kompromis, przyjrzyjmy się, w jaki sposób klasyfikator `SGDClassifier` podejmuje decyzje. Dla każdej próbki zostaje wyliczony wynik na podstawie **funkcji decyzyjnej**. Jeżeli jego wartość przekroczy określony próg, przykład ten zostanie przydzielony do klasy pozytywnej; w przeciwnym wypadku będzie wyznaczony do klasy negatywnej. Na rysunku 3.3 widzimy kilka cyfr umieszczone w kolejności od najniższego do najwyższego wyniku. Założmy, że **próg decyzyjny** znajduje się w miejscu wskazywanym przez środkową strzałkę (pomiędzy dwiema piątkami): w ten sposób otrzymujemy po prawej stronie 4 próbki prawdziwie pozytywne (rzeczywiście piątki) i jedną fałszywie pozytywną (cyfra 6). Zatem przy tak dobranym progu precyzja wynosi 80% (cztery z pięciu). Jednak na sześć dostępnych piątek zostają rozpoznane tylko cztery, dlatego uzyskujemy tu pełność na poziomie 67% (cztery z sześciu). Jeśli podniemy próg (przesuniemy go w prawo), przykład fałszywie



Rysunek 3.3. W powyższym kompromisie pomiędzy precyzacją a pełnością obrazy są oceniane zgodnie z wynikiem klasyfikatora, a te, których wynik przekroczy wyznaczony próg decyzyjny, zostają uznane za pozytywne; im wyższy próg, tym mniejsza pełność, ale (zazwyczaj) większa precyzaja

pozytywny (cyfra 6) stanie się prawdziwie negatywny, dzięki czemu wzrośnie precyzaja (do 100% w omawianym przykładzie), ale z kolei jedna wartość prawdziwie pozytywna przekształci się w fałszywie negatywną, przez co wartość pełności zmaleje do 50%. Z kolei obniżenie progu (przesunięcie go w lewo) zwiększy pełność przy jednoczesnej redukcji precyzaji.

Moduł Scikit-Learn nie pozwala na bezpośrednie dobieranie progów, ale daje nam dostęp do wyników decyzyjnych, za pomocą których są wyliczane prognozy. Zamiast wywoływać metodę `predict()`, możemy skorzystać z metody `decision_function()`, która zwraca obliczony wynik dla każdej próbki, a następnie wyliczyć prognozy na podstawie tych wyników przy użyciu dowolnego progu:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2412, 53175101])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([True])
```

Klasyfikator `SGDClassifier` korzysta z progu o wartości 0, dlatego powyższy kod zwraca taki sam wynik, jak metoda `predict()` (np. `True`). Podnieśmy teraz ten próg.

```
>>> threshold = 8000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

Uzyskaliśmy w ten sposób dowód, że podniesienie progu zmniejsza pełność. Analizowany obraz w rzeczywistości symbolizuje cyfrę 5 i klasyfikator rozpoznaje ją przy progu o wartości 0, ale myli się, gdy wartość progu zostanie podniesiona do wartości 8000.

W jaki sposób należy dobierać próg? Najpierw skorzystaj z funkcji `cross_val_predict()`, aby uzyskać wyniki wszystkich przykładów zestawu uczącego, tym razem jednak zaznacz, że chcesz otrzymać nie prognozy, lecz wyniki decyzyjne:

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

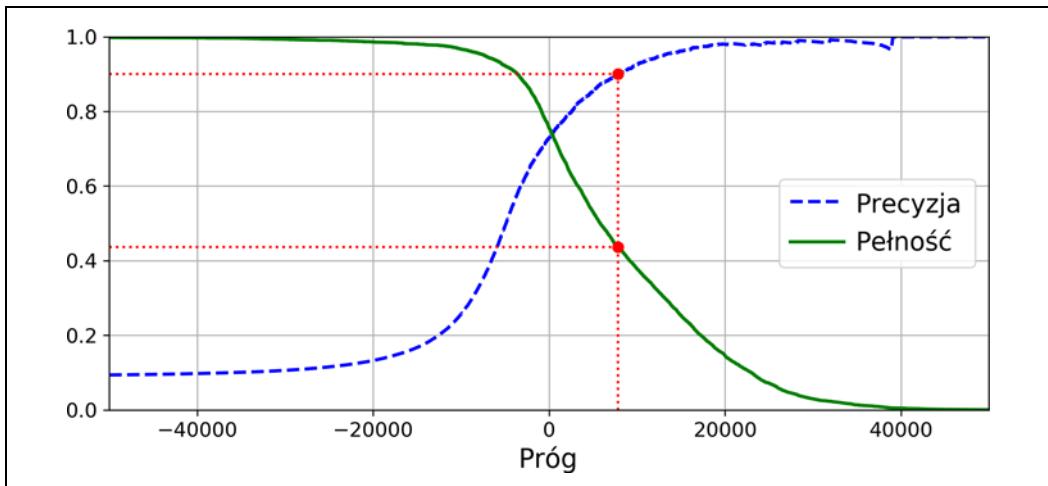
Mając te wyniki, możesz wyliczyć precyzaję i pełność dla wszystkich możliwych progów za pomocą funkcji `precision_recall_curve()`:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Na koniec za pomocą modułu Matplotlib wygeneruj wykres precyzji i pełności w funkcji progu decyzyjnego (rysunek 3.4):

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precyzja")
    plt.plot(thresholds, recalls[:-1], "g-", label="Pełność")
    [...] # Zaznaczenie progu, dodanie legendy, etykiet osi i siatki

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



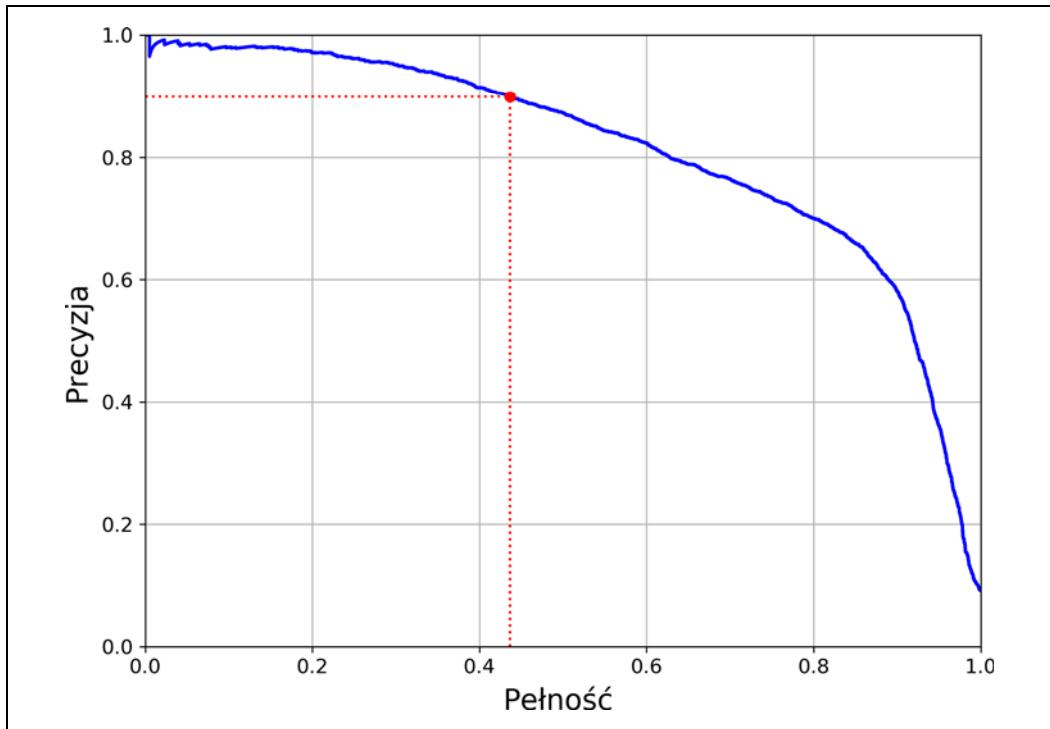
Rysunek 3.4. Wykres precyzji i pełności w funkcji progu decyzyjnego



Pewnie zastanawiasz się, dlaczego na rysunku 3.4 krzywa precyzji nie jest tak równa, jak krzywa pełności. Wynika to z faktu, że precyzja może czasami maleć wraz z podniesieniem wartości progowej (choćż zasadniczo jej wartość również wzrasta). Aby zrozumieć ten mechanizm, przyjrzyj się ponownie rysunkowi 3.3; zobacz, co się będzie działo, gdy zaczniesz od środkowego progu i będziesz przesuwać się w prawo po jednej pozycji naraz: wartość precyzji zmienia się z 4/5 (80%) na 3/4 (75%). Z drugiej strony, wraz ze wzrostem wartości progowej pełność może wyłącznie maleć, dlatego jej krzywa jest taka gładka.

Innym sposobem wyznaczenia dobrego kompromisu jest narysowanie wykresu precyzji bezpośrednio w funkcji pełności, co zostało zaprezentowane na rysunku 3.5 (wyznaczyliśmy tu ten sam próg co poprzednio).

Jak widać, krzywa precyzji zaczyna stromo opadać dopiero przy wartości mniej więcej 80% pełności. Prawdopodobnie będziemy chcieli wybrać kompromis pomiędzy precyzją a pełnością wyznaczony tuż przed początkiem opadania krzywej — mniej więcej przy 60% pełności. Oczywiście wybór odpowiedniego progu zależy od danego projektu.



Rysunek 3.5. Wykres precyzji w funkcji pełności

Załóżmy, że zależy Ci na precyjji rzędu 90%. Przyglądamy się pierwszemu wykresowi (w razie potrzeby nieznacznie go przybliżamy) i dowiadujemy się, że powinniśmy użyć progu o wartości ok. 8000. Dla zwiększenia dokładności możesz poszukać najmniejszego progu zapewniającego precyjję rzędu do najmniej 90% (funkcja `np.argmax()` wyznaczy pierwszy indeks wartości maksymalnej, co w tym przypadku oznacza pierwszą wartość `true`):

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # ~7816
```

Aby wyliczyć prognozy (na razie na zbiorze uczącym), nie będziemy wywoływać metody `predict()` klasyfikatora, lecz skorzystamy z następującego kodu:

```
y_train_pred_90 = (y_scores > threshold_90_precision)
```

Sprawdźmy teraz precyjję i pełność tych prognoz:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
0.4368197749492714
```

Świętne! Otrzymaliśmy klasyfikator cechujący się 90-procentową precyją! Jak widać, nie jest trudno uzyskać klasyfikator mający praktycznie dowolną wybraną przez nas precyję: wystarczy wyznaczyć odpowiednio wysoki próg i już. Chwilunia, nie tak szybko. Precyjny klasyfikator nie jest zbyt przydatny, jeśli ma za małą pełność!



Jeśli ktoś powie: „Uzyskajmy 99% precyzji”, zapytaj: „A przy jakiej wartości pełności?“.

Wykres krzywej ROC

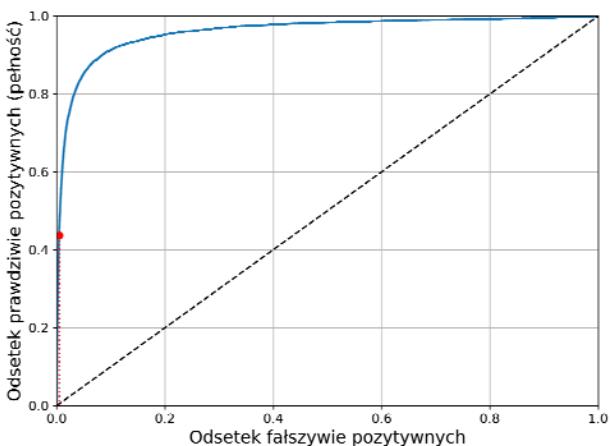
Wykres krzywej **charakterystyki roboczej odbiornika** (ang. *receiver operating characteristic* — ROC) stanowi kolejne popularne narzędzie używane wraz z klasyfikatorami binarnymi. Przypomina on w znacznym stopniu wykres precyzji w funkcji pełności, jednak w tym przypadku rysujemy **odsetek prawdziwie pozytywnych** (inna nazwa pełności; w skrócie OPP) w funkcji **odsetka fałszywie pozytywnych** (OFP) — jest to odsetek negatywnych przykładów, które zostały nieprawidłowo sklasyfikowane jako pozytywne. Otrzymujemy tę wartość, **odejmując odsetek prawdziwie negatywnych** (OPN) od liczby 1; jak można się domyślić, jest to odsetek negatywnych próbek, które zostały prawidłowo sklasyfikowane jako negatywne. Wartość ta bywa również nazywana **specyficznością**. Dlatego krzywa ROC składa się z **czułości** (czyli pełności) w funkcji **1 - specyficzność**.

Aby narysować wykres krzywej ROC, wykorzystujemy funkcję `roc_curve()` do obliczenia przykładów OPP i OFP dla różnych wartości progowych:

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

Teraz przy użyciu modułu Matplotlib możemy narysować wykres OPP w funkcji OFP. Poniższy kod generuje wykres widoczny na rysunku 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--') # Przerwana przekątna wykresu  
    [...] # Dodanie etykiety osi i siatki  
plot_roc_curve(fpr, tpr)  
plt.show()
```



Rysunek 3.6. Wykres krzywej ROC

Także w tym przypadku mamy do czynienia z kompromisem: im wyższa wartość pełności (OPP), tym więcej fałszywie pozytywnych (OFP) uzyskuje klasyfikator. Linia przerywana symbolizuje krzywą ROC całkowicie losowego klasyfikatora; dobry klasyfikator stara się nie zbliżać do tej prostej (czyli dąży do lewej górnej części wykresu).

Jednym ze sposobów porównywania klasyfikatorów jest pomiar ich **obszaru pod krzywą** (ang. *area under the curve* — AUC). Obszar AUC w przypadku doskonałego klasyfikatora byłby równy 1, natomiast w całkowicie losowym klasyfikatorze ma on wartość 0,5. Moduł Scikit-Learn zawiera funkcję służącą do wyliczania obszaru AUC na wykresie ROC:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.9611778893101814
```



Skoro wykres krzywej ROC tak bardzo przypomina wykres krzywej precyzji w funkcji pełności (ang. *precision/recall curve* — PR), skąd mamy wiedzieć, którego z nich użyć? Zgodnie z niepisaną zasadą powinniśmy korzystać z wykresu PR wtedy, gdy klasa pozytywna występuje rzadko lub gdy bardziej Ci zależy na odsetku fałszywie pozytywnych niż fałszywie negatywnych. Z kolei wykres ROC stosujemy w pozostałych sytuacjach. Przykładowo, jeśli spojrzymy na powyższy wykres krzywej ROC (oraz na wynik obszaru pod tą krzywą), moglibyśmy uznać, że odpowiedzialny za nią klasyfikator jest naprawdę dobry. Wynika to jednak głównie z faktu, że w zestawie danych istnieje niewiele wartości pozytywnych (piątek) w stosunku do negatywnych (niepiątek). Z drugiej strony, wykres krzywej PR wyraźnie pokazuje, że klasyfikator ten można jeszcze usprawnić (krzywa powinna jeszcze bardziej dążyć do lewego górnego rogu wykresu).

Wyuczmy teraz klasyfikator `RandomForestClassifier` i porównajmy jego krzywą ROC oraz obszar AUC z wynikami uzyskanymi dla klasyfikatora `SGDClassifier`. Najpierw musimy obliczyć wyniki dla każdego przykładu ze zbioru uczącego. Jednak z powodu mechanizmu działania (zob. rozdział 7.) klasy `RandomForestClassifier` nie zawiera metody `decision_function()`. Znajdziemy tu jednak metodę `predict_proba()`. Klasyfikatory modułu Scikit-Learn przeważnie zawierają którąś z tych dwóch metod albo obydwie. Metoda `predict_proba()` zwraca tablicę, w której każdy wiersz reprezentuje próbkę, a kolumna — klasę; wyliczone są w niej prawdopodobieństwa przynależności danego przykładu do określonej klasy (np. 70% szans, że dana próbka jest piątką):

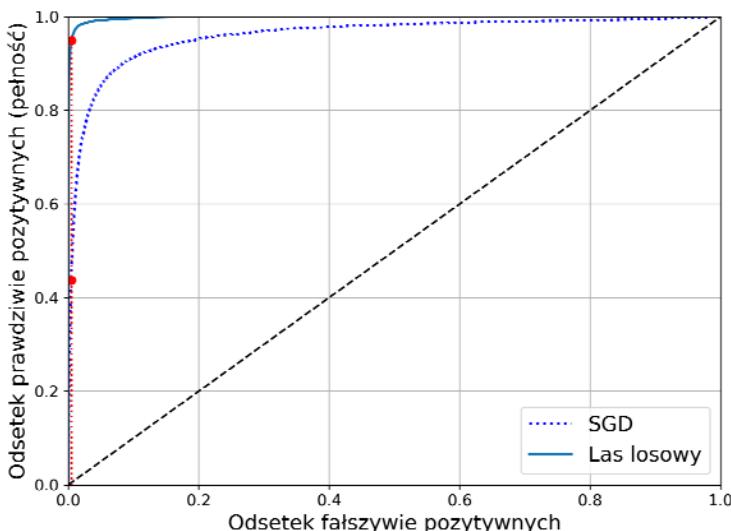
```
from sklearn.ensemble import RandomForestClassifier  
  
forest_clf = RandomForestClassifier(random_state=42)  
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,  
method="predict_proba")
```

Funkcja `roc_curve()` oczekuje etykiet i wyników, ale zamiast tych drugich możemy skorzystać z prawdopodobieństwa przynależności do klasy. Użyjmy prawdopodobieństwa przynależności do klasy pozytywnej jako wyniku:

```
y_scores_forest = y_probas_forest[:, 1] # Wynik = prawd. przynależności do klasy pozytywnej  
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

Jesteśmy teraz w stanie narysować wykres krzywej ROC. Warto również wyrysować wcześniejszą krzywą ROC, aby móc porównać obydwa modele (rysunek 3.7):

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



Rysunek 3.7. Porównanie krzywych ROC: klasyfikator lasu losowego jest lepszy od klasyfikatora SGD, ponieważ jego krzywa ROC znajduje się znacznie bliżej górnego lewego rogu, a jego obszar AUC jest także korzystniejszy

Jak widać na rysunku 3.7, krzywa ROC klasyfikatora RandomForestClassifier wygląda znacznie lepiej od krzywej klasyfikatora SGDClassifier: dociera ona znacznie bliżej lewego górnego rogu wykresu. W związku z tym jego wynik obszaru AUC jest również o wiele lepszy:

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Spróbuj zmierzyć również wartości precyzji i pełności: powinnaś/powinieneś uzyskać wyniki rzędu, odpowiednio, 99% i 86.6%. Całkiem nieźle!

Potrafisz już trenować klasyfikatory binarne, wybierać odpowiedni wskaźnik dla danego zadania, oceniać wydajność klasyfikatorów za pomocą sprawdzianu krzyżowego, dobierać do swoich potrzeb odpowiedni kompromis pomiędzy precyją a pełnością oraz porównywać modele przy użyciu krzywych ROC i wyników AUC. Spróbujmy teraz wykrywać jakieś inne cyfry oprócz piątek.

Klasyfikacja wieloklasowa

Klasyfikatory binarne służą do rozpoznawania dwóch klas, natomiast **klasyfikatory wieloklasowe** (zwane również **klasyfikatorami wielomianowymi**) służą do rozróżniania większej ich liczby.

Niektóre algorytmy (takie jak klasyfikatory SGD, klasyfikatory losowego lasu lub naiwne klasyfikatory bayesowskie) są w stanie natywnie zajmować się wieloma klasami naraz. Inne (takie jak maszyny wektorów nośnych czy klasyfikatory regresji logistycznej) mają charakter typowo binarny. Istnieje jednak wiele strategii, dzięki którym możemy przeprowadzać klasyfikację wieloklasową przy użyciu wielu klasyfikatorów binarnych.

Jeden ze sposobów stworzenia systemu klasyfikującego 10 klas obrazów cyfr (od 0 do 9) polega na wyuczeniu 10 klasyfikatorów binarnych, po jednym na każdą cyfrę (wykrywacz zer, jedynek, dwójkę itd.). Następnie na etapie klasyfikacji obrazu uzyskujemy wynik z każdego klasyfikatora dla danej cyfry i wybieramy klasę, która uzyskała najwyższy wynik. Jest to tzw. strategia **jeden przeciw reszcie** (ang. *one-versus-rest* — OvR), znana również jako **jeden przeciw wszystkim** (ang. *one-versus-all* — OvA).

Jeszcze innym rozwiązaniem jest wyuczenie klasyfikatora binarnego dla każdej pary cyfr: jeden rozpoznaje zera i jedynki, drugi — zera i dwójkę, jeszcze inny — jedynki i dwójkę itd. Jest to strategia **jeden przeciw jednemu** (ang. *one-versus-one* — OvO). Jeśli masz do czynienia z N klasami, musisz wytrenować $N \times (N-1)/2$ klasyfikatorów. W przypadku zestawu danych MNIST oznacza to wyuczenie 45 klasyfikatorów binarnych! Aby zaklasyfikować dany obraz, musielibyśmy przepuścić go przez wszystkie 45 klasyfikatorów i sprawdzić, która klasa wygrywa większość „ pojedynków ”. Główną zaletą strategii OvO jest fakt, że każdy klasyfikator musi zostać wytrenowany jedynie wobec części zbioru uczącego składającego się z obydwu porównywanych klas.

Pewne algorytmy (jak choćby maszyny wektorów nośnych) nie skalują się zbyt dobrze do rozmiarów zbioru uczącego. W ich przypadku preferowane jest stosowanie strategii OvO, ponieważ uczenie wielu klasyfikatorów wobec niewielkich zbiorów danych przebiega szybciej niż trenowanie kilku klasyfikatorów wobec dużych zbiorów przykładów. Jednakże w przypadku większości klasyfikatorów binarnych zalecane jest korzystanie ze strategii OvR.

Moduł Scikit-Learn wykrywa sytuacje, gdy próbujemy używać algorytmu klasyfikacji binarnej w procesie klasyfikacji wieloklasowej, i automatycznie przechodzi na strategię OvR lub OvO, w zależności od algorytmu. Sprawdźmy to na klasyfikatorze maszyny wektorów nośnych (zob. rozdział 5.), za pomocą klasy `sklearn.svm.SVC`:

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, a nie y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

To było proste! Powyższy fragment kodu wyucza klasyfikator SVC wobec zbioru uczącego za pomocą pierwotnych klas docelowych w zakresie od 0 do 9 (`y_train`), zatem nie korzystamy tu z klas docelowych 5 przeciw reszcie (`y_train_5`). Następnie zostaje wyliczona prognoza (tym razem prawidłowa). Zgodnie z tą strategią moduł Scikit-Learn skorzystał w rzeczywistości ze strategii OvO: wytrenował 45 klasyfikatorów binarnych, uzyskał wyniki decyzyjne dla analizowanego obrazu i wybrał klasę, która wygrała większość „ pojedynków ”.

Jeżeli wywołasz metodę `decision_function()`, zauważysz, że zostaje zwrócony nie jeden wynik na obraz, ale aż 10 wyników, po jednym na każdą klasę:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[2.92492871, 7.02307409, 3.93648529, 0.90117363, 5.96945908,
       9.5, 1.90718593, 8.02755089, -0.13202708, 4.94216947]])
```

Istotnie, najwyższy wynik dotyczy klasy 5:

```
>>> np.argmax(some_digit_scores)
5
>>> svm_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
>>> svm_clf.classes_[5]
5
```



Klasyfikator po wytrenowaniu przechowuje listę klas docelowych we własnym atrybutie `classes_`, w uporządkowanym szeregu kolejności. W naszym przypadku indeks każdej klasy w tablicy `classes_` dogodnie dla nas jest dopasowany do każdej klasy (np. okazuje się, że klasa o indeksie 5 symbolizuje klasę piątek), zazwyczaj jednak nie powinniśmy liczyć na takie szczęście.

Jeśli chcesz zmusić moduł Scikit-Learn do stosowania strategii OvO lub OvR, możesz skorzystać z klas `OneVsOneClassifier` lub `OneVsRestClassifier`. Wystarczy stworzyć ich wystąpienie i przekazać konstruktorowi klasyfikator (nie musi być on nawet binarny). Na przykład poniższy kod stworzy klasyfikator wieloklasowy (na podstawie klasyfikatora `SVC`) przy użyciu strategii OvR:

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> ovr_clf = OneVsRestClassifier(SVC())
>>> ovr_clf.fit(X_train, y_train)
>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```

Równie łatwe jest uczenie klasyfikatora `SGDClassifier` (lub `RandomForestClassifier`):

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

Tym razem nie zostały wdrożone strategie OvO ani OvR, ponieważ klasyfikatory SGD mogą bezpośrednio klasyfikować przykłady w sposób wieloklasowy. Metoda `decision_function()` zwraca teraz po jednej wartości na każdą klasę. Spójrzmy na wynik wyznaczony każdej klasie przez klasyfikator SGD:

```
>>> sgd_clf.decision_function([some_digit])
array([-15955.22628, -38080.96296, -13326.66695, 573.52692, -17680.68466,
      2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889])
```

Jak widać, klasyfikator wykazuje dość duży stopień pewności w swoich przewidywaniach: niemal wszystkie wyniki mają dużą wartość ujemną, podczas gdy klasa 5 uzyskała wynik 2412,5. Model wykazuje drobne wątpliwości w przypadku klasy 3, której wynik to 573,5. Teraz oczywiście chcemy ocenić skuteczność tych klasyfikatorów. Jak zwykle użyjemy w tym celu sprawdzianu krzyżowego. Sprawdźmy dokładność klasyfikatora `SGDClassifier` za pomocą funkcji `cross_val_score()`:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802, 0.87129356, 0.86988048])
```

Uzyskujemy wynik 84% wobec wszystkich podzbiorów testowych. Gdybyśmy korzystali z losowego klasyfikatora, otrzymalibyśmy wynik 10% dokładności, zatem nie jest źle, ale mogłoby być znacznie lepiej. Po zwyczajnym przeskalowaniu danych wejściowych (zostało to omówione w rozdziale 2.) osiągnęlibyśmy dokładność powyżej 89%:

```
>>> 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([0.89707059, 0.8960948 , 0.90693604])
```

Analiza błędów

Gdybyśmy mieli do czynienia z rzeczywistym projektem, przeprowadzalibyśmy teraz czynności wymienione w dodatku B. Sprawdzalibyśmy możliwości przygotowywania danych, testowalibyśmy różne modele, stworzylibyśmy krótką listę najlepszych modeli i dostroilibyśmy ich hiperparametry za pomocą klasy GridSearchCV, a do tego próbowałibyśmy w jak największym stopniu zautomatyzować cały proces. Obecnie założymy, że znaleźliśmy obiecujący model, i szukamy sposobów, aby go jeszcze usprawnić. Jedną z możliwości jest analiza rodzajów błędów popełnianych przez ten model.

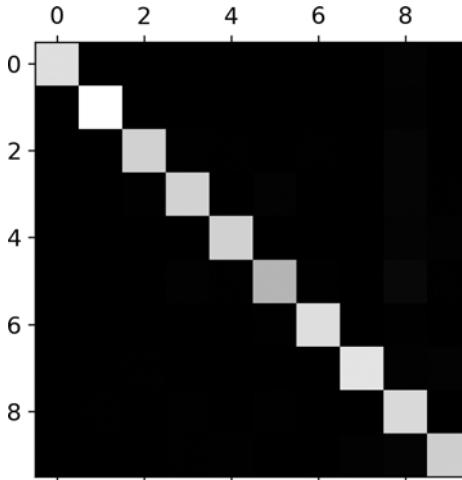
Po pierwsze, przyjrzyjmy się macierzy pomyłek. Podobnie jak robiliśmy to wcześniej, musimy wyliczyć prognozy za pomocą funkcji `cross_val_predict()`, a następnie wywołać funkcję `confusion_matrix()`:

```
>>> 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([[5578,      0,    22,     7,     8,    45,    35,      5,   222,     1],
       [  0,  6410,    35,    26,     4,    44,     4,     8,   198,    13],
       [ 28,    27,  5232,   100,    74,    27,    68,    37,   354,    11],
       [ 23,    18,   115,   5254,     2,   209,    26,    38,   373,    73],
       [ 11,    14,    45,    12,  5219,    11,    33,    26,   299,   172],
       [ 26,    16,    31,   173,    54,  4484,    76,    14,   482,    65],
       [ 31,    17,    45,     2,    42,    98,  5556,     3,   123,      1],
       [ 20,    10,    53,    27,    50,    13,     3,  5696,   173,   220],
       [ 17,    64,    47,   91,     3,   125,    24,    11,  5421,     48],
       [ 24,    18,    29,    67,  116,    39,     1,   174,   329,  5152]])
```

Całkiem sporo tych liczb. Często wygodniej jest spojrzeć na graficzną reprezentację macierzy pomyłek za pomocą funkcji `matshow()`:

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

Macierz ta wygląda całkiem nieźle, gdyż większość obrazów znajduje się na głównej przekątnej, co oznacza, że zostały prawidłowo sklasyfikowane. Piątki wyglądają nieco ciemniej od pozostałych cyfr, co może oznaczać, że w zbiorze danych znajduje się mniej obrazów piątek lub że klasyfikator nie sprawuje się tak dobrze wobec piątek, jak w przypadku pozostałych cyfr. W rzeczywistości możemy zweryfikować obydwa stwierdzenia.

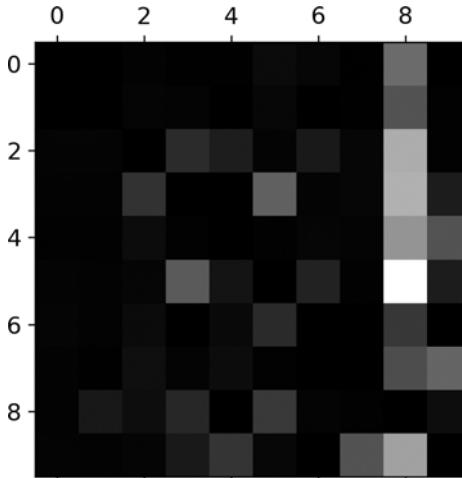


Skoncentrujmy się na narysowaniu wykresu błędów. Najpierw musimy podzielić każdą wartość w macierzy pomyłek przez liczbę obrazów należących do danej klasy, dzięki czemu będziemy w stanie porównać poziomy błędu zamiast bezwzględnej liczby błędów (gdybyśmy przy niej pozostali, klasy zawierające liczne przykłady wypadałyby nazbyt niekorzystnie):

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

Wypełnijmy główną przekątną zerami po to, aby pozostały same błędy, i znowu wyświetlimy macierz pomyłek w postaci graficznej:

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



Wyraźnie widzimy rodzaje błędów popełnianych przez klasyfikator. Przypominam, że rzędy reprezentują rzeczywiste klasy, kolumny zaś — przewidywane klasy. Kolumna reprezentująca klasę 8 jest

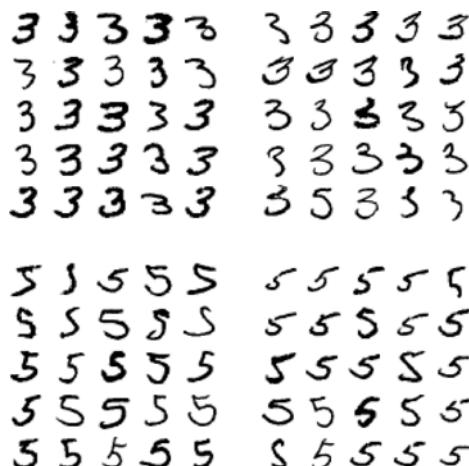
dość jasna, co oznacza, że wiele obrazów zostało nieprawidłowo sklasyfikowanych jako ta właśnie cyfra. Jednak rząd klasy 8 nie wygląda tak źle, co oznacza, że zasadniczo większość ósemek jest prawidłowo sklasyfikowanych. Jak widać, macierz pomyłek nie jest zbyt symetryczna. Możemy również dostrzec, że trójki i piątki są często mylone (w obydwie strony).

Analiza macierzy pomyłek często dostarcza wskazówek pozwalających usprawnić klasyfikator. Patrząc na nią, możemy stwierdzić, że należy popracować nad zmniejszeniem liczby fałszywych ósemek. Możesz na przykład spróbować zdobyć więcej obrazów cyfr przypominających ósemki (które jednak nimi nie są), dzięki czemu klasyfikator nauczy się je odróżniać od rzeczywistych ósemek. Ewentualnie moglibyśmy spróbować stworzyć nowe cechy pomagające klasyfikatorowi — przykładowo, napisać algorytm zliczający liczbę zamkniętych pętli na obrazie (np. cyfra 8 ma dwie pętle, szóstka ma jedną, a piątka żadnej). Jeszcze innym rozwiązaniem jest wstępne przetwarzanie obrazów (np. za pomocą modułów Scikit-Image, Pillow lub OpenCV) w taki sposób, aby uwypuklić pewne wzorce, takie jak wspomniane już zamknięte pętle.

Dzięki analizowaniu poszczególnych błędów możemy dowiedzieć się również, jak właściwie działa nasz klasyfikator i jaka jest przyczyna pomyłek, proces ten jest jednak bardziej skomplikowany i czasochłonny. Narysujmy, na przykład, wykres przykładowych trójek i piątek (funkcja `plot_digits()` wykorzystuje po prostu funkcję `imshow()` modułu Matplotlib; szczegóły znajdziesz w notatniku Jupyter dołączonym do tego rozdziału):

```
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()
```



Dwa bloki o rozmiarze 5×5 widoczne po lewej stronie przedstawiają cyfry sklasyfikowane jako trójkę, natomiast po prawej stronie widzimy prognozowane piątki. Niektóre nieprawidłowo rozpoznane cyfry (np. w lewym dolnym i prawym górnym bloku) zostały tak fatalnie zapisane, że nawet człowiek miałby problem z ich klasyfikacją (np. piątka znajdująca się na przecięciu pierwszego rzędu i drugiej kolumny wygląda jak wyjątkowo niechlujnie zapisana trójka). Jednak większość nieprawidłowo sklasyfikowanych obrazów to z perspektywy ludzkiego mózgu wyraźne błędy i ciężko nam zrozumieć, dlaczego klasyfikator popełnił takie gafy³. Wynika to z faktu, że skorzystaliśmy z klasy SGDClassifier, która jest w istocie prostym modelem liniowym. Przydziela on jedynie wagi przynależności do klasy dla każdego piksela, a w trakcie analizy nowej próbki po prostu sumuje ważone poziomy szarości pikseli, aby uzyskać wynik dla każdej klasy. Skoro więc trójkę i piątki różnią się zaledwie kilkoma pikselami, klasyfikator może łatwo się mylić w ich rozróżnianiu.

Główną różnicą pomiędzy trójkami i piątkami jest położenie małej linii łączącej górną kreskę z dolnym łukiem. Jeśli narysujeśmy cyfrę trzy z tym połączeniem przesuniętym nieznacznie w lewą stronę, klasyfikator może ją uznać za piątkę, i odwrotnie. Innymi słowy, klasyfikator ten jest czuły na przesunięcia i obroty. Z tego wynika, że w celu ułatwienia rozróżniania trójelek od piątek należałoby wstępnie przetworzyć obrazy tak, aby zostały wyśrodkowane i jak najmniej przechylone. Prawdopodobnie rozwiązanie to zmniejszyłoby występowanie również innych błędów.

Klasyfikacja wieloetykietowa

Do tej pory każdy przykład był przydzielany wyłącznie do jednej klasy. W niektórych sytuacjach chcemy, aby klasyfikator wyznaczał wiele klas dla jednego wystąpienia. Weźmy pod uwagę klasyfikator rozpoznawania twarzy: co powinien zrobić, jeśli rozpozna kilka osób na jednym zdjęciu? Powinien przydzielić po jednej etykiecie na każdą rozpoznaną osobę. Założymy, że model taki został wyuczony do rozpoznawania trzech osób: Alicji, Władka i Karola. Po zaprezentowaniu zdjęć Alicji i Karola powinien zostać wygenerowany wynik [1, 0, 1] (czyli „Alicja tak, Władek nie, Karol tak”). Tego typu system klasyfikujący zdolny do wyznaczania wielu binarnych znaczników nosi nazwę systemu **klasyfikacji wieloetykietowej** (ang. *multilabel classification*).

Nie będziemy jeszcze zajmować się zagadnieniem rozpoznawania twarzy, ale przyjrzymy się prostszemu przykładowi, wyłącznie w celach poglądowych:

```
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)
```

Powyższy kod generuje tablicę `y_multilabel` zawierającą dwie docelowe etykiety dla każdego obrazu cyfry: pierwsza z nich określa, czy mamy do czynienia z cyfrą o dużej wartości (7, 8 i 9), natomiast

³ Nie zapominaj jednak, że nasz mózg stanowi doskonali system rozpoznawania wzorców, a układ wzrokowy przeprowadza mnóstwo wstępnych modyfikacji obrazu, zanim trafi on do naszej świadomości, dlatego cechująca nas łatwość klasyfikowania obrazów wcale nie oznacza, że jest to prosty proces.

druga mówi nam, czy dana cyfra jest nieparzysta. Następnie zostaje utworzone wystąpienie klasyfikatora KNeighborsClassifier (obsługuje on klasyfikację wieloetykietową, czego nie można powiedzieć o wszystkich klasyfikatorach), po czym uczymy go za pomocą tablicy zawierającej wiele docelowych elementów. Teraz możemy wyliczyć prognozy, w wyniku czego otrzymujemy dwie etykiety:

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

Zgadza się! Cyfra 5 rzeczywiście nie ma dużej wartości (False) i jest nieparzysta (True).

Istnieje wiele sposobów oceniania wydajności klasyfikatora wieloetykietowego, a dobór wskaźnika zależy w rzeczywistości od natury projektu. Jedną z metod jest obliczenie wyniku F_1 dla poszczególnych etykiet (lub użycie dowolnej innej omówionego wcześniej wskaźnika klasyfikatora binarnego), a następnie po prostu wyliczenie średniej. Poniższy kod wylicza uśredniony wynik F_1 dla wszystkich etykiet:

```
>>> 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")
0.976410265560605
```

Zakładamy tu jednak, że wszystkie etykiety mają taką samą wagę, co nie zawsze jest prawdą. Jest to istotne zwłaszcza w sytuacji, gdybyśmy mieli znacznie więcej zdjęć Alicji niż Władka lub Karola — należałoby wtedy wyznaczyć większą wagę dla wyników zdjęć Alicji. Prostym rozwiązaniem jest nadanie każdej etykiecie wagi równej jej **współczynnikowi wsparcia** (ang. *support*; np. liczbie próbek zawierających etykietę docelową). W tym celu wystarczy w powyższym fragmencie kodu wstawić wyrażenie `average="weighted"`⁴.

Klasyfikacja wielowyściowa

Ostatnim rodzajem zadania klasyfikacyjnego, jakie omówimy, jest **klasyfikacja wielowyściowo-wieloklasowa** (ang. *multioutput-multiclass classification*), zwana również **klasyfikacją wielowyściową** (ang. *multioutput classification*). Jest to po prostu uogólnienie klasyfikacji wieloetykietowej, w którym każda etykieta może być wieloklasowa (tj. może mieć ponad dwie możliwe wartości).

Aby to zilustrować, stwórzmy system usuwający szumy z obrazów cyfr. Danymi wejściowymi będą zaszumione obrazy cyfr, na wyjściu natomiast uzyskamy (taką mamy nadzieję) czysty obraz cyfry w postaci tablicy zawierającej poziomy szarości poszczególnych pikseli, zupełnie jak w przypadku zestawu danych MNIST. Zwrót uwagi, że wynik klasyfikatora jest wieloetykietowy (na jeden piksel przypada jedna etykieta), a każda z etykiet może mieć różne wartości (poziomy szarości piksela mieszczą się w zakresie od 0 do 255). Jak zatem widać, jest to doskonały przykład systemu klasyfikacji wielowyściowej.

⁴ Moduł Scikit-Learn zawiera kilka innych opcji uśredniania oraz wskaźników klasyfikatora wieloetykietowego; szczegóły znajdziesz w dokumentacji.



Granica pomiędzy klasyfikacją a regresją bywa czasami bardzo zamazana, tak jak w omawianym przypadku. Zapewne przewidywanie poziomu szarości pikseli bardziej przypomina regresję niż klasyfikację. Do tego systemy wielowyjściowe nie ograniczają się wyłącznie do zadań klasyfikacji; możesz nawet stworzyć system, którego rezultatami jest wiele etykiet na każdy przykład, w tym zarówno etykiety klas, jak i etykiety wartości.

Zaczniemy od utworzenia zbiorów uczących i testowych — za pomocą funkcji `randint()` dołączymy zaszumienie do poziomów szarości pikseli w obrazach zbioru danych MNIST. Obrazami docelowymi będą pierwotne próbki:

```
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 = y_train
y_test_mod = y_test
```

Zobaczmy, jak wygląda przykładowy obraz ze zbioru testowego (tak, zagładamy do danych testowych, dlatego wskazany jest srogi mars na Twoim czole):



Po lewej stronie widzimy zaszumiony obraz wejściowy, a po prawej — czysty obraz docelowy. Wyuczmy teraz nasz klasyfikator i oczyszcmy za jego pomocą obraz:

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



Wygląda całkiem podobnie do obrazu docelowego! Na tym zakończymy naszą wycieczkę po krainie klasyfikacji. Umiesz już dobierać odpowiednie wskaźniki do zadań klasyfikacji, wyznaczać optymalny kompromis pomiędzy precyzją a pełnością, porównywać klasyfikatory, a także — bardziej ogólnie — tworzyć dobre systemy klasyfikujące dopasowane do różnorodnych zadań.

Ćwiczenia

1. Spróbuj stworzyć klasyfikator zbioru danych MNIST osiągający ponad 97% dokładności dla zbioru testowego. Podpowiedź: całkiem nieźle sprawdza się klasyfikator `KNeighborsClassifier`; musisz tylko dobrać odpowiednie wartości hiperparametrów (zastosuj metodę przeszukiwania siatki wobec hiperparametrów `weights` i `n_neighbors`).
2. Napisz funkcję przesuwającą obraz z zestawu danych MNIST o jeden piksel w dowolnym kierunku (w lewo, prawo, do góry lub w dół)⁵. Następnie dla każdego obrazu w zbiorze uczącym stwórz cztery przesunięte kopie (po jednej na każdy kierunek przesunięcia) i dołącz je do zbioru uczącego. Na koniec wytrenuj swój najlepszy model wobec takiego rozszerzonego zestawu danych i zmierz jego dokładność przy użyciu zbioru testowego. Twój model powinien teraz sprawować się jeszcze lepiej! Taka technika sztucznego powiększania zestawu uczącego bywa nazywana **dogenerowaniem danych** (ang. *data augmentation*) lub **rozszerzeniem zbioru uczącego** (ang. *training set expansion*).
3. Pobaw się zbiorem danych *Titanic*. Warto rozpocząć od serwisu Kaggle (<https://www.kaggle.com/c/titanic>).
4. Trudniejsze zadanie: stwórz klasyfikator spamu:
 - Pobierz przykłady spamu i normalnych wiadomości z publicznych zestawów danych aplikacji Apache SpamAssassin (<https://spamassassin.apache.org/old/publiccorpus/>).
 - Wypakuj te dane z archiwów i zapoznaj się z ich strukturą.
 - Rozdziel te dane na zbiory uczący i testowy.
 - Stwórz potok przygotowawczy przekształcający każdą wiadomość w wektor cech. Wiadomości powinny być przekształcane w wektor (rzadki) wskazujący obecność lub nieobecność każdego możliwego słowa. Na przykład, jeśli wszystkie wiadomości zawierają tylko cztery wyrazy: „Hello”, „how”, „are” i „you”, to wiadomość o treści „Hello you Hello Hello you” powinna zostać skonwertowana do postaci wektora [1, 0, 0, 1] (co oznaczałoby, że wyraz „Hello” jest obecny, „how” — nieobecny, „are” — nieobecny i „you” — obecny) lub [3, 0, 0, 2], jeśli wolisz zliczać liczbę wystąpień danego wyrazu.
Możesz chcieć dodać hiperparametry do potoku przygotowawczego, aby decydować, czy brać pod uwagę nagłówki wiadomości, konwertować rozmiar znaków do jednolitej postaci (np. małe litery), usuwać znaki interpunkcyjne, zastępować wszystkie adresy URL wyrazem „URL”, zastępować wszystkie liczby wyrazem „LICZBA”, a nawet żeby przeprowadzać **analizę słowotwórczą** (np. usuwać końcówki wyrazów; dostępne są odpowiednie biblioteki środowiska Python),
Na koniec wypróbuj kilka różnych klasyfikatorów i sprawdź, czy jesteś w stanie stworzyć wydajny filtr spamu (cechujący się zarówno dużą precyzją, jak i pełnością).

Rozwiązań tych ćwiczeń znajdziesz w notatnikach Jupyter dostępnych na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

⁵ Możesz użyć funkcji `shift()` stanowiącej część modułu `scipy.ndimage.interpolation`. Przykładowo, funkcja `shift(→image, [2, 1], cval=0)` przesuwa obraz o dwa piksele w dół i jeden piksel w prawo.

Uczenie modeli

Dotychczas traktowaliśmy modele uczenia maszynowego i ich algorytmy uczące niczym czarne skrzynki. Jeśli poświęciłaś/poświęciłeś czas na ćwiczenia umieszczone w poprzednich rozdziałach, dziwisz się zapewne, jak wiele możemy dokonać, nie znając podstaw działania modelu: zoptymalizować system regresyjny, usprawnić klasyfikator rozpoznawania cyfr, a nawet stworzyć od podstaw filtr spamu — wszystko to bez wiedzy na temat tworzących je mechanizmów. Faktycznie, w wielu przypadkach nie musimy znać szczegółów implementacji.

Jednak znajomość architektury modelu pomaga się w nim odnaleźć, dobrać właściwy algorytm uczący oraz zestaw hiperparametrów. Ułatwia nam to również usuwanie usterek z kodu, a także poprawia naszą skuteczność analizy błędów. Poza tym większość zagadnień poruszonych w tym rozdziale stanowi podwaliny zrozumienia, tworzenia i uczenia sieci neuronowych (którymi zajmiemy się w części II).

W niniejszym rozdziale zaczniemy od przyjrzenia się jednemu z najprostszych modeli — modelowi regresji liniowej. Omówimy dwa bardzo odmienne sposoby jego uczenia:

- Za pomocą „jawnego” wzoru bezpośrednio wyliczającego parametry, dzięki czemu model zostanie optymalnie wyuczony wobec zbioru uczącego (np. parametry minimalizujące funkcję kosztu dla zbioru uczącego).
- Za pomocą iteracyjnej metody optymalizacyjnej zwanej metodą gradientu prostego, dzięki której parametry modelu są stopniowo poprawiane w celu zminimalizowania funkcji kosztu wobec zestawu uczącego, co pozwala ostatecznie uzyskać te same wartości parametrów, co w przypadku pierwszej metody. Przeanalizujemy kilka odmian metody gradientu prostego, których będziemy bez przerwy używać podczas omawiania sieci neuronowych w części II, m.in. algorytmy: wsadowy gradientu prostego, schodzenia po gradiencie z minigrupami oraz stochastycznego spadku wzduł gradientu.

Następnie przejdziemy do regresji wielomianowej, czyli bardziej skomplikowanego modelu zdolnego do uczenia się wobec nieliniowych zbiorów danych. Model ten zawiera więcej parametrów niż algorytm regresji liniowej, przez co okazuje się bardziej podatny na przetrenowanie, dlatego skoncentrujemy się na sposobach wczesnego wykrywania tego problemu za pomocą krzywych uczenia, po czym zajmiemy się kilkoma technikami regularyzacji pozwalającymi zmniejszyć ryzyko przetrenowania.

Na koniec przyjrzymy się dwóm modelom powszechnie stosowanym w zadaniach klasyfikacji: regresji logistycznej i regresji softmax.



W rozdziale tym znajdziesz kilka równań matematycznych, wykorzystujących podstawową notację algebry liniowej i analizy matematycznej. Do ich zrozumienia wymagana jest podstawowa znajomość wektorów i macierzy, musisz znać pojęcia transpozycji, iloczynu skalarnego, macierzy odwrotnej i pochodnej cząstkowej. Jeśli te nazwy nic Ci nie mówią, zajrzyj do notatnika Jupyter zawierającego wprowadzenie do algebry liniowej i analizy matematycznej (<ftp://ftp.helion.pl/przykłady/troya.zip>). Jeśli nie znośisz matematyki, powinnaś/powinieneś tak czy siak przebrnąć przez ten rozdział, pomijając wszelkie wzory; mam nadzieję, że sam opis tekstowy wystarczy do zrozumienia większości pojęć.

Regresja liniowa

W rozdziale 1. przyjrzieliśmy się prostemu modelowi regresji liniowej prognozującemu satysfakcję z życia: $\text{satysfakcja_z_życia} = \theta_0 + \theta_1 \times \text{PKB_per_capita}$.

Model ten stanowi jedynie funkcję liniową cechy wejściowej PKB_per_capita . Składowe θ_0 i θ_1 są parametrami tego modelu.

Ogólnie mówiąc, model liniowy otrzymuje prognozy poprzez obliczenie ważonej sumy cech wejściowych i stałej zwanej **punktem obciążenia** (ang. *bias term*) lub **punktem przecięcia** (ang. *intercept term*), co zostało ukazane w równaniu 4.1.

Równanie 4.1. Predykcja za pomocą modelu regresji liniowej

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

W tym równaniu:

- \hat{y} — prognozowana wartość,
- n — liczba cech,
- x_i — wartość i-tej cechy,
- θ_j — j-ty parametr modelu (w tym również punkt przecięcia θ_0 oraz wagi cech $\theta_1, \theta_2, \dots, \theta_n$).

Możemy to zapisać w zwartej, zwięzlotyzowanej formie, tak jak w równaniu 4.2.

Równanie 4.2. Predykcja za pomocą modelu regresji liniowej (postać wektorowa)

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

- $\boldsymbol{\theta}$ — wektor parametrów modelu, zawierający punkt obciążenia θ_0 i wagi cech od θ_1 do θ_n ,
- \mathbf{x} — wektor cech danego przykładu, zawierający cechy od x_0 do x_n , gdzie x_0 zawsze równa się 1,
- $\boldsymbol{\theta} \cdot \mathbf{x}$ — iloczyn skalarny wektorów $\boldsymbol{\theta}^T$ i \mathbf{x} , równy oczywiście wyrażeniu $\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$,
- h_{θ} — funkcja hipotezy wykorzystująca parametry $\boldsymbol{\theta}$ modelu.



W uczeniu maszynowym wektory są często reprezentowane jako **wektory kolumnowe**, czyli tablice dwuwymiarowe zawierające jedną kolumnę. Jeżeli θ i \mathbf{x} są wektorami kolumnowymi, to predykcja przyjmuje postać $\hat{y} = \theta^T \mathbf{x}$, gdzie θ^T jest **transpozycją** macierzy θ (wektorem wierszowym), natomiast operacja $\theta^T \mathbf{x}$ to iloczyn macierzowy θ^T i \mathbf{x} . Uzyskiwana prognoza jest oczywiście taka sama, jedyną różnicą jest jej reprezentowanie w jednoelementowej macierzy, a nie jako wartość skalarną. W tej książce będę używał tej notacji po to, aby unikać lawirowania pomiędzy iloczynem skalarnym a wektorowym.

No dobrze, tak wygląda model regresji liniowej, ale jak możemy go trenować? Przypominam, że uczenie modelu oznacza wyznaczenie jego parametrów w taki sposób, że będą optymalnie dopasowane do zbioru uczącego. W tym celu musimy najpierw zmierzyć, jak dobrze (lub jak źle) model jest dopasowany do tych danych. Z rozdziału 2. wiemy, że najpopularniejszą miarą wydajności dla modelu regresyjnego jest pierwiastek błędu średniokwadratowego (RMSE; równanie 2.1). Zatem w celu wyuczenia modelu regresji liniowej musimy określić wartość wektora θ minimalizującą błąd RMSE. W praktyce znacznie łatwiej zminimalizować wartość błędu średniokwadratowego (ang. *mean square error* — MSE), przy czym uzyskujemy takie same rezultaty (ponieważ wartość minimalizująca funkcję minimalizuje również jej pierwiastek kwadratowy)¹.

Za pomocą równania 4.3 wyliczamy błąd MSE hipotezy h_θ wobec zestawu uczącego X .

Równanie 4.3. Funkcja kosztu MSE dla modelu regresji liniowej

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

Większość tych notacji została wyjaśniona w rozdziale 2. (zob. ramka „Notacje”). Jedyna różnica polega na tym, że zapisujemy h_θ zamiast h , aby podkreślić, że model zostaje sparametryzowany przez wektor θ . W celu uproszczenia będziemy zapisywać jedynie $MSE(\theta)$ zamiast $MSE(\mathbf{X}, h_\theta)$.

Równanie normalne

Aby określić wartość wektora θ minimalizującą funkcję kosztu, używamy **jawnego rozwiązania** (ang. *closed-form solution*) — innymi słowy, wzoru matematycznego dającego bezpośredni wynik. Jest to tak zwane **równanie normalne** (rysunek 4.4).

Równanie 4.4. Równanie normalne

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

¹ Często się zdarza, że algorytm uczący będzie próbował zoptymalizować inną funkcję niż miara wydajności używana do oceny ostatecznego modelu. Wynika to najczęściej z faktu, że dana funkcja jest łatwiejsza do wyliczenia, ponieważ zawiera przydatne własności różniczkowania, których brakuje metryce wydajności, lub ponieważ chcemy ograniczyć model na etapie uczenia, o czym się przekonasz, gdy będę omawiał regularyzację.

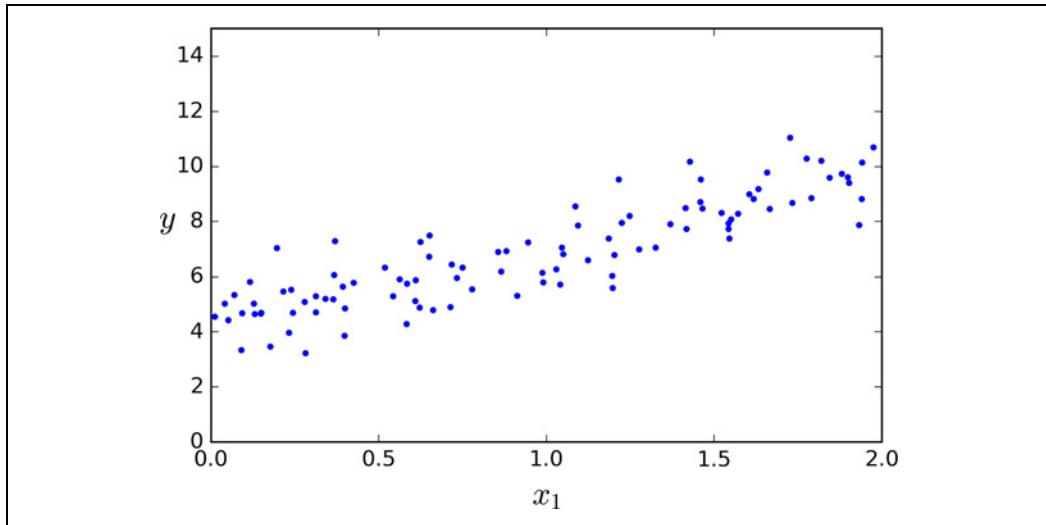
W tym równaniu:

- $\hat{\theta}$ — wartość wektora θ minimalizująca funkcję kosztu,
- y — wektor wartości docelowych od $y^{(1)}$ do $y^{(m)}$.

Wygenerujmy teraz dane przypominające z grubsza liniowe, aby przetestować omawiany wzór (rysunek 4.1):

```
import numpy as np

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



Rysunek 4.1. Losowo wygenerowany liniowy zbiór danych

Obliczmy teraz wartość $\hat{\theta}$ za pomocą równania normalnego. Skorzystamy z funkcji `inv()` dostępnej w module algebry liniowej Numpy (`np.linalg`), aby obliczyć macierz odwrotną, natomiast metoda `dot()` posłuży nam do przemnożenia macierzy:

```
X_b = np.c_[np.ones((100, 1)), X] # Dodaje  $x_0 = 1$  do każdej próbki
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Dane wygenerowaliśmy za pomocą funkcji $y = 4 + 3x_1 + \text{szum gaussowski}$. Sprawdźmy, co ten wzór wyliczył:

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

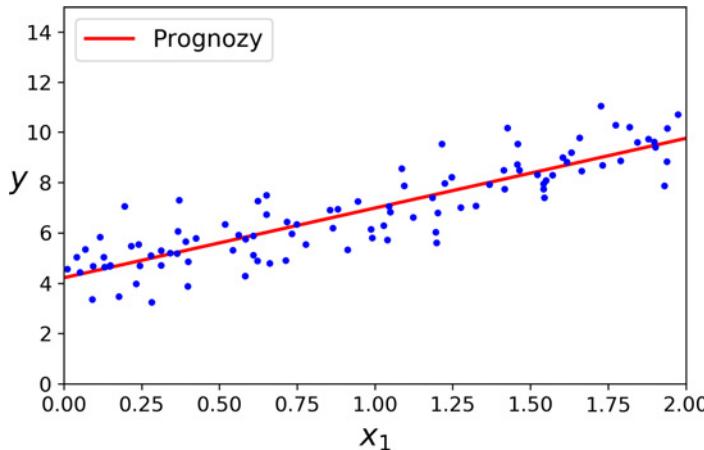
Oczekiwaliśmy, że otrzymamy $\theta_0 = 4$ i $\theta_1 = 3$ zamiast $\theta_0 = 4,215$ i $\theta_1 = 2,770$. Wynik jest całkiem bliski optymalnemu, ale z powodu szumu nie jesteśmy w stanie odzyskać dokładnych parametrów pierwotnej funkcji.

Teraz za pomocą $\hat{\theta}$ możemy wyliczyć prognozy:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # Dodaje  $x_0 = 1$  do każdej próbki
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[4.21509616],
       [9.75532293]])
```

Narysujmy wykres prognoz wyliczonych przez ten model (rysunek 4.2):

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



Rysunek 4.2. Prognozy modelu regresji liniowej

Przeprowadzenie regresji liniowej za pomocą Scikit-Learn jest proste²:

```
>>> 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]])
```

Klasa `LinearRegression` bazuje na funkcji `scipy.linalg.lstsq()` (skrót `lstsq` wywodzi się od angielskiej nazwy *least squares*, czyli „najmniejsze kwadraty”), którą możemy wywołać bezpośrednio:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)
>>> theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

² Zwróć uwagę, że moduł Scikit-Learn rozgranicza punkt przecięcia (`intercept_`) od wag cech (`coef_`).

Funkcja ta oblicza $\hat{\boldsymbol{\theta}} = \mathbf{X}^+ \mathbf{y}$, gdzie \mathbf{X}^+ jest pseudoodwrotnością macierzy \mathbf{X} (mówiąc dokładniej, uogólnioną macierzą pseudoodwrotną Moore'a-Penrose'a). Możesz wykorzystać funkcję np.linalg.pinv() do bezpośredniego obliczenia macierzy pseudoodwrotnej:

```
>>> np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

Sama w sobie macierz pseudoodwrotna jest obliczana za pomocą standardowej techniki faktoryzacji macierzy zwanej **rozkładem według wartości osobliwych** (ang. *Singular Value Decomposition* — SVD). Dzięki niej jesteśmy w stanie rozłożyć macierz zestawu danych uczących \mathbf{X} na iloczyn trzech macierzy $\mathbf{U} \Sigma \mathbf{V}^T$ (więcej informacji znajdziesz w dokumentacji funkcji numpy.linalg.svd()). Macierz pseudoodwrotna jest obliczana jako $\mathbf{X}^+ = \mathbf{V} \Sigma^+ \mathbf{U}^T$. W celu obliczenia macierzy Σ^+ wyznacza w macierzy Σ zera w miejscu wszystkich wartości nieprzekraczających małej wartości progowej, następnie następuje wszystkie wartości niezerowe ich odwrotnosciami, a na koniec transponuje tak uzyskaną macierz. Rozwiążanie to jest skuteczniejsze od rozwiązywania równania normalnego, a do tego nie sprawiają tu problemu skrajne przypadki: w rzeczy samej, równanie normalne może okazać się bezużyteczne, jeżeli macierz $\mathbf{X}^T \mathbf{X}$ jest odwracalna (tzn. osobliwa) tak, że $m < n$, lub w przypadku nadmiarowości niektórych cech, ale macierz pseudoodwrotna jest zawsze zdefiniowana.

Złożoność obliczeniowa

Równanie normalne wylicza odwrotność macierzy $\mathbf{X}^T \mathbf{X}$, dzięki czemu otrzymujemy macierz $(n+1) \times (n+1)$ (gdzie n oznacza liczbę cech). **Złożoność obliczeniowa** odwrócenia takiej macierzy wynosi zazwyczaj od około $O(n^{2.4})$ do $O(n^3)$ (w zależności od implementacji). Inaczej mówiąc podwojenie liczby cech wydłuża czas obliczeń od około $2^{2.4} = 5,3$ do $2^3 = 8$ razy.

Technika SVD wykorzystywana w klasie LinearRegression ma złożoność rzędu mniej więcej $O(n^2)$. Jeżeli podwoisz liczbę cech, zwiększasz czas obliczeniowy w przybliżeniu czterokrotnie.



Szybkość zarówno równania normalnego, jak i algorytmu SVD znacznie się zmniejsza, gdy mamy do czynienia z dużą liczbą cech (np. 100 000). Dobra informacja jest taka, że obydwa rozwiązania są liniowe pod względem liczby przykładów tworzących zbiór danych uczących (złożoność obliczeniowa wynosi $O(m)$), co oznacza, że radzą sobie całkiem nieźle z dużymi zbiorami danych, przy założeniu, że zestawy te mieszczą się w pamięci.

Poza tym po wyuczeniu modelu regresji liniowej (za pomocą równania normalnego lub dowolnego innego algorytmu) prognozy są bardzo szybko wyliczane: złożoność obliczeniowa jest liniowa zarówno wobec liczby próbek, dla których chcemy uzyskać prognozy, jak i liczby cech. Innymi słowy, wyliczanie predykcji dla dwukrotnie większej liczby przykładów (lub cech) podwoi czas obliczeń.

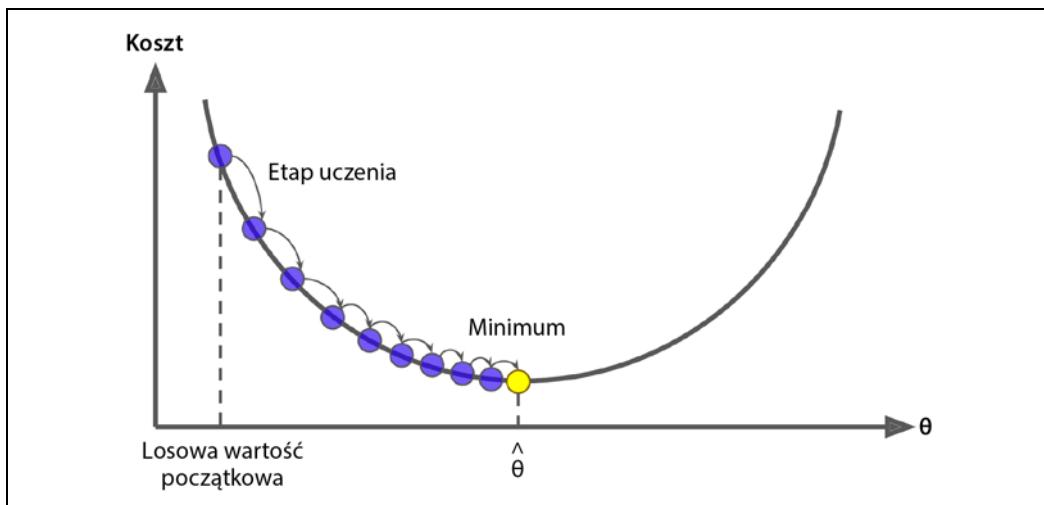
Teraz przyjrzymy się zgoła odmiennemu sposobowi uczenia modelu regresji liniowej, bardziej dostosowanemu do olbrzymiej liczby cech lub zbiorów danych niemieszczących się w pamięci.

Gradient prosty

Metoda **gradientu prostego** (ang. *gradient descent*) stanowi prosty algorytm optymalizacyjny służący do znajdowania optymalnych rozwiązań dla bardzo szerokiej gamy problemów. Ogólna koncepcja gradientu prostego polega na wielokrotnym poprawianiu wartości parametrów w celu zminimalizowania funkcji kosztu.

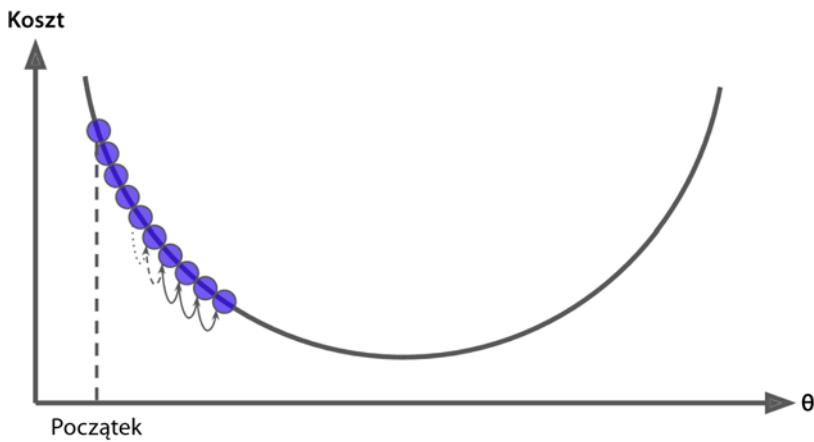
Załóżmy, że zabłędziliśmy w górach z powodu gęstej mgły i wyczuwamy jedynie nachylenie terenu pod stopami. Logicznym rozwiązaniem zejścia na dno doliny jest podążanie w dół po jak największej pochyłości. Dokładnie taki jest mechanizm działania gradientu prostego: algorytm mierzy lokalny gradient funkcji błędu w odniesieniu do wektora parametrów θ , a następnie podąża w kierunku malejącego gradientu. Po uzyskaniu wartości 0 docieramy do minimum funkcji!

Dokładniej mówiąc, zaczynamy od wypełnienia wektora θ losowymi wartościami (jest to tak zwana **inicjacja losowa** — ang. *random initialization*). Następnie stopniowo doprecyzowujemy metodą małych kroków, gdzie w każdym kroku staramy się zredukować funkcję kosztu (np. błąd MSE), aż do momentu uzyskania **zbieżności** algorytmu z wartością minimalną tejże funkcji (rysunek 4.3).



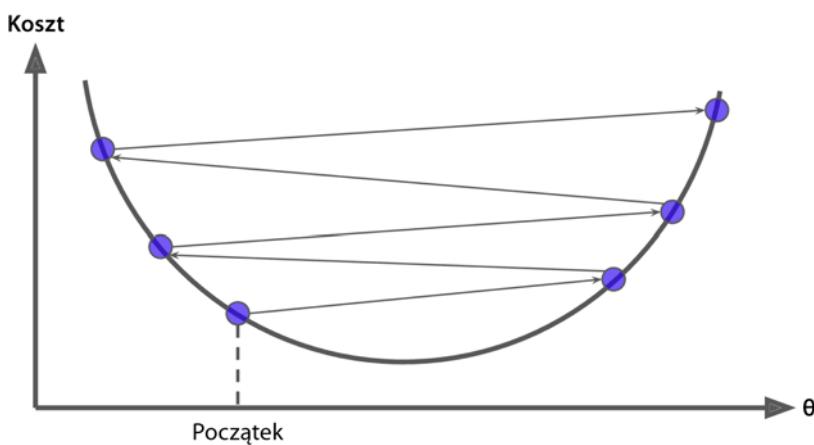
Rysunek 4.3. Na powyższym schemacie gradientu prostego parametry modelu są inicjowane losowo i stopniowo są dostrajane w sposób minimalizujący funkcję kosztu; rozmiar kroku uczącego jest proporcjonalny do nachylenia funkcji kosztu, zatem w miarę zbliżania się do minimum kroki te stają się coraz mniejsze

Ważnym elementem metody gradientu prostego jest rozmiar poszczególnych „kroków”, określony za pomocą hiperparametru zwanego **współczynnikiem uczenia** (ang. *learning rate*). Jeżeli jego wartość jest zbyt mała, to algorytm będzie musiał wykonać wiele przebiegów w celu uzyskania zbieżności, co bywa czasochłonne (rysunek 4.4).



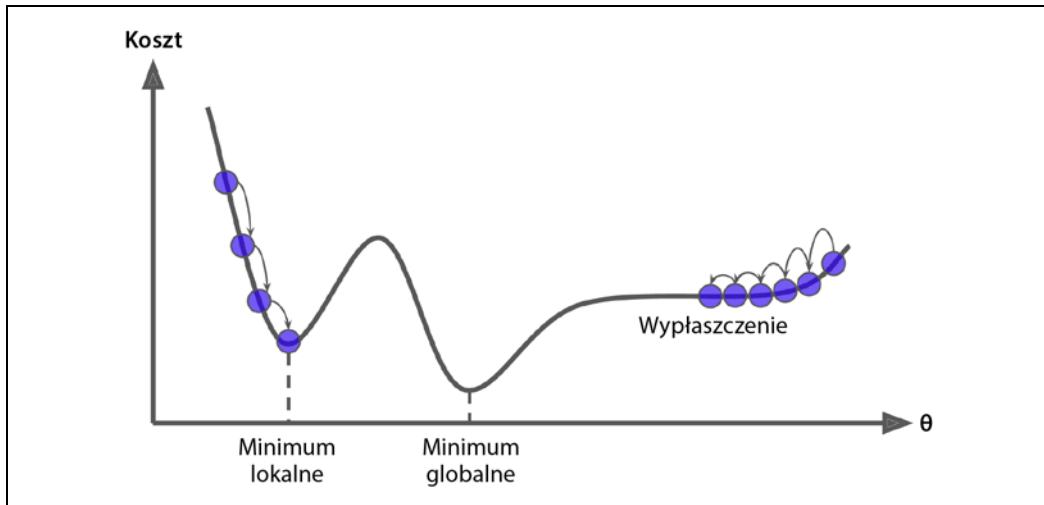
Rysunek 4.4. Zbyt mała wartość współczynnika uczenia

Z drugiej strony, przy zbyt dużej wartości współczynnika uczenia możesz przeoczyć dolinę i wyładować na przeciwnym wznieśieniu, być może nawet na większej wysokości niż wcześniej. W konsekwencji możemy otrzymać rozbieżny algorytm uzyskujący coraz większe wartości i niezdolny do znalezienia prawidłowego rozwiązania (rysunek 4.5).



Rysunek 4.5. Zbyt duża wartość współczynnika uczenia

Pamiętajmy również, że nie wszystkie funkcje kosztu mają taki ładny kształt. Mogą występować doliny i pagórki, wypłaszczenia, a także różnorodne inne nieregularności przebiegu funkcji, utrudniające użycanie zbieżności. Na rysunku 4.6 widzimy dwa główne wyzwania związane z gradientem prostym. Jeżeli inicjacja losowa rozpoczęcie się po lewej stronie wykresu, to algorytm stanie się zbieżny z **minimum lokalnym**, które nie jest tak dobre, jak **minimum globalne**. Z kolei jeśli rozpoczęliśmy proces poszukiwania minimum po prawej stronie wykresu, upłynie mnóstwo czasu, zanim algorytm przebrnie przez wypłaszczenie. Natomiast jeśli zatrzyma się zbyt wcześnie, nigdy nie osiągnie globalnego minimum.



Rysunek 4.6. Pułapki gradientu prostego

Na szczęście funkcja kosztu dla modelu regresji liniowej jest **wypukła**, co oznacza, że po wybraniu dwóch dowolnych punktów na jej przebiegu łączący je odcinek nigdy nie przetnie się z tą krzywą. Oznacza to, że nie istnieją lokalne minima, tylko jedno minimum globalne. Jest to również funkcja ciągła, której nachylenie nigdy nie zmienia się gwałtownie³. Obydwa te fakty mają bardzo ważny skutek: metoda gradientu prostego gwarantuje zbliżenie się na dowolną odległość do minimum globalnego (pod warunkiem że poczekamy wystarczająco długo, a współczynnik uczenia nie ma zbyt dużej wartości).

W rzeczywistości funkcja kosztu ma stożkowaty kształt, ale jeżeli cechy znajdują się w bardzo różniących się skalach, stożek ten może przyjąć wydłużoną postać. Na rysunku 4.7 widzimy metodę gradientu prostego stosowaną wobec zestawu uczącego, gdzie cechy 1. i 2. znajdują się w takiej samej skali (po lewej), oraz dla zbioru uczącego, w którym cecha 1. ma znacznie mniejsze wartości niż cecha 2. (po prawej)⁴.

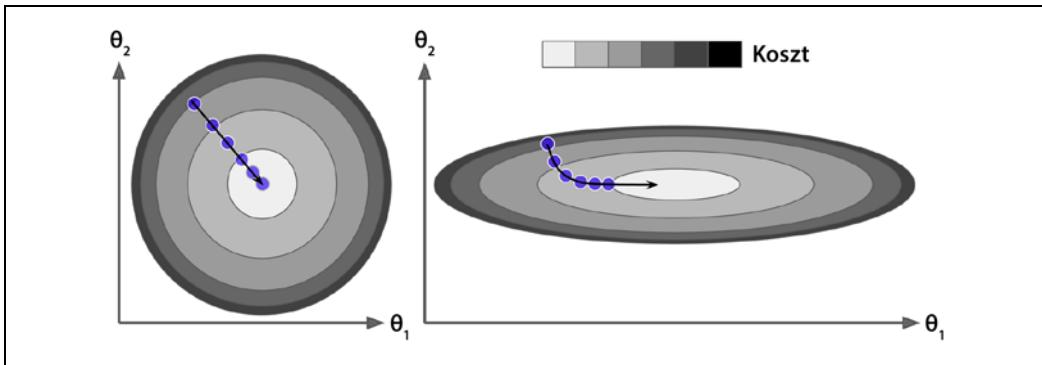
Jak widać na rysunku po lewej stronie, algorytm gradientu prostego dąży wprost do minimum, dzięki czemu szybko do niego dociera, natomiast na wykresie po prawej dąży najpierw niemal równolegle do osi y, po czym mnóstwo czasu spędza, poruszając się w niemal poziomym kierunku. W końcu osiągnie globalne minimum, ale minie wiele czasu.



Podczas korzystania z metody gradientu prostego należy umieścić wszystkie cechy w zblizonej skali (np. za pomocą klasy `StandardScaler` modułu Scikit-Learn), gdyż w przeciwnym wypadku algorytm będzie potrzebował znacznie więcej czasu na osiągnięcie zbieżności.

³ W ujęciu matematycznym ciągłość tej funkcji wynika z **warunku Lipschitza**.

⁴ Cecha 1. ma mniejsze wartości, dlatego trzeba w większym stopniu zmienić wartość θ_1 , aby wpływać na funkcję kosztu, dlatego wykres funkcji jest wydłużony wzdłuż osi x.



Rysunek 4.7. Metoda gradientu prostego przy skalowaniu cech (po lewej) i bez skalowania cech (po prawej)

Wykresy na rysunku 4.7 mówią nam również, że uczenie modelu oznacza poszukiwanie kombinacji parametrów minimalizujących funkcję kosztu (wobec zbioru uczącego). Są to poszukiwania w **przestrzeni parametrycznej** modelu: wraz z liczbą parametrów rośnie liczba wymiarów przestrzeni parametrycznej i tym bardziej stają się poszukiwania; szukanie igły w trzywymiarowym stogu siana jest znacznie trudniejsze niż w przestrzeni trójwymiarowej. Na szczęście w przypadku regresji liniowej funkcja kosztu jest wypukła, dlatego igła znajduje się na samym dnie stożka.

Wsadowy gradient prosty

Aby zaimplementować metodę gradientu prostego, musimy obliczyć gradient funkcji kosztu wobec każdego parametru θ_j modelu. Innymi słowy, musimy policzyć, jak bardzo funkcja kosztu ulegnie zmianie, jeżeli tylko troszeczkę zmodyfikujemy θ_j . Proces ten nosi nazwę wyliczania **pochodnej cząstkowej**. Można porównać go do pytania: „Jakie będzie nachylenie terenu pod moimi nogami, gdy się ustawię w kierunku wschodu?”, następnie do zadania takiego samego pytania dla kierunku północnego itd. (oraz wielu innych kierunków, jeśli jesteś w stanie wyobrazić sobie Wszechświat składający się z większej liczby wymiarów niż standardowe trzy przestrzenne). Równanie 4.5 wylicza pochodną cząstkową funkcji kosztu dla parametru θ_j , co zapisujemy jako $\frac{\partial}{\partial \theta_j} MSE(\boldsymbol{\theta})$.

Równanie 4.5. Wzór na pochodną cząstkową funkcji kosztu

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

Zamiast pojedynczo wyliczać poszczególne pochodne cząstkowe, możesz wykorzystać równanie 4.6, aby policzyć je wszystkie naraz. Wektor gradientów, zapisywany jako $\nabla_{\boldsymbol{\theta}} MSE(\boldsymbol{\theta})$, zawiera wszystkie pochodne cząstkowe funkcji kosztu (po jednej dla każdego parametru modelu).

Równanie 4.6. Wektor gradientów funkcji kosztu

$$\nabla_{\theta} MSE(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} MSE(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$



Zwróc uwagę, że powyższy wzór na każdym etapie gradientu prostego przeprowadza obliczenia wobec całego zbioru uczącego \mathbf{X} . Dlatego algorytm ten jest nazywany **wsadowym gradientem prostym** (ang. *batch gradient descent*): wykorzystuje on pełny zbiór danych uczących w każdym przebiegu (prawdopodobnie lepszą nazwą byłaby *pełny gradient prosty*). Wynika z tego, że jest on bardzo powolny w przypadku dużych zestawów danych (niedługo jednak poznamy znacznie szybsze algorytmy gradientu prostego). Jednak algorytm ten znakomicie dostosowuje się do liczby cech; uczenie modelu regresji liniowej za pomocą algorytmu gradientu prostego przebiega znacznie szybciej niż przy użyciu równania normalnego lub rozkładu SVD.

Po wyliczeniu wektora gradientów (będzie on wskazywał jakiś punkt na wzniesieniu funkcji) wystarczy skierować się w przeciwną stronę, aby zacząć schodzić w dół funkcji. W ujęciu matematycznym oznacza to odjęcie wartości $\nabla_{\theta} MSE(\boldsymbol{\theta})$ od $\boldsymbol{\theta}$. Właśnie tutaj przydaje się współczynnik uczenia⁵: przemnażamy go (η) przez wektor gradientów, aby określić długość kroku (równanie 4.7).

Równanie 4.7. Określanie kroku gradientu prostego

$$\boldsymbol{\theta}^{(kolejny\ krok)} = \boldsymbol{\theta} - \eta \nabla_{\theta} MSE(\boldsymbol{\theta})$$

Zobaczmy, jak wygląda implementacja tego algorytmu:

```
eta = 0.1 # Współczynnik uczenia
n_iterations = 1000
m = 100

theta = np.random.randn(2,1) # Losowa inicjacja

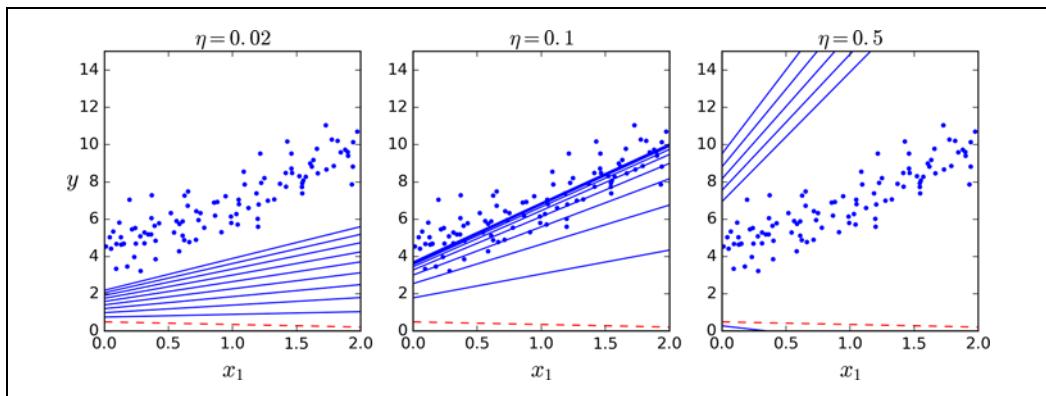
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

To nie było takie trudne! Sprawdźmy, jaką otrzymamy wartość współczynnika θ :

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

⁵ Czyli η (eta) — siódma litera w alfabetie greckim.

Dokładnie taki sam rezultat uzyskaliśmy za pomocą równania normalnego! Algorytm gradientu prostego zadziałał znakomicie. A co by się stało, gdybyśmy użyli innej wartości współczynnika uczenia eta? Na rysunku 4.8 widzimy 10 pierwszych kroków gradientu prostego wykorzystujących trzy różne współczynniki uczenia (linia przerywana ukazuje początkowy etap).



Rysunek 4.8. Różne współczynniki uczenia w algorytmie gradientu prostego

Na lewym wykresie współczynnik uczenia ma zbyt małą wartość: algorytm ostatecznie znajdzie rozwiązanie, ale zajmie mu to mnóstwo czasu. Wartość współczynnika uczenia na środkowym wykresie prezentuje się całkiem nieźle: wystarczyło kilka przebiegów, aby algorytm uzyskał zbieżność. Z kolei na prawym wykresie wyznaczyliśmy zbyt dużą wartość współczynnika uczenia: algorytm staje się rozbieżny z funkcją, „porusza się” po całym układzie współrzędnych i w konsekwencji coraz bardziej oddala się od rozwiązania.

W celu znalezienia optymalnej wartości współczynnika uczenia możemy skorzystać z metody przeszukiwania siatki (opisanej w rozdziale 2.). Jednak możesz chcieć ograniczyć liczbę przebiegów, dzięki czemu metoda przeszukiwania siatki wyeliminuje modele, którym uzyskanie zbieżności zajmuje zbyt wiele czasu.

Zastanawiasz się pewnie, jak wyznaczyć liczbę przebiegów. Jeżeli będzie zbyt mała, algorytm nie zdąży zbliżyć się do optymalnego rozwiązania, natomiast przy zbyt dużej liczbie przebiegów będziesz marnować czas, a parametry modelu nie będą się już zmieniać. Prostym rozwiązaniem jest wyznaczenie dużej liczby przebiegów, a następnie przerwanie działania algorytmu, gdy wartość wektora gradientów stanie się bardzo mała — tzn. wtedy, gdy stanie się mniejsza od wartości parametru ϵ (tzw. tolerancji) — gdyż tak się dzieje, kiedy algorytm gradientu prostego osiąga (niemal) minimum.

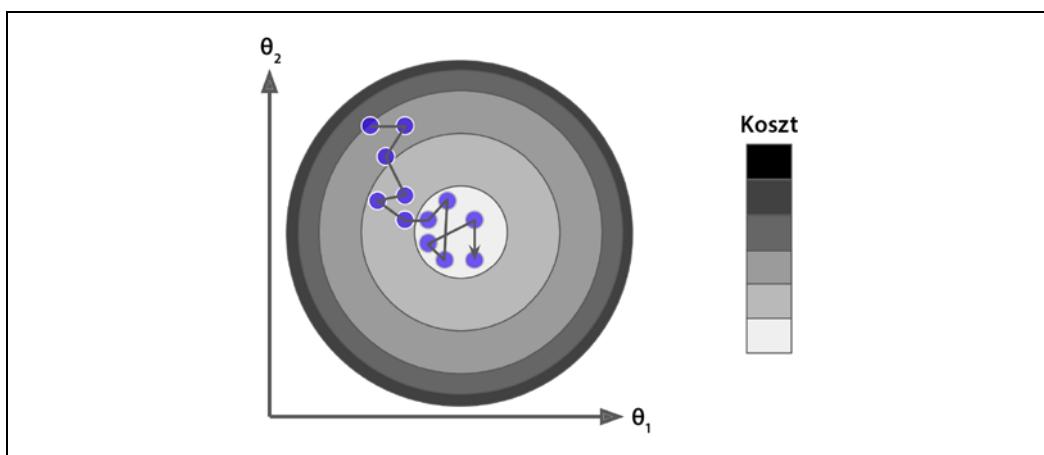
Szybkość uzyskania zbieżności

W przypadku wypukłych funkcji kosztu, których nachylenie nie zmienia się gwałtownie (czyli np. funkcji kosztu MSE), algorytm wsadowego gradientu prostego o stałym współczynniku uczenia końcu osiągnie optymalne rozwiązanie, musimy jednak trochę na to poczekać: uzyskanie optimum w zakresie ϵ może zająć $O(1/\epsilon)$ przebiegów w zależności od kształtu funkcji kosztu. Jeżeli podzielimy wartość tolerancji przez 10 w celu uzyskania precyzyjnego rozwiązania, to algorytm może poświęcić na to jakieś dziesięć razy więcej czasu.

Stochastyczny spadek wzdłuż gradientu

Główny problem algorytmu wsadowego gradientu prostego polega na tym, że wykorzystuje on pełny zbiór danych uczących do obliczania gradientu podczas każdego przebiegu, przez co wraz ze wzrostem rozmiaru zbioru uczącego spada jego szybkość. Jego przeciwieństwo stanowi algorytm **stochastycznego spadku wzdłuż gradientu** (ang. *stochastic gradient descent* — SGD), w którym podczas każdego przebiegu jest dobierana losowa próbka ucząca, za pomocą której są wyliczane gradienty. Oczywiście przetwarzanie pojedynczych przykładów sprawia, że algorytm ten jest znacznie szybszy, gdyż w każdym przebiegu wykorzystuje niewiele danych. Dzięki temu możemy go również używać wobec dużych zestawów danych, ponieważ podczas każdego przebiegu jest przechowywany w pamięci tylko jeden przykład (algorytm SGD możemy zaimplementować również w postaci modelu pozakorowego; zob. rozdział 1.).

Z drugiej strony, w wyniku stochastycznej (tj. losowej) natury algorytm ten jest znacznie bardziej „chaotyczny” od jego wsadowego odpowiednika: funkcja kosztu nie schodzi tu łagodnie w kierunku minimum, lecz przeskakuje raz wyżej, raz niżej i dąży do minimum jedynie po uśrednieniu wyników. Po pewnym czasie w końcu trafi w okolicę bliską minimum, ale nie zatrzyma się i ciągle będzie „skakać” (rysunek 4.9). Zatem po zatrzymaniu algorytmu uzyskane wartości będą dobre, ale nie idealnie optymalne.



Rysunek 4.9. Dzięki algorytmowi stochastycznego spadku wzdłuż gradientu każdy krok uczący staje się znacznie szybszy, ale jednocześnie bardziej losowy w stosunku do algorytmu wsadowego gradientu prostego

W przypadku bardzo nieregularnych funkcji (rysunek 4.6) kosztu rozwiązywanie to może pomóc w ucieczce z minimum lokalnego, zatem algorytm stochastycznego spadku wzdłuż gradientu ma większe szanse na znalezienie minimum globalnego niż wsadowy gradient prosty.

Zatem losowość przydaje się do ucieczki z minimów lokalnych, ale jednocześnie nie pozwala osiągnąć najniższego punktu funkcji kosztu. Jednym z rozwiązań tego problemu jest stopniowe zmniejszanie współczynnika uczenia. Początkowe kroki są duże (przyśpieszamy w ten sposób działanie algorytmu i wydostajemy się z minimów lokalnych), a następnie stają się coraz krótsze, dzięki czemu algorytm pozostaje w okolicach minimum globalnego. Proces ten jest podobny do **symulowanego wyżarzania** (ang. *simulated annealing*), algorytmu przypominającego proces wyżarzania w metalurgii, który

polega na tym, że roztopiony metal jest stopniowo chłodzony. Funkcja określająca współczynnik uczenia w każdym przebiegu jest nazywana **harmonogramem uczenia** (ang. *learning schedule*). Jeżeli zbyt szybko zredukujemy ten współczynnik, możemy utknąć w minimum lokalnym, a nawet pozostać gdzieś w połowie stoku. Z kolei jeżeli harmonogram uczenia będzie malał zbyt powoli, możemy przez długi czas pomijać minimum globalne i uzyskać nieoptymalne rozwiązanie, jeśli zbyt wcześnie zakończymy działanie algorytmu.

Za pomocą poniższego kodu implementujemy algorytm stochastycznego spadku wzduż gradientu wykorzystujący prosty harmonogram uczenia:

```
n_epochs = 50
t0, t1 = 5, 50 # Hiperparametry harmonogramu uczenia

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # Losowa inicjacja

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
```

Zgodnie z konwencją nasz algorytm wykonuje m przebiegów; każdy przebieg jest nazywany **epoką**. Wcześniej używany algorytm wsadowego gradientu prostego wykonywał tysiąc przebiegów wobec pełnego zbioru danych uczących, natomiast tutaj uzyskujemy całkiem niezły wynik już po 50 epokach:

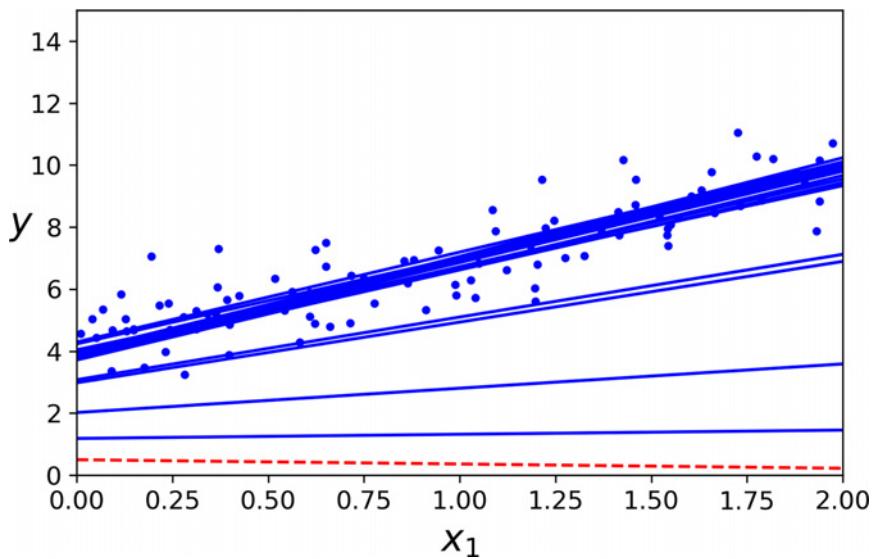
```
>>> theta
array([[4.21076011],
       [2.74856079]])
```

Na rysunku 4.10 widzimy 20 pierwszych etapów uczenia (zwróć uwagę, jak bardzo są one nieregularne).

Zwróć uwagę, że z powodu losowości te same próbki mogą być kilkukrotnie pobierane w danej epoce, natomiast niektóre mogą wcale nie być użyte. Jeżeli chcesz mieć pewność, że zostaną wykorzystane różne dane w każdej epoce, dobrym rozwiązaniem jest przetasowanie wszystkich danych uczących (tak, aby cechy wejściowe i etykiety były wspólnie tasowane), następnie użycie każdego przykładu w danej epoce, przetasowanie zbioru danych przez kolejną epoką itd. Metoda ta jednak spowalnia osiągnięcie zbieżności przez algorytm.



Podczas korzystania z algorytmu stochastycznego spadku wzduż gradientu przykłady uczące muszą być niezależne i o tym samym rozkładzie (ang. *independent and identically distributed* — IIN), dzięki czemu średnio parametry będą dążyły ku optimum globalnemu. Łatwo to osiągnąć, tasując przykłady w trakcie uczenia (tzn. dobierając losowo każdy przykład lub tasując zestaw uczący na początku każdej epoki). Jeżeli nie przetasujesz przykładów (np. dane są posortowane zgodnie z etykietami), to algorytm SGD przeprowadzi optymalizację dla jednej etykiety, następnie dla następnej itd., przez co nie osiągnie minimum globalnego



Rysunek 4.10. Dwadzieścia pierwszych przebiegów algorytmu stochastycznego spadku wzdłuż gradientu

Aby wyuczyć model regresji liniowej za pomocą algorytmu SGD w module Scikit-Learn, możesz wykorzystać klasę SGDRegressor, która domyślnie wykonuje optymalizację wobec funkcji kosztu najmniejszych kwadratów. W poniższym kodzie algorytm będzie działał maksymalnie przez 1000 epok lub do momentu spadku funkcji straty poniżej 0,001 w trakcie jednej epoki (`max_iter=1000, tol=1e-3`). Początkowa wartość współczynnika uczenia wynosi 0,1 (`eta0=0.1`), oraz używamy domyślnego harmonogramu uczenia (różni się on od zastosowanego wcześniej harmonogramu). Na koniec nie korzystamy z żadnej formy regularyzacji (`penalty=None`; wkrótce wróćmy do tego zagadnienia):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

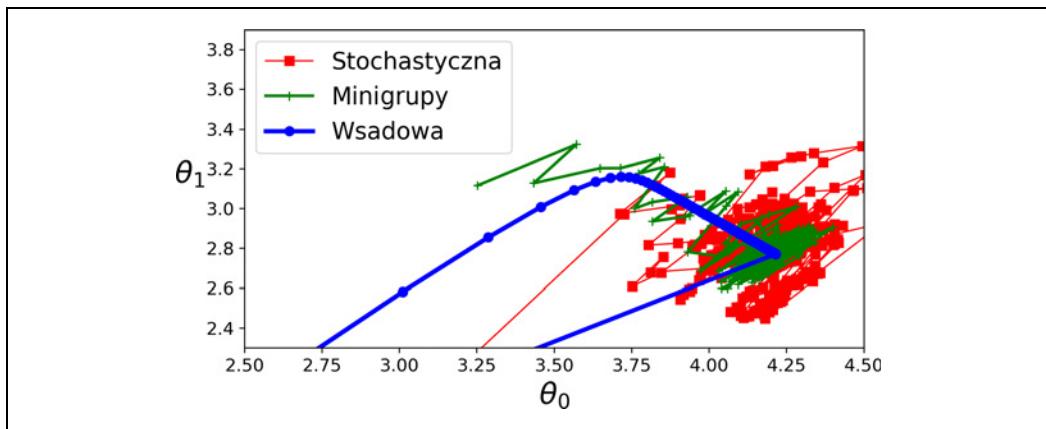
Również tym razem wynik będzie dość zbliżony do uzyskanego za pomocą równania normalnego:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

Schodzenie po gradiencie z minigrupami

Ostatnim rodzajem algorytmu gradientu prostego, jakim się zajmiemy, jest metoda **schodzenia po gradiencie z minigrupami** (ang. *mini-batch gradient descent*). Łatwo go zrozumieć, gdy znamy już pojęcia wsadowego gradientu prostego i stochastycznego spadku wzdłuż gradientu: tutaj w każdym przebiegu nie obliczamy gradientu na podstawie całego zbioru danych (metoda wsadowa) ani pojedynczych próbek (stochastyczny spadek), lecz przy użyciu niewielkich grup losowo dobieranych danych, tzw. **minigrup** lub **minipaczek** (ang. *mini-batches*). W porównaniu do stochastycznego spadku wzdłuż gradientu główną zaletą tej metody jest znaczne przyspieszenie sprzętowej optymalizacji operacji na macierzach, zwłaszcza jeśli korzystamy z procesorów graficznych.

Przebieg algorytmu w przestrzeni parametrycznej jest mniej chaotyczny niż metody SGD, zwłaszcza w przypadku dość dużych minigrup. W wyniku tego dotrze on nieco bliżej minimum od algorytmu stochastycznego. Może jednak mu być trudniej wydostawać się z minimów lokalnych (w przypadku problemów generowanych przez lokalne minima, w przeciwieństwie do modelu regresji liniowej). Na rysunku 4.11 widzimy przebiegi trzech algorytmów gradientu prostego w przestrzeni parametrycznej na etapie uczenia. Wszystkie trzy docierają w okolicę minimum, ale algorytm wsadowego gradientu prostego praktycznie tam się zatrzymuje, natomiast metody stochastyczna i minigrupy „błakają się” dalej. Pamiętaj jednak, że metoda wsadowa potrzebuje znacznie więcej czasu na każdą epokę, a pozostałe dwa algorytmy również są w stanie osiągnąć minimum przy wprowadzeniu odpowiedniego harmonogramu uczenia.



Rysunek 4.11. Ścieżki algorytmów gradientu prostego w przestrzeni parametrycznej

Porównajmy omówione algorytmy, użyte przez nas do uczenia modelu regresji liniowej⁶ (przypomniam, że m oznacza liczbę przykładów uczących, natomiast n definiuje liczbę cech) (tabela 4.1).

Tabela 4.1. Porównanie algorytmów uczących model regresji liniowej

Algorytm	Duża wartość m	Uczenie pozakorowe	Duża wartość n	Hiperparametry	Wymagane skalowanie	Scikit-Learn
Równanie normalne	Szybko	Nie	Powoli	0	Nie	Brak
SVD	Szybko	Nie	Powoli	0	Nie	Linear ↳Regression
Wsadowy gradient prosty	Powoli	Nie	Szybko	2	Tak	SGD ↳Regressor
Stochastyczny spadek wzdłuż gradientu	Szybko	Tak	Szybko	≥ 2	Tak	SGD ↳Regressor
Schodzenie po gradiencie z minigrupami	Szybko	Tak	Szybko	≥ 2	Tak	SGD ↳Regressor

⁶ Równanie normalne może być stosowane wyłącznie wobec modelu regresji liniowej, natomiast, jak się niebawem przekonamy, metody gradientu prostego są używane do trenowania również wielu innych modeli.



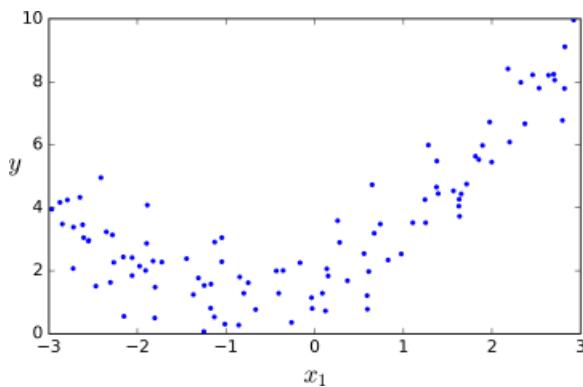
Różnica po wytrenowaniu tymi algorytmami jest niemal niezauważalna: za pomocą każdego z nich uzyskujemy niemal identyczny model, a prognozy są uzyskiwane prawie w taki sam sposób.

Regresja wielomianowa

Co w przypadku, jeśli Twoje dane nie układają się liniowo na wykresie? Co ciekawe, jesteśmy w stanie wytrenować model liniowy do rozpoznawania nieliniowych danych. Najprostszym sposobem jest dodanie potęg każdej cechy w postaci nowej cechy, a następnie wytrenowanie modelu wobec tak rozszerzonego zestawu cech. Technikę tę nazywamy **regresją wielomianową** (ang. *polynomial regression*).

Przeanalizujmy to na przykładzie. Najpierw wygenerujmy jakieś nieliniowe dane przy użyciu prostego **równania kwadratowego**⁷ (oraz dodatkowego szumu) (rysunek 4.12):

```
m = 100  
X = 6 * np.random.rand(m, 1) - 3  
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```



Rysunek 4.12. Wygenerowany zbiór nieliniowych i zaszumionych danych

Wyraźnie widać, że linia prosta nigdy nie będzie dobrze dopasowana do tych danych. Skorzystamy zatem z klasy `PolynomialFeatures` modułu Scikit-Learn, aby przekształcić zbiór danych uczących, i dodamy kwadrat (wielomian drugiego stopnia) każdej cechy jako nową cechę (w naszym przykładzie będzie dodana tylko jedna cecha):

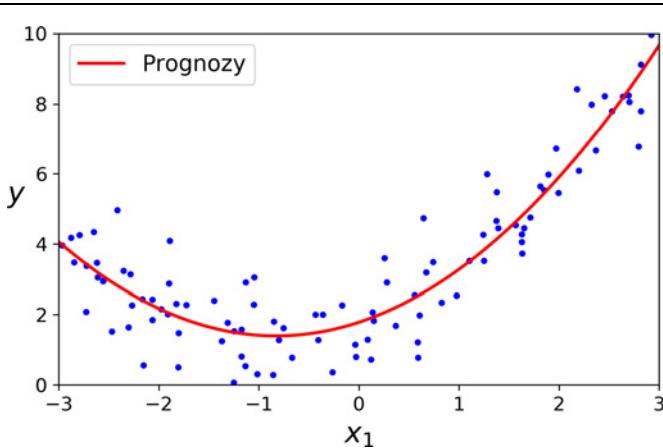
```
>>> from sklearn.preprocessing import PolynomialFeatures  
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)  
>>> X_poly = poly_features.fit_transform(X)  
>>> X[0]  
array([-0.75275929])  
>>> X_poly[0]  
array([-0.75275929, 0.56664654])
```

⁷ Równanie kwadratowe ma postać wzoru $y = ax^2 + bx + c$.

Obiekt `X_poly` zawiera teraz oryginalną cechę X oraz jej kwadrat. Teraz możemy dopasować model `LinearRegression` do tak rozszerzonego zbioru danych uczących (rysunek 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]]))
```

Nieźle: model szacuje wzór $\hat{y} = 0,56x_1^2 + 0,93x_1 + 1,78$, podczas gdy oryginalna funkcja ma postać $y = 0,5x_1^2 + 1,0x_1 + 2,0 + szum\ Gaussowski$.



Rysunek 4.13. Prognozy modelu regresji wielomianowej

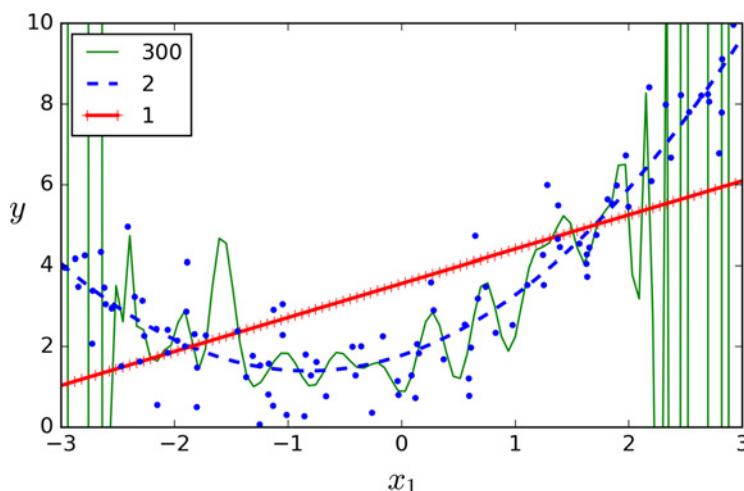
Zauważ, że w przypadku istnienia wielu cech model regresji wielomianowej jest w stanie znajdować powiązania pomiędzy nimi (czego nie jest w stanie dokonać klasyczny model regresji liniowej). Jest to możliwe, ponieważ klasa `PolynomialFeatures` dodaje również kombinacje cech aż do wyznaczonego stopnia. Przykładowo gdybyśmy mieli do czynienia z dwiema cechami, a i b , klasa `PolynomialFeatures` przy założonym parametrze `degree=3` dodałaby nie tylko cechy a^2 , a^3 , b^2 i b^3 , lecz również ich kombinacji ab , a^2b i ab^2 .



Klasa `PolynomialFeatures(degree=d)` przekształca każdą macierz zawierającą n cech w tablicę składającą się z $\frac{(n+d)!}{n!d!}$ cech, gdzie $n!$ oznacza silnię z n (czyli $1 \times 2 \times 3 \times \dots \times n$). Wystrzegaj się kombinatorycznej eksplozji liczby cech!

Krzywe uczenia

Jeżeli przeprowadzasz regresję wielomianową wysokiego stopnia, najprawdopodobniej uda Ci się znacznie łatwiej dopasować model do danych niż za pomocą standardowej regresji liniowej. Na przykład na rysunku 4.14 widzimy efekt regresji liniowej trzysetnego stopnia użytej wobec wygenerowanego uprzednio zbioru danych uczących; porównujemy wyniki ze zwykłą regresją liniową oraz



Rysunek 4.14. Regresja wielomianowa wysokiego stopnia

modelem kwadratowym (wielomianowym drugiego stopnia). Zwróć uwagę, jak bardzo jest „rozbu-jany” wykres regresji trzeciego stopnia w celu zbliżenia się do jak największej liczby próbek uczących.

Taki model wysokiego stopnia w znacznym stopniu ulega przetrenowaniu, z kolei regresja liniowa — niedotrenowaniu. W tym przypadku najlepiej przeprowadza generalizację model kwadratowy, co ma sens, ponieważ zbiór danych został wygenerowany za pomocą równania kwadratowego. Zasadniczo jednak nie będziemy wiedzieć, jaki wzór został wykorzystany do utworzenia zestawu danych, w jaki więc sposób mamy określić złożoność modelu? Skąd mamy wiedzieć, czy dany model ulega przetrenowaniu lub niedotrenowaniu?

W rozdziale 2. korzystaliśmy z metody sprawdzianu krzyżowego do oszacowania wydajności uogólniania modelu. Jeżeli model sprawuje się dobrze wobec danych uczących, ale według wskaźników kroswalidacji nie radzi sobie z generalizowaniem, to znaczy, że ulega przetrenowaniu. W przypadku, gdy model nie radzi sobie zarówno ze zbiorem uczącym, jak i z uogólnianiem, jest niedotrenowany. Jest to jeden ze sposobów na stwierdzenie, czy model jest za prosty lub zbyt złożony.

Inną metodą określenia stopnia złożoności modelu jest analiza **krzywych uczenia** (ang. *learning curves*): są to krzywe wydajności modelu wobec zbioru uczącego i walidacyjnego w funkcji rozmiaru zbioru uczącego (lub przebiegu uczącego). Aby uzyskać taki wykres, wystarczy kilkakrotnie wytrenować model wobec podzbiorów zestawu uczącego cechujących się różnymi rozmiarami. W późniejszym kodzie definiujemy funkcję generującą wykres krzywych uczenia modelu przy użyciu przykładowych danych uczących:

```
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 i in range(1, len(X_train) + 1):
        model.fit(X_train[:i], y_train[:i])
        y_pred = model.predict(X_val)
        train_error = mean_squared_error(y_train[:i], y_pred)
        val_error = mean_squared_error(y_val, y_pred)
        train_errors.append(train_error)
        val_errors.append(val_error)
    plt.plot(range(1, len(X_train) + 1), train_errors, label='Training Error')
    plt.plot(range(1, len(X_train) + 1), val_errors, label='Validation Error')
    plt.xlabel('Number of training samples')
    plt.ylabel('Mean Squared Error')
    plt.legend()
    plt.show()
```

```

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[:m], y_train_predict))
    val_errors.append(mean_squared_error(y_val, y_val_predict))
plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="train")
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")

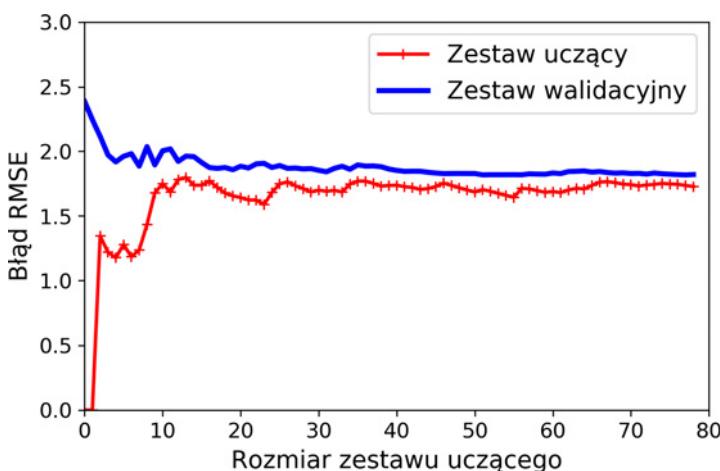
```

Przyjrzyjmy się krzywym uczenia standardowego modelu regresji liniowej (rysunek 4.15):

```

lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)

```



Rysunek 4.15. Krzywe uczenia

Ten niedotrenowany model wymaga nieco dłuższego wyjaśnienia. Najpierw przyjrzyjmy się wydajności wobec zbioru danych uczących: gdy znajdują się w nim tylko jeden lub dwa przykłady, model jest w stanie wyuczyć się ich doskonale, dlatego wykres rozpoczyna się w punkcie 0. Jednak w miarę dodawania kolejnych przykładów zmniejsza się wydajność algorytmu, ponieważ dane stają się zaszurowane i nieliniowe. Z tego powodu błąd dla zbioru uczącego pnie się w górę aż do osiągnięcia wypłaszczenia — od tej chwili dodawanie kolejnych próbek nie będzie ani zmniejszało, ani zwiększało średniego błędu. Przyjrzyjmy się teraz wydajności modelu wobec zbioru walidacyjnego. Jeżeli jest wykorzystywanych tylko kilka przykładów, model nie jest w stanie dobrze uogólniać wyników, przez co błąd walidacyjny jest na samym początku całkiem duży. Następnie wraz z pojawianiem się kolejnych przykładów następuje uczenie modelu, w wyniku czego wartość błędu stopniowo maleje. Jednak również w tym przypadku funkcja liniowa nie opisuje dobrze tych danych, dlatego ostatecznie poziom błędu staje się wypłaszczony, o wartości bardziej zbliżonej do błędu zbioru uczącego.

Te krzywe uczenia są charakterystyczne dla niedotrenowanego modelu. Obydwie osiągnęły wypłaszczenie; znajdują się blisko siebie i mają dość duże wartości.

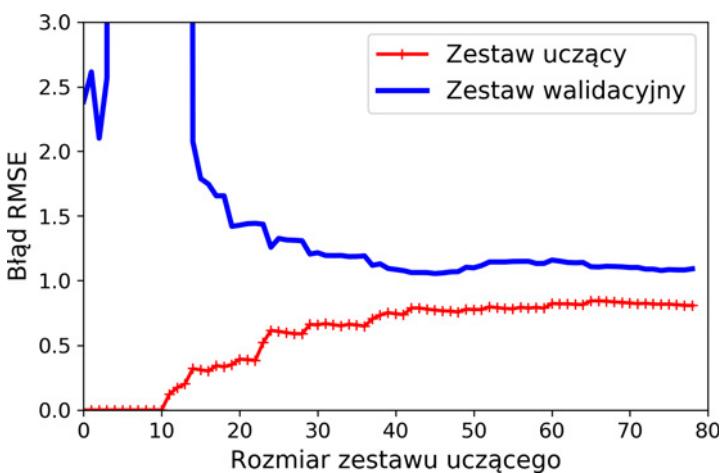


Jeżeli Twój model jest niedotrenowany, dodawanie kolejnych próbek uczących niewiele pomoże. Musisz skorzystać z bardziej złożonego modelu lub wprowadzić lepsze cechy.

Sprawdźmy teraz krzywe uczenia dziesięciostopniowego modelu wielomianowego wobec tych samych danych (rysunek 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)
```



Rysunek 4.16. Krzywe uczenia dziesięciostopniowego modelu wielomianowego

Wykresy te przypominają nieco krzywe uczenia modelu liniowego, możemy jednak dostrzec dwie bardzo istotne różnice:

- Błąd dla zbioru uczącego osiąga znacznie niższy poziom niż w przypadku modelu regresji liniowej.
- Istnieje przerwa pomiędzy obydwoma krzywymi. Oznacza to, że model radzi sobie znacznie lepiej z danymi uczącymi niż ze zbiorem walidacyjnym, co stanowi wyznacznik przetrenowania modelu. Gdybyśmy jednak korzystali ze znacznie większych zestawów danych, odległość pomiędzy krzywymi zmniejszałaby się stopniowo.



Jednym ze sposobów na poprawienie przetrenowanego modelu jest wprowadzenie dodatkowych przykładów aż do zrównania się błędu danych uczących z błędem danych walidacyjnych.

Kompromis pomiędzy obciążeniem a wariancją

Ważną konsekwencją rozważań łączących statystykę z uczeniem maszynowym jest fakt, że błąd generalizacji modelu można wyrazić w postaci sumy trzech odmiennych rodzajów błędów:

- **Obciążenie** (ang. *bias*): ta składowa błędu uogólnienia wynika z nieprawidłowych założeń, np. założenia, że dane są liniowe, podczas gdy w rzeczywistości są opisane funkcją kwadratową. Model o wysokiej wartości obciążenia ma największą szansę na niedotrenowanie wobec danych uczących⁸.
- **Wariancja** (ang. *variance*): ta składowa wynika z nadmiernej czułości modelu na drobne wahania w wartościach danych uczących. Model cechujący się dużą liczbą stopni swobody (np. wielostopniowy model wielomianowy) prawdopodobnie będzie miał dużą wariancję, a przez to będzie przetrenowywany wobec danych uczących.
- **Błąd niereduksywalny** (ang. *irreducible error*): ta składowa stanowi konsekwencję zaszumienia danych. Jedynym sposobem pozbycia się jej jest oczyszczenie danych (np. naprawa źródeł danych takich jak zepsute czujniki lub wykrywanie i usuwanie elementów odstających).

Zwiększanie złożoności modelu zazwyczaj zwiększa jego wariancję i zmniejsza obciążenie. Odwrotny proces następuje w przypadku uproszczenia modelu. Dlatego mówimy tu o kompromisie.

Regularyzowane modele liniowe

Jak widzieliśmy w rozdziałach 1. i 2., dobrym sposobem zmniejszenia przetrenowania modelu jest jego regularyzacja (tzn. ograniczenie): im mniej stopni swobody, tym trudniej przetrenować model wobec danych. Prostym sposobem na regularyzację modelu wielomianowego jest zmniejszenie liczby stopni wielomianu.

W przypadku modelu liniowego regularyzacja jest przeważnie osiągana poprzez ograniczenie wag modelu. Przyjrzymy się teraz modelom regresji grzbietowej, regresji metodą LASSO i metodzie elastycznej siatki, w których stosowane są różne metody ograniczania wag.

Regresja grzbietowa

Regresja grzbietowa (ang. *ridge regression*, zwana także **regularyzacją Tichonowa** — ang. *Tikhonov regularization*) stanowi regularyzowaną odmianę regresji liniowej: do funkcji kosztu zostaje dodany **człon regularizacyjny** o postaci $\alpha \sum_{i=1}^n \theta_i^2$. W ten sposób model jest zmuszony nie tylko do dopasowywania się do danych, lecz także do utrzymywania jak najmniejszych wartości wag. Zwróć uwagę, że człon regularizacyjny powinien zostać dodany do funkcji kosztu jedynie na etapie nauki. Po wytrenowaniu modelu chcemy ocenić wydajność modelu za pomocą nieregularyzowanego wskaźnika wydajności.

Z pomocą hiperparametru α określamy stopień regularizacji modelu. Jeżeli $\alpha = 0$, to mamy do czynienia ze standardową regresją liniową. W przypadku bardzo dużych wartości hiperparametru α wartości

⁸ Nie należy mylić błędu obciążenia z punktem obciążenia (ang. *bias term*) modeli liniowych.

wszystkich wag są bardzo bliskie zeru, w wyniku czego uzyskujemy prostą przebiegającą wzduż średniej zbioru danych. Równanie 4.8 przedstawia funkcję kosztu regresji grzbietowej⁹.

Równanie 4.8. Funkcja kosztu regresji grzbietowej

$$J(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$



Dość powszechnie się zdarza, że funkcja kosztu użyta w czasie uczenia różni się od po-miaru wydajności zastosowanego na etapie testowania. Powodem różnic może być nie tylko regularyzacja, lecz także fakt, że dobra funkcja kosztu użyta podczas uczenia powinna mieć nadające się do optymalizacji pochodne, natomiast wskaźnik wydaj-ności stosowany podczas testowania powinna być jak najbliższy ostatecznemu celowi. Na przykład klasyfikatory są często trenowane za pomocą takiej funkcji kosztu jak logarytmiczna funkcja straty (do której powrócimy za chwilę), ale oceniane przy użyciu precyzji/pełności.

Zwróć uwagę, że punkt obciążenia θ_0 nie jest regularyzowany (suma rozpoczyna się od $i = 1$, nie $i = 0$). Jeśli zdefiniujemy w jako wektor wag cech (od θ_1 do θ_n), to człon regularizacyjny będzie równy $\frac{1}{2} (\|w\|_2)^2$, gdzie $\|w\|_2$ oznacza normę ℓ_2 wektora wag¹⁰. W przypadku gradientu prostego po prostu dodajemy człon αw do wektora gradientów MSE (równanie 4.6).



Przed przeprowadzeniem regresji grzbietowej należy przeskalać dane (np. za pomocą klasy StandardScaler), gdyż metoda ta jest czuła na skalę cech wejściowych. Dotyczy to większości regularyzowanych modeli.

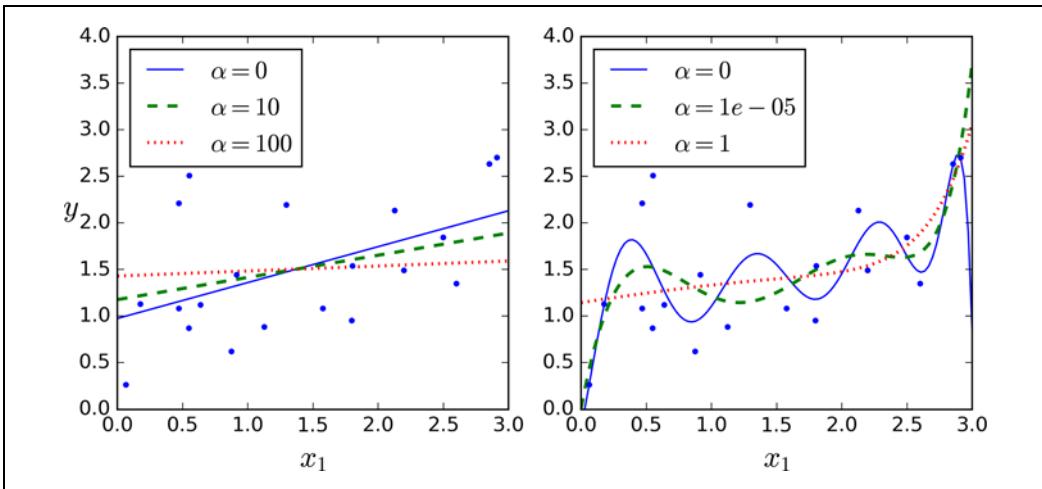
Rysunek 4.17 prezentuje kilka modeli regresji grzbietowej wyuczonych wobec danych liniowych; modele te różnią się wartościami hiperparametru α . Na lewym wykresie używamy standardowych modeli regresji grzbietowej, dzięki czemu uzyskujemy liniowe prognozy. Na prawym wykresie rozszerzamy najpierw dane za pomocą klasy PolynomialFeatures (degree=10), następnie je skalujemy (StandardScaler), a na koniec stosujemy regresję grzbietową wobec otrzymanych cech: mamy tu do czynienia z regresją wielomianową z regularizacją grzbietową. Zwróć uwagę, że wzrost wartości hiperparametru α prowadzi do uzyskania gładzych (mniej poszarpanych, rozsądniejszych) predykcji; zmniejszamy w ten sposób wariancję, ale zwiększymy obciążenie modelu.

Podobnie jak w przypadku regresji liniowej, możemy przeprowadzać regularyzację grzbietową przy użyciu jawnego równania lub stosując metodę gradientu prostego. Również wady i zalety obydwu rozwiązań są takie same. Jawni wzór pokazuję w równaniu 4.9, gdzie A stanowi **macierz jednostkowa**¹¹ o rozmiarze $(n+1) \times (n+1)$; wyjątek stanowi w niej wartość 0 umieszczona w lewej górnej komórce — reprezentuje ona punkt obciążenia.

⁹ Funkcje kosztu o długich nazwach są często zapisywane jako $J(\theta)$; będziemy korzystać z tej notacji w dalszej części książki. Rodzaj omawianej funkcji kosztu z łatwością wywnioskujesz z kontekstu.

¹⁰ Normy zostały omówione w rozdziale 2.

¹¹ Macierz kwadratowa wypełniona zerami oprócz komórek tworzących główną przekątną (od lewego górnego rogu do prawego dolnego), które zawierają wartość 1.



Rysunek 4.17. Model liniowy (po lewej) i wielomianowy (po prawej) z wyznaczonymi różnymi stopniami regresji grzbietowej

Równanie 4.9. Jawny wzór regresji grzbietowej

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^T \mathbf{y}$$

Poniżej prezentuję sposób przeprowadzania regresji grzbietowej w module Scikit-Learn za pomocą jawnego wzoru (odmiany równania 4.9, w której wykorzystujemy technikę faktoryzacji macierzy zaprojektowaną przez André-Louisa Cholesky'ego):

```
>>> 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])
```

A tu widzimy to samo, tym razem za pomocą stochastycznego spadku wzduż gradientu¹²:

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([-1.47012587])
```

Hiperparametr penalty służy do wyznaczania rodzaju członu regularizacyjnego. Wartość 12 oznacza, że chcemy dodać do funkcji kosztu człon regularizacyjny równy połowie kwadratu normy ℓ_2 wektora wag — czyli, krótko mówiąc, chcemy zastosować regresję grzbietową.

¹² Możesz ewentualnie zastosować klasę Ridge z mechanizmem rozwiązywania (ang. *solver*) sag, czyli algorytm stochastycznego uśrednionego gradientu, który stanowi odmianę standardowego algorytmu SGD. Więcej informacji znajdziesz w prezentacji „Minimizing Finite Sums with the Stochastic Average Gradient Algorithm” (https://www.cs.ubc.ca/~schmidtm/Documents/2014_Google_SAG.pdf) autorstwa Marka Schmidta i in. (Uniwersytet Kolumbijskiej).

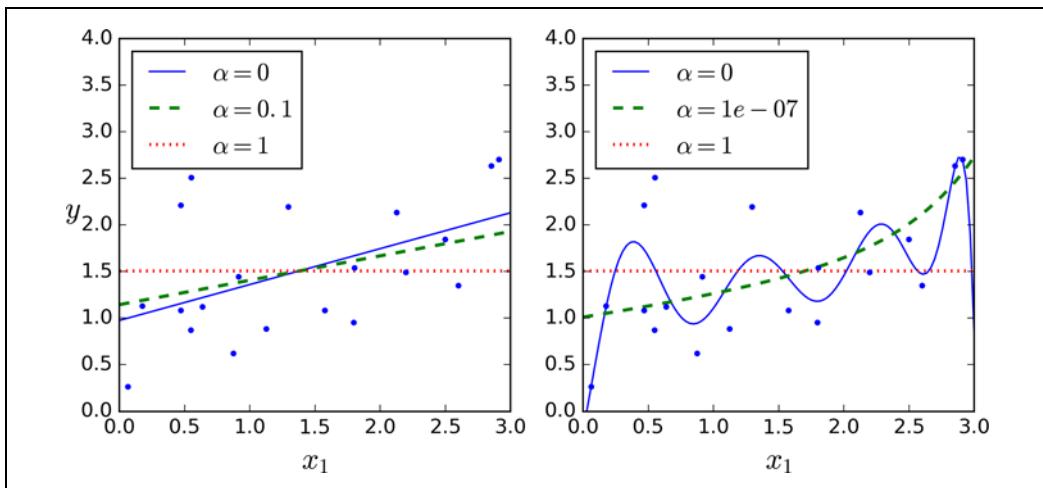
Regresja metodą LASSO

Regresja metodą LASSO (ang. *least absolute shrinkage and selection operator regression* —operator najmniejszej bezwzględnej redukcji i wyboru) stanowi kolejną regularyzowaną odmianę regresji liniowej; podobnie jak w regresji liniowej dodajemy tu człon regularizacyjny do funkcji kosztu, w tym jednak przypadku korzystamy z normy ℓ_1 wektora wag, a nie z połowy kwadratu normy ℓ_2 (równanie 4.10).

Równanie 4.10. Funkcja kosztu regresji metodą LASSO

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \alpha \sum_{i=1}^n |\theta_i|$$

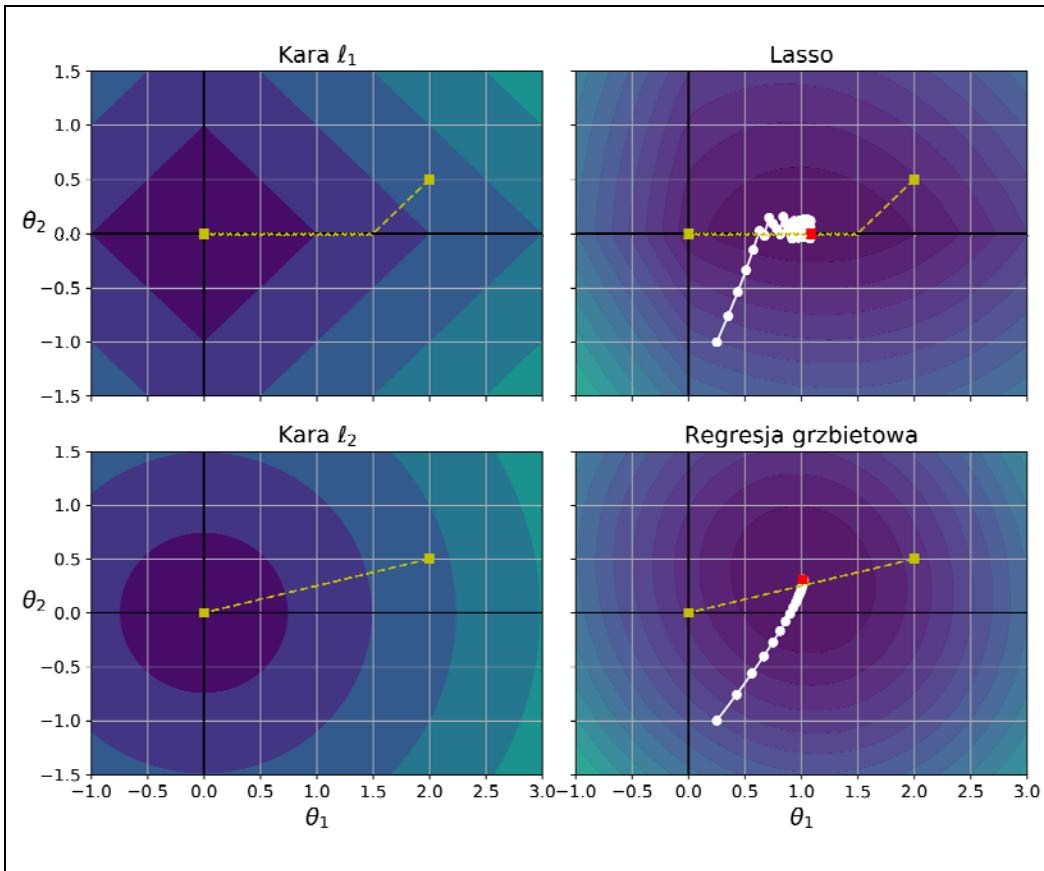
Na rysunku 4.18 wykorzystujemy te same dane, co na rysunku 4.17, ale zastępujemy modele regresji gęrbietowej modelami regresji LASSO i wprowadzamy mniejsze wartości parametru α .



Rysunek 4.18. Model liniowy (po lewej) i wielomianowy (po prawej) z wyznaczonymi różnymi stopniami regresji metodą LASSO

Ważną właściwością regresji metodą LASSO jest fakt, że dąży ona do całkowitej eliminacji wag w najmniej istotnych cechach (np. poprzez zmianę ich wartości na 0). Przykładowo, linia przerywana na prawym wykresie (rysunek 4.18; wykres o wartości $\alpha = 10^{-7}$) przypomina niemalże liniowy wykres funkcji kwadratowej: wszystkie wagi cech wielomianowych wyższego stopnia są równe zero. Innymi słowy, regresja metodą LASSO automatycznie przeprowadza dobór cech i generuje **model rzadki** (tj. zawierający niewiele niezerowych wag cech).

Gdy spojrzesz na rysunek 4.19, zrozumiesz, dlaczego jest to takie ważne: osie reprezentują dwa parametry modelu, natomiast kontury tła ukazują różne funkcje straty. Na lewym górnym wykresie kontury tła symbolizują funkcję straty $\ell_1(|\theta_1| + |\theta_2|)$, która maleje liniowo w miarę zbliżania się do dowolnej osi. Na przykład jeśli zainicjujesz model z parametrami $\theta_1 = 2$ i $\theta_2 = 0,5$, algorytm gradientu prostego będzie równomiernie zmniejszał obydwa parametry (co zostało pokazane za pomocą żółtej linii przerywanej), zatem parametr $\theta_2 = 0,5$ osiągnie wartość 0 jako pierwszy (ponieważ od początku znajdował się bliżej zera). Następnie algorytm będzie pędził, dopóki θ_1 nie osiągnie wartości 0 (algorytm



Rysunek 4.19. Porównanie regularyzacji grzbietowej i metody LASSO

będzie trochę „skakał”, gdyż gradienty ℓ_1 nigdy nie zbliżają się do 0 — dla każdego parametru uzyskują one wartość -1 albo 1). Na prawym górnym wykresie kontury przedstawiają funkcję kosztu LASSO (tzn. funkcję kosztu MSE wraz z funkcją straty ℓ_1). Małe białe kółka pokazują ścieżkę gradientu prostego dającą do optymalizowania niektórych parametrów modelu, które zostały zainicjowane z wartościami ok. $\theta_1 = 0,25$ i $\theta_2 = -1$ — zwróć uwagę, że również tutaj algorytm dociera szybko do $\theta_2 = 0$, a następnie znowu zaczyna „skakać” wokół optimum globalnego (symbolizowanego przez czerwony kwadrat). Gdybyśmy zwiększyli wartość parametru α , optimum globalne zostałoby przesunięte w lewo wraz z żółtą linią przerwywaną, natomiast w przypadku zmniejszenia wartości α optimum globalne przemieściłoby się w prawo (w tym przykładzie optymalne parametry nieregularyzowanej funkcji MSE przyjmują wartości $\theta_1 = 2$ i $\theta_2 = 0,5$).

Dwa dolne wykresy przedstawiają to samo, ale regularyzację ℓ_1 zastępujemy karą ℓ_2 . Na lewym dolnym wykresie widzimy, że funkcja straty ℓ_2 maleje w miarę oddalania się od środka układu współrzędnych, zatem algorytm gradientu prostego dąży wprost w kierunku tego punktu. Kontury wiadoczne na prawym dolnym wykresie reprezentują funkcję kosztu regresji grzbietowej (funkcję kosztu MSE wraz z funkcją straty ℓ_2). Występują tu dwie główne różnice w stosunku do metody

LASSO. Po pierwsze, gradienty maleją w miarę zbliżania się do optimum globalnego, zatem algorytm gradientu prostego zwalnia w naturalny sposób, co pomaga w uzyskaniu zbieżności (gdyż nie występuje zjawisko „skakania”). Po drugie, optymalne parametry (symbolizowane czerwonym kwadratem) stopniowo zbliżają się do środka układu współrzędnych w miarę zwiększania parametru α , ale nigdy nie zostają całkowicie wyeliminowane.



Aby w metodzie gradientu prostego uniknąć „skakania” wokół optimum pod koniec realizowania algorytmu LASSO, musisz stopniowo zmniejszać wartość współczynnika uczenia w trakcie uczenia modelu (nie wyeliminujemy całkowicie tego zjawiska, ale kolejne kroki będą coraz mniejsze aż do uzyskania zbieżności).

Funkcja kosztu LASSO nie jest różniczkowalna w punkcie $\theta_i = 0$ (dla $i = 1, 2, \dots, n$), ale algorytm gradientu prostego będzie dobrze działał, jeśli użyjesz **wektora podgradientów g** ¹³, gdy dowolny $\theta_i = 0$. Równanie 4.11 zawiera wzór wektora podgradientów, który możemy wykorzystać w algorytmie gradientu prostego wraz z funkcją kosztu LASSO.

Równanie 4.11. Wektor podgradientów w regresji metodą LASSO

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} MSE(\boldsymbol{\theta}) + \alpha \begin{pmatrix} sign(\theta_1) \\ sign(\theta_2) \\ \vdots \\ sign(\theta_n) \end{pmatrix} \text{ gdzie } sign(\theta_i) = \begin{cases} -1 & \text{jesli } \theta_i < 0 \\ 0 & \text{jesli } \theta_i = 0 \\ +1 & \text{jesli } \theta_i > 0 \end{cases}$$

Poniżej prezentuję mały przykład użycia klasy Lasso w module Scikit-Learn. Zauważ, że możesz użyć zamiast tego klasy SGDRegressor(penalty="l1").

```
>>> 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])
```

Zauważ, że możesz użyć zamiast tego klasy SGDRegressor(penalty="l1").

Metoda elastycznej siatki

Metoda elastycznej (ang. *elastic net*) siatki stanowi rozwiązanie pośrednie pomiędzy regresją grzbietową a metodą LASSO. Człon regularizacyjny w tym przypadku tworzy prosta kombinacja członów obydwu wspomnianych technik, my zaś możemy nim sterować za pomocą współczynnika proporcji r . Gdy $r = 0$, metoda elastycznej siatki staje się równoważna regresji grzbietowej, natomiast przy $r = 1$ zachowuje się jak metoda LASSO (równanie 4.12).

¹³ Możesz traktować wektor podgradientów w nieróżniczkowalnym punkcie jako wektor pośredni wektorów gradientów wokół tego punktu.

Równanie 4.12. Funkcja kosztu metody elastycznej siatki

$$J(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^n \theta_i^2$$

Kiedy więc należy stosować regresję liniową (tj. nieregularyzowaną), a kiedy metodę grzbietową, LASSO albo elastyczną siatkę? Niemal zawsze warto wprowadzić chociaż odrobinę regularyzacji, dlatego staraj się unikać zwykłej regresji liniowej. Dobrym domyślnym rozwiązańiem okazuje się regresja grzbietowa, ale jeśli podejrzewasz, że będzie przydatnych tylko kilka cech, lepiej wybrać metodę LASSO lub elastycznej siatki, ponieważ, jak już wiemy, dążą one do zerowania wag najmniej użytecznych cech. Generalnie bardziej preferowana jest metoda elastycznej siatki, gdyż metoda LASSO zaczyna nieprawidłowo działać, gdy liczba cech przewyższa liczbę próbek uczących lub gdy istnieje silna korelacja pomiędzy kilkoma cechami.

Poniżej prezentuję przykład użycia klasy ElasticNet (parametr l1_ratio odpowiada współczynnikowi proporcji r):

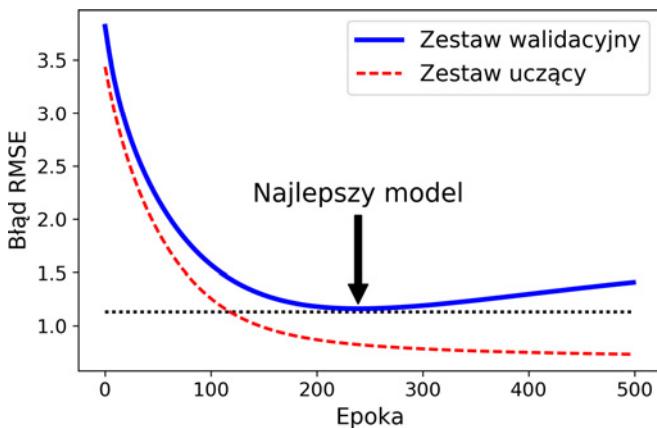
```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.5433232])
```

Wczesne zatrzymywanie

Zupełnie innym mechanizmem regularyzacji iteracyjnych algorytmów uczących, takich jak metody gradientu prostego, jest zakończenie uczenia w momencie osiągnięcia minimum błędu walidacyjnego. Jest to tak zwane **wczesne zatrzymywanie** (ang. *early stopping*). Rysunek 4.20 przedstawia złożony model (w tym przypadku model regresji wielomianowej wysokiego stopnia), który jest trenowany za pomocą algorytmu wsadowego gradientu prostego. W miarę upływu kolejnych epok algorytm uczy się, a jego błąd predykcji (RMSE) wobec zestawu uczącego maleje w naturalny sposób, wraz z błędem prognozowania wobec danych walidacyjnych. Jednak po pewnym czasie błąd prognozowania przestaje maleć i zaczyna znów rosnąć. Jest to wyraźny znak, że model zaczyna ulegać przetrenowaniu. Dzięki wczesnemu zatrzymywaniu przerywamy proces uczenia w momencie osiągnięcia minimalnej wartości błędu predykcji. Jest to tak prosta i skuteczna technika regularyzacji, że Geoffrey Hinton określił ją jako „piękny, darmowy obiad”.



W przypadku metod stochastycznego spadku wzduż gradientu i schodzenia po gradiencie z minigrupami generowane krzywe nie są takie gładkie i może być trudno określić, czy algorytm osiągnął minimum. Jedno z rozwiązań polega na zatrzymaniu procesu uczenia, gdy zauważymy, że błąd walidacyjny już od pewnego czasu biegnie w pobliżu wartości minimalnej (gdy będziemy pewni, że model nie będzie już lepiej działał), a następnie wprowadzeniu parametrów modelu występujących w momencie osiągnięcia minimalnej wartości błędu.



Rysunek 4.20. Regularyzacja metodą wczesnego zatrzymywania

Tak wygląda implementacja metody wczesnego zatrzymywania:

```
from sklearn.base import clone

# Przygotowuje dane
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(max_iter=1, tol=-np.infty, 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) # Kontynuuje z miejsca, w którym zostało przerwane działanie
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val, y_val_predict)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Zwrót uwagi, że dzięki parametrowi `warm_start=True` po wywołaniu metody `fit()` model będzie kontynuował uczenie od miejsca, w którym zostało przerwane.

Regresja logistyczna

W rozdziale 1. wyjaśniłem, że pewne algorytmy regresyjne można stosować w zadaniach klasyfikujących (i odwrotnie). **Regresja logistyczna** (ang. *logistic regression*, zwana także **regresją logitową** – ang. *logit regression*) służy powszechnie do szacowania prawdopodobieństwa przynależności przykładowi do określonej klasy (np. jakie jest prawdopodobieństwo, że dana wiadomość jest

spamem?). Jeżeli oszacowane prawdopodobieństwo przekracza 50%, to model prognozuje, że próbka ta należy do tej klasy (zwanej klasą pozytywną; jest ona oznaczona etykietą 1), w przeciwnym razie stwierdza, że przykład nie stanowi części określonej klasy (tzn. że przynależy do klasy negatywnej, oznaczanej etykietą 0). Model ten wykazuje więc cechy klasyfikatora binarnego.

Szacowanie prawdopodobieństwa

Jaki jest więc mechanizm działania modelu regresji logistycznej? Podobnie jak w przypadku regresji liniowej, w modelu regresji logistycznej wyliczamy ważoną sumę cech wejściowych (wraz z punktem obciążenia), nie są jednak wyświetlane bezpośrednie wyniki, lecz **funkcja logistyczna** rezultatu (równanie 4.13).

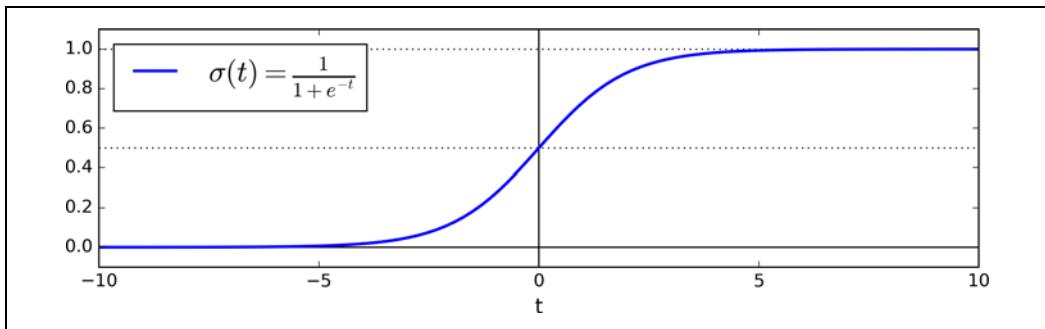
Równanie 4.13. Szacowane prawdopodobieństwo w modelu regresji logistycznej (postać wektorowa)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{x}^T \boldsymbol{\theta})$$

Funkcja logistyczna zapisywana w postaci $\sigma(\cdot)$ stanowi **funkcję sigmoidalną** (tzn. jej przebieg przypomina literę S), której wartości mieszczą się w przedziale pomiędzy 0 i 1. Została ona zaprezentowana w równaniu 4.14 i na rysunku 4.21.

Równanie 4.14. Funkcja logistyczna

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$



Rysunek 4.21. Wykres funkcji logistycznej

Po oszacowaniu prawdopodobieństwa $\hat{p} = h_{\theta}(\mathbf{x})$ przynależności przykładu \mathbf{x} do pozytywnej klasy model regresji logistycznej z łatwością może obliczyć prognozę \hat{y} (równanie 4.15).

Równanie 4.15. Prognoza modelu regresji logistycznej

$$\hat{y} = \begin{cases} 0 & \text{jeśli } \hat{p} < 0,5 \\ 1 & \text{jeśli } \hat{p} \geq 0,5 \end{cases}$$

Zwróc uwagę, że $\sigma(t) < 0,5$, gdy $t < 0$, a $\sigma(t) \geq 0,5$, gdy $t \geq 0$, zatem model regresji logistycznej przewiduje 1, jeżeli wynik $\mathbf{x}^T \boldsymbol{\theta}$ jest dodatni, a 0, jeżeli jest ujemny.



Wynik t często jest nazywany logitem. Nazwa wywodzi się z faktu, że funkcja logitowa, zdefiniowana jako $\text{logit}(p) = \log(p/(1-p))$, stanowi odwrotność funkcji logistycznej. Rzeczywiście, jeśli obliczysz logit szacowanego prawdopodobieństwa p , okaże się, że wynikiem jest t . Możesz spotkać się również z nazwą **logarytm szans** (ang. *log odds*), gdyż jest to logarytm ilorazu szacowanego prawdopodobieństwa klasy pozytywnej i szacowanego prawdopodobieństwa klasy negatywnej.

Funkcje ucząca i kosztu

Wiesz już, w jaki sposób model regresji logistycznej oszacowuje prawdopodobieństwa i wylicza prognozy. Jak jednak możemy go uczyć? Celem uczenia jest dobranie takiego wektora parametrów $\boldsymbol{\theta}$, żeby model szacował wysokie prawdopodobieństwo dla pozytywnych próbek ($y = 1$), a niskie dla przykładów negatywnych ($y = 0$). Koncepcja ta znajduje odzwierciedlenie w funkcji kosztu dla pojedynczego przykładu uczącego x (równanie 4.16).

Równanie 4.16. Funkcja kosztu dla pojedynczej próbki uczącej

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{jeśli } y = 1 \\ -\log(1 - \hat{p}) & \text{jeśli } y = 0 \end{cases}$$

Taka funkcja kosztu ma sens, ponieważ $-\log(t)$ osiąga bardzo duże wartości, gdy t dąży do zera, zatem koszt będzie duży, jeśli model oszacuje prawdopodobieństwo bliskie zeru dla klasy pozytywnej; będzie on osiągać również znaczne wartości, gdy wartość prawdopodobieństwa będzie bliska 1 dla klasy negatywnej. Z drugiej strony, funkcja $-\log(t)$ będzie dążyła do zera, gdy t będzie zbliżało się do 1, więc koszt będzie bliski零, jeśli oszacowane prawdopodobieństwo przynależności do klasy negatywnej będzie niemal zerowe lub bardzo bliskie 1 dla klasy pozytywnej, a przecież dokładnie o to nam chodzi.

Funkcja kosztu dla całego zbioru uczącego stanowi średni koszt wyliczony dla każdego przykładu w zbiorze. Możemy to zapisać w postaci pojedynczego wzoru, zwanego **logarytmiczną funkcją straty** (ang. *log loss*), zaprezentowanego w równaniu 4.17.

Równanie 4.17. Funkcja kosztu regresji logistycznej (logarytmiczna funkcja straty)

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

Zła wiadomość jest taka, że nie znamy jawnego wzoru pozwalającego na obliczanie wartości θ minimalizującej tę funkcję kosztu (nie istnieje odpowiednik równania normalnego). Dobra wieść jest taka, że funkcja ta jest wypukła, zatem metodą gradientu prostego (lub dowolnym innym algorytmem optymalizacji) z pewnością znajdziemy minimum lokalne (pod warunkiem że współczynnik uczenia nie jest za duży, a my mamy odpowiednio dużo czasu). Równanie 4.18 ukazuje pochodne cząstkowe funkcji kosztu dla j-tego parametru θ_j .

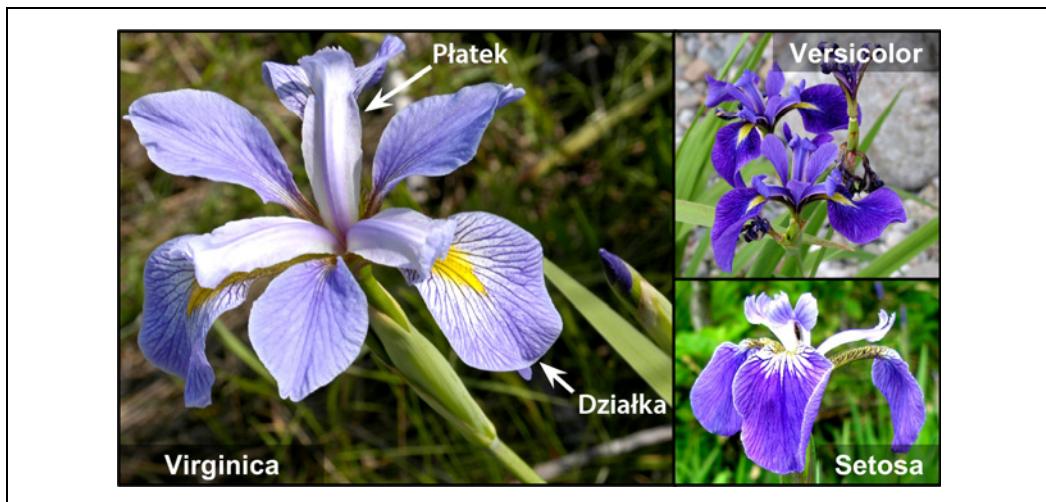
Równanie 4.18. Pochodne cząstkowe logistycznej funkcji kosztu

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

Wzór ten bardzo przypomina równanie 4.5: dla każdej próbki jest wyliczany błąd predykcji, który zostaje następnie pomnożony przez wartość j-tej cechy, po czym następuje wyliczenie wartości średniej ze wszystkich przykładów uczących. Po uzyskaniu wektora gradientów zawierającego wszystkie pochodne cząstkowe możemy go użyć w algorytmie wsadowego gradientu prostego. I to by było na tyle — wiesz już, jak wyuczyć model regresji logistycznej. W przypadku metody stochastycznego spadku wzgórza pobieralibyśmy po jednej próbce, a w metodzie schodzenia po gradiencie z minigrupami korzystalibyśmy z niewielkich podzbiorów uczących.

Granice decyzyjne

Model regresji logistycznej zilustrujemy na przykładzie zbioru danych *Iris*. Jest to słynny zbiór danych zawierający parametry (długość i szerokość płatków oraz działałek kielicha) 150 kwiatów z trzech gatunków kosaćca: *Iris setosa*, *Iris versicolor* i *Iris virginica* (rysunek 4.22).



Rysunek 4.22. Kwiaty trzech gatunków kosaćca¹⁴

Spróbujmy stworzyć klasyfikator rozpoznający gatunek *Iris virginica* jedynie na podstawie szerokości płatka. Najpierw wczytajmy dane:

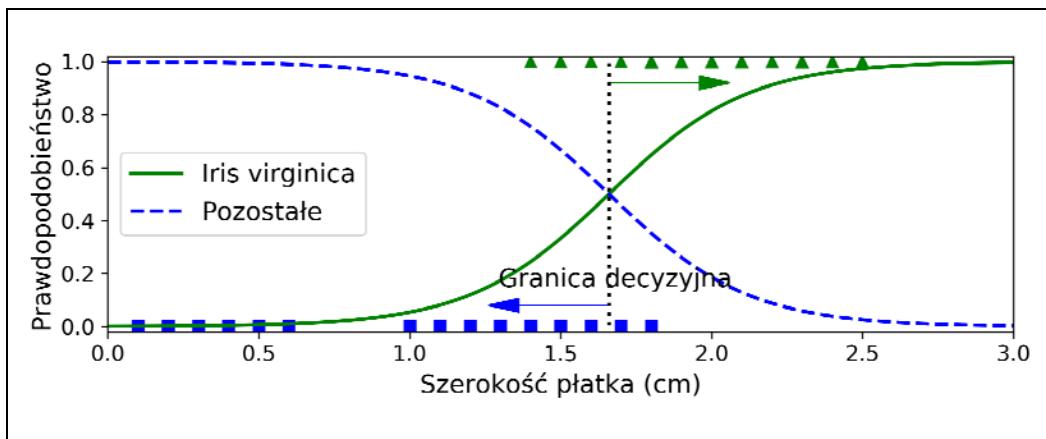
```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> list(iris.keys())  
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'target']  
>>> X = iris["data"][:, 3:] # Szerokość płatka  
>>> y = (iris["target"] == 2).astype(np.int) # 1, jeśli wykryje Iris virginica, w przeciwnym razie 0
```

¹⁴ Zdjęcia pochodzą ze stron serwisu Wikipedia poświęconych poszczególnym gatunkom omawianej rośliny. Fotografia kwiatu *Iris virginica* została wykonana przez Franka Mayfielda (licencja Uznanie autorstwa — na tych samych warunkach 2.0, <https://creativecommons.org/licenses/by-sa/2.0/deed.pl>), autorem zdjęcia przedstawiającego *Iris versicolor* jest D. Gordon E. Robertson (Uznanie autorstwa — na tych samych warunkach 3.0, <https://creativecommons.org/licenses/by-sa/3.0/deed.pl>), natomiast zdjęcie *Iris setosa* stanowi własność publiczną.

Wytrenujmy teraz model regresji logistycznej:

```
from sklearn.linear_model import LogisticRegression  
  
log_reg = LogisticRegression()  
log_reg.fit(X, y)
```

Sprawdźmy oszacowane prawdopodobieństwa przynależności do wyznaczonego gatunku dla kwiatów o szerokości płatków w przedziale od 0 do 3 cm (rysunek 4.23)¹⁵:



Rysunek 4.23. Oszacowane prawdopodobieństwa i granica decyzyjna

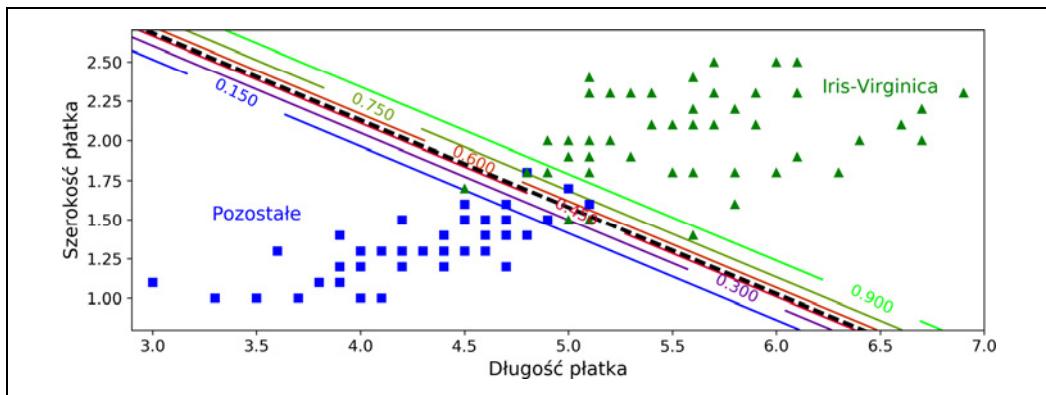
```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)  
y_proba = log_reg.predict_proba(X_new)  
plt.plot(X_new, y_proba[:, 1], "g-", label="Iris virginica")  
plt.plot(X_new, y_proba[:, 0], "b--", label="Pozostałe")  
# Dodatkowo nie pokazany w książce kod modułu Matplotlib, dzięki któremu wykres będzie ładnie wyglądał
```

Szerokość płatków w kwiatach *Iris virginica* (reprezentowana przez trójkąty) mieści się w zakresie pomiędzy 1,4 a 2,5 cm, natomiast jest ona zazwyczaj mniejsza u pozostałych gatunków kosaćca (symbolizowanych przez kwadraty) — w granicach od 0,1 do 1,8 cm. Jak widać, szerokości płatków we wszystkich gatunkach nakładają się w pewnym zakresie. W przypadku szerokości powyżej 2 cm klasyfikator jest całkiem pewny, że kwiat przynależy do gatunku *Iris virginica* (uzyskujemy duże prawdopodobieństwo przynależności do klasy *Iris virginica*), natomiast w przypadku węższych płatków (poniżej 1 cm) klasyfikator z dużą pewnością uzna, że dany kwiat nie należy do gatunku *Iris virginica* (duże prawdopodobieństwo przynależności do klasy *Pozostałe*). W przypadku wartości środkowych algorytm klasyfikujący nie jest już taki pewny. Jeśli jednak chcemy przewidzieć klasę (za pomocą metody predict() zamiast predict_proba()), zostanie zwrócona wartość najbardziej prawdopodobnej klasy. Dlatego w okolicy szerokości 1,6 cm istnieje **granica decyzyjna** wyznaczająca prawdopodobieństwo 50% przynależności do którejś z dwóch klas: jeżeli szerokość płatka przekracza 1,6 cm, klasyfikator uzna, że kwiat należy do gatunku *Iris virginica*, w przeciwnym wypadku zaklasyfikuje go do grupy pozostałych gatunków (nawet jeśli nie będzie całkowicie tego pewny):

¹⁵ Funkcja reshape() z modułu NumPy pozwala wyznaczyć jeden wymiar o wartości -1, co oznacza „niezdefiniowany”: wartość zostaje wywnioskowana z długości tablicy i pozostałych wymiarów.

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Na rysunku 4.24 widzimy ten sam zbiór danych, tym razem jednak korzystamy z dwóch cech: szerokości i długości płatka. Klasyfikator regresji logistycznej po wytrenowaniu jest w stanie przy użyciu tych dwóch cech szacować prawdopodobieństwo przynależności nowego przykładu do klasy *Iris-Virginica*. Linia przerywana pokazuje punkty, w których klasyfikator oszacowuje 50% prawdopodobieństwa: jest to granica decyzyjna modelu. Zauważmy, że granica ta jest liniowa¹⁶. Równolegle do niej linie określają punkty wskazujące określone prawdopodobieństwo, począwszy od 15% (lewa dolna część wykresu) aż do 90% (prawa górna część). Zgodnie z tym modelem wszystkie kwiaty znajdujące się po prawej górnej stronie od zielonej linii mają ponad 90% szans na przynależność do gatunku *Iris virginica*.



Rysunek 4.24. Liniowa granica decyzyjna

Podobnie jak w przypadku poprzednich modeli liniowych, jesteśmy w stanie regularyzować algorytm regresji logistycznej za pomocą kar ℓ_1 lub ℓ_2 . Istotnie, moduł Scikit-Learn domyślnie stosuje tu regularyzację ℓ_2 .



W modelu LogisticRegression za siłę regularizacji odpowiada nie hiperparametr alpha (występujący w innych modelach liniowych), lecz jego odwrotność: C. Im większa jego wartość, tym mniejszy stopień regularizacji modelu.

Regresja softmax

Możemy uogólnić model regresji logistycznej do bezpośredniej obsługi wielu klas, bez konieczności uczenia i łączenia wielu klasyfikatorów binarnych (proces ten został omówiony w rozdziale 3.). Tego typu model nosi nazwę **regresji softmax** lub **wielomianowej/wielorakiej regresji logistycznej** (ang. *multinomial logistic regression*).

Jej koncepcja jest prosta: dla danej próbki x model regresji softmax najpierw oblicza wynik $s_k(x)$ dla każdej klasy k , następnie oszacowuje prawdopodobieństwo przynależności do danej klasy, stosując wobec

¹⁶ Jest to zbiór punktów x określanych funkcją $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, której wykres ma przebieg liniowy.

tych wyników **funkcję typu softmax** (zwaną także **znormalizowaną funkcją wykładniczą**). Wzór służący do wyliczania wyników $s_k(\mathbf{x})$ powinien wyglądać znajomo, gdyż jest tożsamy ze wzorem na prognozy regresji liniowej (równanie 4.19).

Równanie 4.19. Wynik funkcji softmax dla klasy k

$$s_k(\mathbf{x}) = \mathbf{x}^T \boldsymbol{\theta}^{(k)}$$

Zwróc uwagę, że każda klasa ma własny, wyspecjalizowany wektor parametrów $\boldsymbol{\theta}^{(k)}$. Wszystkie te wektory są zazwyczaj przechowywane w rzędach **macierzy parametrów $\boldsymbol{\Theta}$** .

Po wyliczeniu wyników każdej klasy dla przykładu \mathbf{x} możemy oszacować prawdopodobieństwo \hat{p}_k przynależności tej próbki do klasy k , przekazując funkcji softmax (równanie 4.20) te wyniki. Dla każdego zostaje wyliczona jego eksponenta, po czym zostaje przeprowadzona normalizacja (otrzymany rezultat dzielimy przez sumę wszystkich eksponentów). Wyniki są zazwyczaj nazywane logitami lub logarytmami szans (chociaż w rzeczywistości są one nienormalizowanymi logarytmami szans).

Równanie 4.20. Funkcja softmax

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

W tym równaniu:

- K — liczba klas,
- $s(\mathbf{x})$ — wektor zawierający wyniki każdej klasy dla przykładu \mathbf{x} ,
- $\sigma(s(\mathbf{x}))_k$ — szacowane prawdopodobieństwo przynależności próbki \mathbf{x} do klasy k przy znanych wynikach wszystkich klas dla tego przykładu.

Podobnie jak w przypadku klasyfikatora regresji logistycznej model regresji softmax prognozuje klasę o najwyższym oszacowanym prawdopodobieństwie (czyli klasę o największym wyniku), o czym możemy przekonać się, spoglądając na równanie 4.21.

Równanie 4.21. Prognoza klasyfikatora regresji Softmax

$$\hat{y} = \arg \max_k \sigma(s(\mathbf{x}))_k = \arg \max_k s_k(\mathbf{x}) = \arg \max_k (\boldsymbol{\theta}^{(k)})^T \cdot \mathbf{x}$$

Operator *argmax* zwraca wartość zmiennej maksymalizującej funkcję. W tym równaniu zwraca ona wartość parametru k maksymalizującą szacowane prawdopodobieństwo $\sigma(s(\mathbf{x}))_k$.



Klasyfikator regresji softmax prognozuje tylko jedną klasę naraz (tzn. jest modelem wieloklasowym, nie wielowyciślowym), dlatego należy go używać wyłącznie ze wzajemnie wykluczającymi się klasami, takimi jak różne gatunki roślin. Nie możemy stosować go do rozpoznawania wielu osób na jednym zdjęciu.

Skoro wiemy już, w jaki sposób model oszacowuje prawdopodobieństwa i przeprowadza prognozy, możemy zabrać się za jego uczenie. Celem trenowania jest tu uzyskanie modelu oszacowującego wysokie prawdopodobieństwo dla klasy docelowej (a zatem niskie prawdopodobieństwo dla pozosta-

łych klas). Minimalizowanie widocznej w równaniu 4.22 funkcji kosztu (zwanej **entropią krzyżową**) powinno nam w tym pomóc, ponieważ w ten sposób model jest karany w przypadku szacowania niewielkiego prawdopodobieństwa dla klasy docelowej. Entropia krzyżowa jest często wykorzystywana do sprawdzania, w jakim stopniu zbiór oszacowanych prawdopodobieństw zgadza się z klasami docelowymi.

Równanie 4.22. Funkcja kosztu — entropia krzyżowa

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(p_k^{(i)})$$

W tym równaniu:

- $y_k^{(i)}$ jest docelowym prawdopodobieństwem przynależności i -tego przykładu do klasy k . Zasadniczo przyjmuje ono wartość 1 lub 0, w zależności od tego, czy dany przykład przynależy do klasy k .

Zwróć uwagę, że jeżeli dostępne są tylko dwie klasy ($K = 2$), ta funkcja kosztu przekształca się w funkcję kosztu regresji logistycznej (logarytmiczną funkcję straty) (równanie 4.17).

Entropia krzyżowa

Pojęcie entropii krzyżowej (ang. *cross entropy*) wywodzi się z teorii informacji. Założymy, że chcesz codziennie skutecznie przesyłać informacje o pogodzie. Jeżeli mamy osiem możliwości (słonecznie, deszczowo itd.), możemy zakodować każdą z nich za pomocą trzech bitów, ponieważ $2^3 = 8$. Jeżeli jednak uważasz, że niemal codziennie będzie słoneczna pogoda, znacznie wydajniejsze byłoby zakodowanie stanu „słonecznie” w postaci tylko jednego bitu (0), a pozostałych siedmiu opcji w formie czterobitowej (pierwszy bit miałby wartość 1). Entropia krzyżowa mierzy uśrednioną liczbę bitów, jaką rzeczywiście przeznaczasz na każdą możliwość. Jeżeli Twoje założenia dotyczące pogody są niezawodne, wartość entropii krzyżowej będzie równa entropii samej pogody (np. jej naturalnej nieprzewidywalności). Jeśli jednak Twoje założenia są nieprawidłowe (np. często pada), wartość entropii krzyżowej będzie większa o czynnik zwany **dywergencją Kullbacka-Leiblera (KL)**.

Entropia krzyżowa pomiędzy dwoma rozkładami prawdopodobieństwa p i q jest definiowana wzorem $H(p, q) = -\sum_x p(x) \log q(x)$ (przynajmniej wtedy, gdy obydwa rozkłady są dyskretnie). Więcej informacji znajdziesz w nagranym przeze mnie filmie poświęconym temu zagadnieniu (<https://www.youtube.com/watch?v=ErfnhcEV1O8>).

Wektor gradientów dla tej funkcji kosztu w odniesieniu do $\theta^{(k)}$ został ukazany w równaniu 4.23.

Równanie 4.23. Wektor gradientów entropii krzyżowej dla klasy k

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Możemy teraz obliczyć wektor gradientów dla każdej klasy, a następnie użyć metody gradientu prostego (lub dowolnego innego algorytmu optymalizacji) do wyszukania macierzy parametrów Θ minimalizującej funkcję kosztu.

Użyjmy regresji softmax do rozdzielenia kwiatów kosaćca na trzy klasy. Klasa LogisticRegression korzysta domyślnie ze strategii OvR podczas trenowania modelu przy użyciu przynajmniej trzech klas, mimo to możemy wyznaczyć wartość multinomial hiperparametru `multi_class` w celu przejścia na metodę regresji softmax. Musimy także zdefiniować mechanizm rozwiązywania obsługujący ten typ regresji, np. `solver="lbfgs"` (więcej szczegółów znajdziesz w dokumentacji modułu Scikit-Learn). Domyślnie jest również wykorzystywana regularyzacja ℓ_2 , którą regulujemy przy użyciu hiperparametru `C`.

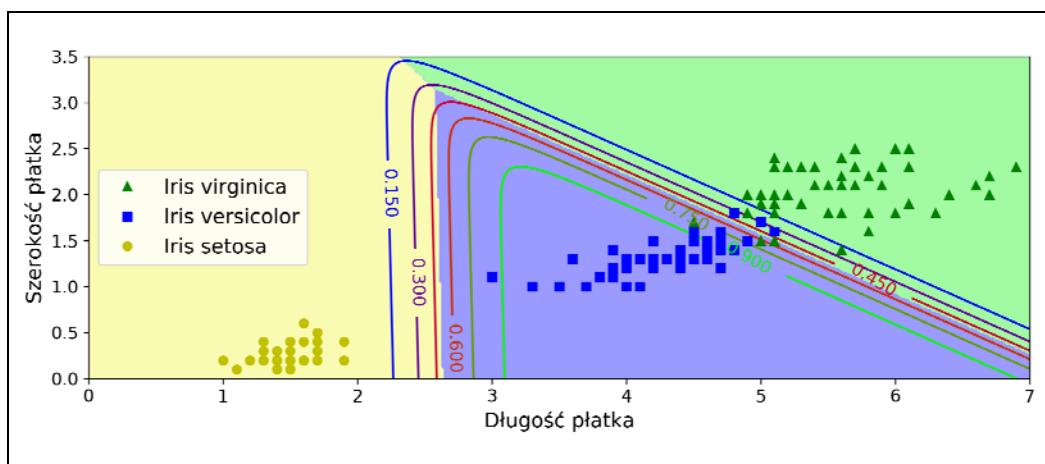
```
X = iris["data"][:, (2, 3)] # Długość płatka, szerokość płatka
y = iris["target"]

softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Teraz, gdy znajdziesz na polu kwiat kosaćca o płatkach długich na 5 cm i szerokich na 2 cm, możesz zapytać swój model, na jaki gatunek trafiasz/trafiłeś, a uzyskasz odpowiedź, że na 94,2% jest to *Iris virginica* (klasa 2.), ewentualnie *Iris versicolor* (ale prawdopodobieństwo wynosi zaledwie 5,8%):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

Na rysunku 4.25 widzimy uzyskane granice decyzyjne, symbolizowane przez różne kolory tła. Zwróć uwagę, że są one liniowe pomiędzy dwiema poszczególnymi klasami. Widzimy ponadto na wykresie prawdopodobieństwa przynależności do klasy *Iris versicolor*, ukazane za pomocą linii (np. linia oznaczona cyfrą 0.450 symbolizuje granicę decyzyjną wyznaczającą 45% przynależności do tejże klasy). Zwróć uwagę, że model może prognozować klasę, dla której zostało oszacowane prawdopodobieństwo mniejsze niż 50%. Przykładowo, w punkcie styku wszystkich trzech granic decyzyjnych prawdopodobieństwo przynależności do którejś klasy wynosi dla każdej z nich po 33%.



Rysunek 4.25. Granice decyzyjne w regresji softmax

Ćwiczenia

1. Którego modelu uczącego regresji liniowej należy użyć, jeżeli zbiór danych uczących zawiera miliony cech?
2. Założmy, że cechy w zbiorze danych uczących mieszczą się w różnych skalach. Którym algorytmom może to zaszkodzić i w jaki sposób? Jak możemy temu zaradzić?
3. Czy podczas uczenia modelu regresji logistycznej algorytm gradientu prostego może utknąć w minimum lokalnym?
4. Czy wszystkie metody gradientu prostego prowadzą do tego samego modelu, jeżeli damy im wystarczająco wiele czasu?
5. Założmy, że korzystamy z algorytmu wsadowego gradientu prostego i stworzyliśmy wykres błędu walidacyjnego w funkcji epoki. Jeżeli wartość tego błędu stale rośnie, co to może oznaczać? Jak można rozwiązać ten problem?
6. Czy zatrzymanie algorytmu schodzenia po gradiencie z minigrupami natychmiast po wykryciu wzrostu wartości błędu walidacyjnego jest dobrym pomysłem?
7. Który z omówionych algorytmów gradientu prostego najszybciej osiągnie optymalne rozwiązanie? Który z nich rzeczywiście uzyska zbieżność? W jaki sposób możesz doprowadzić pozostałe do zbieżności?
8. Założmy, że korzystamy z regresji wielomianowej. Rysujemy wykres krzywych uczenia i zauważamy dużą przerwę pomiędzy wykresem błędu dla danych uczących a wykresem błędu dla danych walidacyjnych. Co się dzieje? Na jakie trzy sposoby możesz rozwiązać ten problem?
9. Założmy, że używamy metody regresji grzbietowej i zauważamy, że błędy danych uczących i walidacyjnych są niemal równe i mają dużą wartość. Czy taki model cechuje się dużą wariancją, czy obciążeniem? Należy w takim wypadku zwiększyć wartość parametru α czy ją zmniejszyć?
10. Dlaczego należy używać:
 - a. algorytmu regresji grzbietowej zamiast standardowej regresji liniowej (tzn. nieregularyzowanej)?
 - b. algorytmu regresji typu LASSO zamiast regresji grzbietowej?
 - c. metody elastycznej siatki zamiast regresji typu LASSO?
11. Założmy, że chcesz klasyfikować zdjęcia wykonywane w pomieszczeniach/na zewnątrz oraz dzienne/nocne. Należały w tym celu zaimplementować dwa klasyfikatory regresji logistycznej czy jeden klasyfikator typu softmax?
12. Zaimplementuj algorytm wsadowego gradientu prostego wykorzystujący mechanizm wczesnego zatrzymywania w modelu regresji softmax (bez korzystania z modułu Scikit-Learn).

Rozwiązanie tych zadań znajdziesz w dodatku A.

Maszyny wektorów nośnych

Maszyna wektorów nośnych (ang. *support vector machine* — SVM) stanowi potężny i wszechstronny model uczenia maszynowego, zdolny do przeprowadzania klasyfikacji liniowej, nielinowej, regresji, a nawet do wykrywania elementów odstających. Jest to jedno z najpopularniejszych rozwiązań w świecie uczenia maszynowego i każdy analityk danych powinien potrafić się nim posługiwać. Maszyny SVM przydają się zwłaszcza do klasyfikowania złożonych, małych lub średnich zbiorów danych.

W tym rozdziale poznasz podstawowe koncepcje związane z maszynami SVM, ich mechanizm działania oraz metody korzystania z nich.

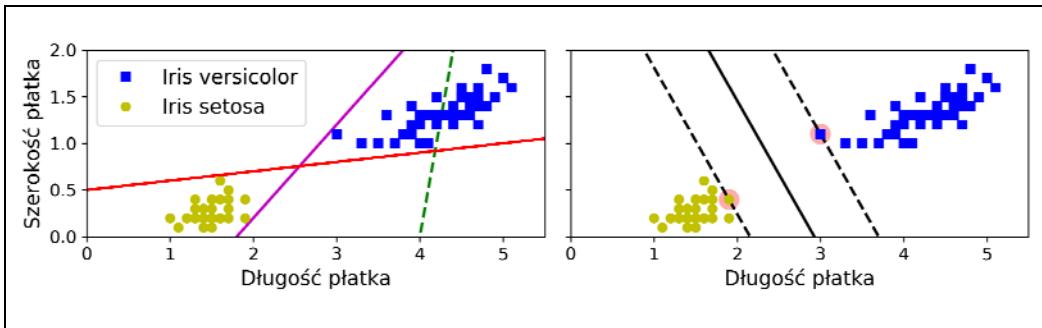
Liniowa klasyfikacja SVM

Podstawy działania maszyn SVM najlepiej wyjaśnić wizualnie. Rysunek 5.1 prezentuje część zbioru danych Iris, którym zajmowaliśmy się pod koniec rozdziału 4. Możemy wyraźnie rozdzielić dwie klasy za pomocą prostej (są one **liniowo rozdzielne/separowalne**). Na lewym wykresie widzimy granice decyzyjne trzech możliwych klasyfikatorów liniowych. Model, którego granica decyzyjna jest symbolizowana linią przerywaną, jest tak zły, że nawet nie rozdziela prawidłowo klas. Pozostałe dwa modele są znakomicie dopasowane do tego zbioru uczącego, ale ich granice decyzyjne znajdują się tak blisko przykładów, że prawdopodobnie nie będą spisywały się dobrze wobec nowych danych. Z kolei na prawym wykresie widzimy granicę decyzyjną klasyfikatora SVM; widoczna linia nie tylko rozdziela obydwie klasy, ale także utrzymuje możliwie duży dystans od najbliższej próbki. Możemy traktować maszynę SVM jako wyznaczanie najszerzej możliwej „ulicy” (symbolizowanej liniami przerywanymi) pomiędzy klasami. Proces ten nazywamy **klasyfikowaniem maksymalnego marginesu** (ang. *large margin classification*).

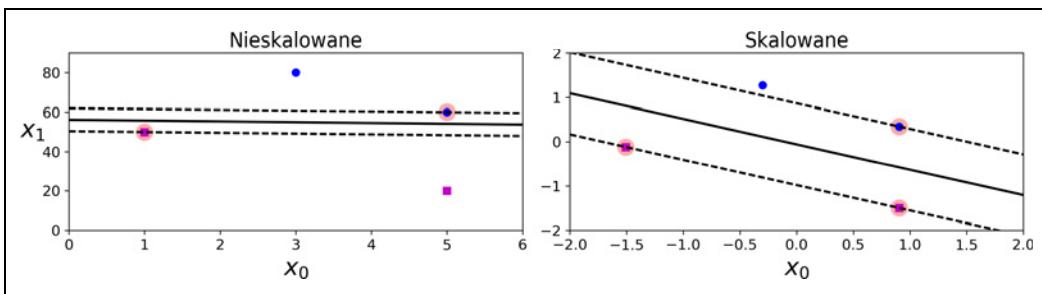
Zwróci uwagę, że dodawanie kolejnych przykładów znajdujących się na zewnątrz marginesu w żaden sposób nie wpłynie na granicę decyzyjną: jest całkowicie określona („noszona”) przez próbki znajdujące się na krańcach „ulicy”. Przykłady te są nazywane **wektorami nośnymi** (ang. *support vectors*; są one ujęte w kółka na rysunku 5.1).



Maszyny SVM są czułe na skale cech, o czym możemy się przekonać, patrząc na rysunek 5.2: na lewym wykresie skala w osi pionowej jest znacznie większa od skali w osi poziomej, dlatego najszerza możliwa „ulica” jest ułożona niemal horyzontalnie. Po przeskalowaniu cech (np. za pomocą klasy StandardScaler) granica decyzyjna na prawym wykresie prezentuje się znacznie lepiej.



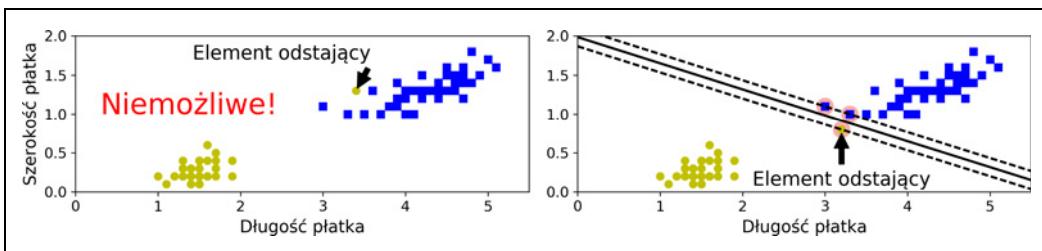
Rysunek 5.1. Klasyfikowanie maksymalnego marginesu



Rysunek 5.2. Czułość na skalę cech

Klasyfikacja miękkiego marginesu

Jeśli rygorystycznie wymusimy, aby wszystkie znajdowały się poza „ulicą” i po prawej stronie wykresu, to będziemy mieli do czynienia z **klasyfikacją twardego marginesu** (ang. *hard margin classification*). Rozwiążanie to cechują dwa problemy. Po pierwsze, działa ono jedynie wtedy, gdy dane są liniowo rozdzielne. Po drugie, jest ono wrażliwe na elementy odstające. Rysunek 5.3 przedstawia zestaw danych *Iris* z umieszczonym tylko jednym elementem odstającym; z tego powodu na lewym wykresie znalezienie twardego marginesu okazuje się niemożliwe, prawy wykres ukazuje zupełnie odmienną granicę decyzyjną od widocznej na rysunku 5.1 — oznacza to, że model prawdopodobnie nie będzie zbyt dobrze generalizował wyników.

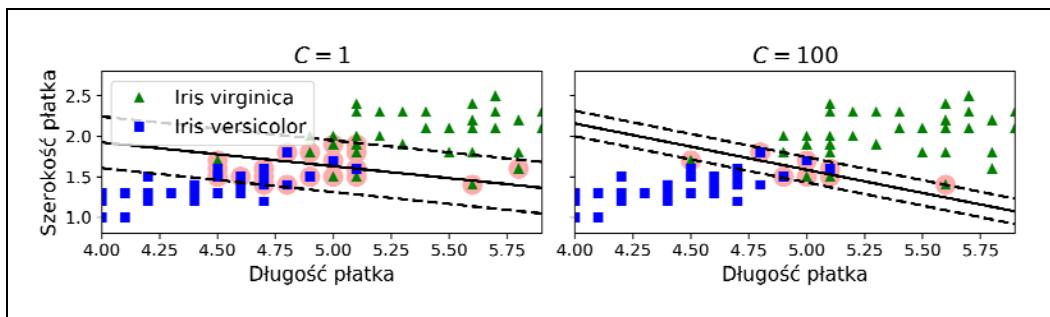


Rysunek 5.3. Wrażliwość klasyfikacji twardego marginesu na elementy odstające

Aby uniknąć tych problemów, skorzystaj z elastyczniejszego modelu. Naszym celem jest znalezienie dobrego kompromisu pomiędzy jak największą szerokością „ulicy” a ograniczeniem **naruszania marginesu**.

su (ang. *margin violation*; np. próbki przekraczające granice marginesu lub nawet trafiające na jego nie właściwą stronę). Jest to tzw. **klasyfikacja miękkiego marginesu** (ang. *soft margin classification*).

Podczas tworzenia modelu SVM za pomocą modułu Scikit-Learn możemy wyznaczyć liczbę hiperparametrów. Jednym z nich jest hiperparametr C. Jeżeli wprowadzimy jego małą wartość, to uzyskamy model widoczny po lewej stronie na rysunku 5.4. W przypadku dużej wartości hiperparametru C otrzymamy model pokazany po prawej stronie. Przekroczenia marginesów są złe. Zazwyczaj staramy się, żeby było ich jak najmniej. W tym przypadku jednak model po lewej stronie ma wiele naruszeń marginesów, ale prawdopodobnie będzie cechował się lepszym uogólnianiem.



Rysunek 5.4. Mniejsza liczba naruszeń marginesu (po prawej) a większa szerokość marginesu (po lewej)



Jeżeli Twój model ulega przetrenowaniu, możesz go regularyzować za pomocą hiperparametru C.

Poniższy fragment kodu wczytuje zbiór danych Iris, skaluje cechy, a następnie trenuje liniowy model SVM (korzystamy z klasy LinearSVC, hiperparametru C=1 i **zawiasowej funkcji straty** — ang. *hinge loss* — do której niebawem powrócimy) do rozpoznawania kwiatów odmiany *Iris virginica*:

```
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)] # Długość płatka, szerokość płatka
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)
```

Otrzymany model jest widoczny po prawej stronie na rysunku 5.4.

Jak zwykle możemy użyć wyuczonego modelu do wyliczenia prognoz:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```



Klasyfikatory SVM, w przeciwieństwie do klasyfikatorów regresyjnych, nie generują prawdopodobieństw przynależności próbki do poszczególnych klas.

Zamiast korzystać z klasy `LinearSVC` możemy użyć klasy `SVC` z wyznaczonym jądrem liniowym. Podczas tworzenia modelu `SVC` możemy napisać `SVC(kernel="linear", C=1)`. Ewentualnie jesteśmy w stanie zaimplementować klasę `SGDClassifier` — `SGDClassifier(loss="hinge", alpha=1/(m*C))`. Wykorzystujemy w tym przypadku algorytm stochastycznego spadku wzduł gradientu (zob, rozdział 4.) do wytrenowania liniowego klasyfikatora SVM. Metoda ta osiąga zbieżność wolniej niż klasa `LinearSVC`, ale przydaje się do przetwarzania zadań klasyfikacji przyrostowej lub olbrzymich zbiorów danych, które nie mieszczą się w pamięci (uczenie pozakorowe).



Klasa `LinearSVC` regularyzuje punkt obciążenia, dlatego należy najpierw wyśrodkować zbiór uczący, odejmując od niego wartość średnią. Jeżeli skalujemy dane za pomocą klasy `StandardScaler`, proces ten jest przeprowadzany automatycznie. Wyznacza również wartość `hinge` w hiperparametrze `loss`, gdyż nie jest ona domyślna. Na koniec, uzyskasz lepszą wydajność, jeśli wstawisz wartość `False` do hiperparametru `dual`, chyba że w zestawie uczącym znajduje się więcej cech niż przykładów (w dalszej części rozdziału przyjrzymy się uważniej dualności).

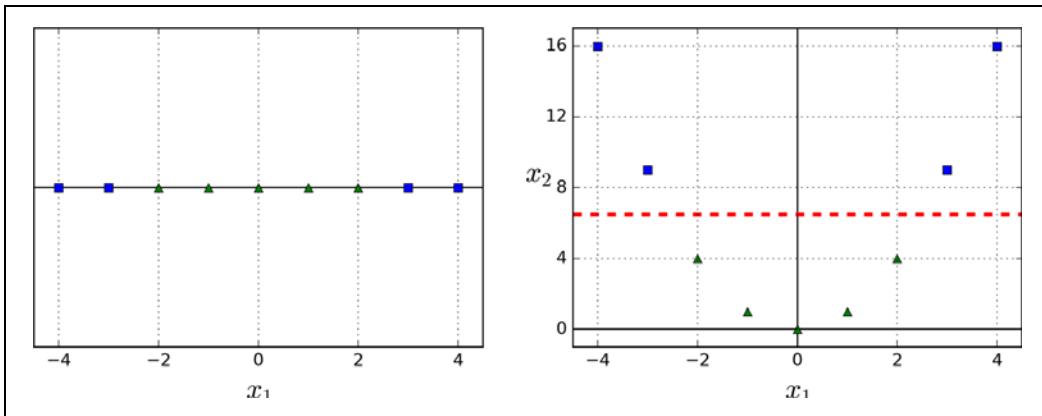
Nieliniowa klasyfikacja SVM

Liniowe klasyfikatory SVM są skuteczne i w wielu przypadkach sprawdzają się zdumiewająco dobrze, jednak wiele zbiorów danych nawet w części nie jest liniowo rozdzielnymi. Jednym ze sposobów na radzenie sobie z takimi nieliniowymi zbiorami danych jest dodawanie kolejnych (np. wielomianowych; zob. rozdział 4.); w niektórych przypadkach możemy uzyskać w ten sposób zestaw liniowo rozdzielnego danych. Spójrz na lewy wykres na rysunku 5.5: widzimy na nim prosty zbiór danych zawierający tylko jedną cechę x_1 . Łatwo zauważyc, że nie jest on liniowo rozdzielnny. Jeśli jednak dodamy drugą cechę, $x_2 = (x_1)^2$, wynikowy dwuwymiarowy wykres stanie się idealnie separowałny liniowo.

Aby zaimplementować tę koncepcję w module Scikit-Learn, stwórz potok (`Pipeline`) zawierający transformator `PolynomialFeatures` (omówiony w rozdziale 4., w podrozdziale „Regresja wielomianowa”) oraz kolejno klasy `StandardScaler` i `LinearSVC`. Przetestujmy to rozwiązanie na zestawie danych sierpowatych (`moons`): jest to testowy zestaw danych wykorzystywany w klasyfikacji binarnej, w którym punkty danych tworzą kształt dwóch nachodzących na siebie półksiężyćów (rysunek 5.6). Tworzymy ten zestaw danych za pomocą funkcji `make_moons()`:

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
```

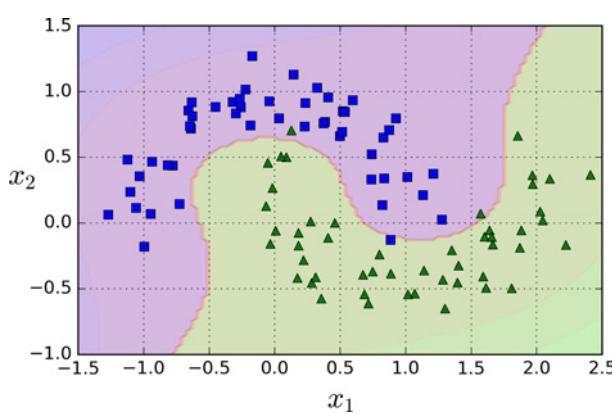


Rysunek 5.5. Dodawanie cech w celu uzyskania zbioru rozdzielnych liniowo danych

```

("svm_clf", LinearSVC(C=10, loss="hinge"))
])
polynomial_svm_clf.fit(X, y)

```



Rysunek 5.6. Liniowy klasyfikator SVM wykorzystujący cechy wielomianowe

Jądro wielomianowe

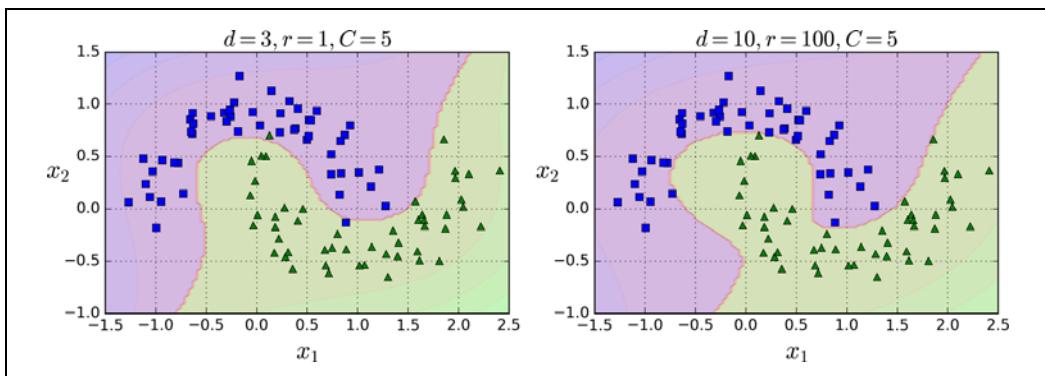
Dodawanie wielomianowych cech jest rozwiązaniem bardzo prostym do zaimplementowania i świetnie sprawującym się z różnego rodzaju algorytmami uczenia maszynowego (nie tylko z maszynami SVM). Pomimo tego metoda ta przy wielomianach niższego stopnia nie radzi sobie zbyt dobrze ze złożonymi zbiorami danych, natomiast w przypadku wielomianów dużego stopnia generuje olbrzymią liczbę cech, co znacznie spowalnia działanie modelu.

Na szczeźcie podczas stosowania maszyn SVM możemy wprowadzić niemal cudowną technikę matematyczną, zwaną **sztuczką z funkcją jądra** (ang. *kernel trick*; niebawem zostanie ona dokładniej wyjaśniona). Dzięki niej jesteśmy w stanie uzyskiwać takie same wyniki, jak po dodaniu wielu cech wielomianowych, nawet jeśli są nimi wielomiany dużego stopnia, bez potrzeby ich generowania.

Zatem nie musimy obawiać się gwałtownego wzrostu liczby cech, gdyż w rzeczywistości wcale ich nie dodajemy. Sztuczka ta jest zaimplementowana w klasie SVC. Sprawdźmy jej działanie na znany już nam zbiorze danych sierpowatych:

```
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)
```

Uczymy tu klasyfikator SVM za pomocą jądra wielomianowego trzeciego stopnia. Odzwierciedla to lewy wykres na rysunku 5.7. Na prawym wykresie widzimy z kolei inny klasyfikator SVM wykorzystujący jądro wielomianowe dziesiątego stopnia. Oczywiście jeśli model wykazuje cechy przetrenowania, powinniśmy zmniejszyć stopień wielomianu. Z kolei jeśli jest niedotrenowany, warto spróbować zwiększyć stopień wielomianu. Hiperparametr `coef0` reguluje proporcję wpływu wielomianów dużego stopnia do wielomianu małego stopnia na zachowanie modelu.



Rysunek 5.7. Klasyfikatory SVM wykorzystujące jądro wielomianowe



Powszechnie stosowanym sposobem wyszukiwania optymalnych wartości hiperparametrów jest metoda przeszukiwania siatki (zob. rozdział 2.). Często najszybciej jest przeprowadzić najpierw bardzo ogólne przeszukiwanie siatki, a następnie zwiększyć dokładność w okolicach najlepszych znalezionych wartości. Znajomość funkcji wykonywanej przez każdy hiperparametr również pozwala nam skoncentrować się na odpowiednim obszarze przestrzeni hiperparametrycznej.

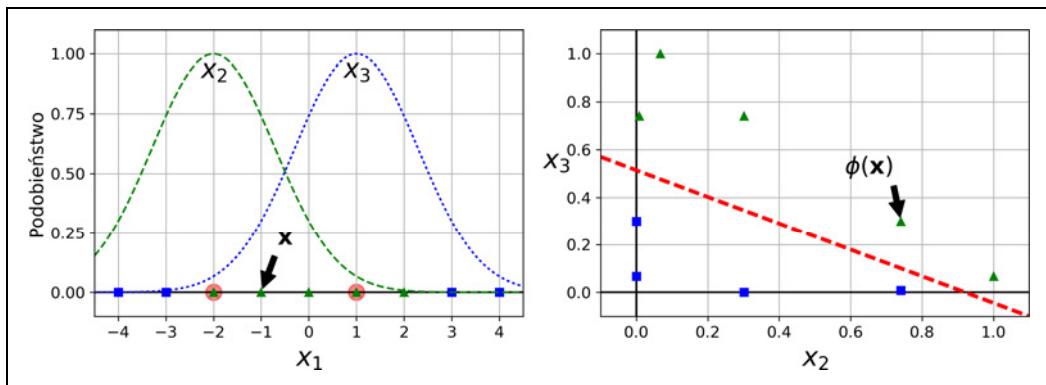
Cechy podobieństwa

Inną techniką pozwalającą na przetwarzanie problemów nielinowych jest dodawanie cech wyliczonych za pomocą **funkcji podobieństwa** (ang. *similarity function*), sprawdzającej, w jakim stopniu dany przykład przypomina określony **punkt charakterystyczny** (ang. *landmark*). Na przykład weźmy wspomniany już jednowymiarowy zestaw danych i dodajmy do niego dwa punkty charakterystyczne $x_1 = -2$ i $x_1 = 1$ (lewy wykres na rysunku 5.8). Następnie na funkcję podobieństwa wyznaczmy gaussowską **radialną funkcję bazową** (ang. *radial basis function* — RBF) o parametrze $\gamma = 0.3$ (zob. równanie 5.1).

Równanie 5.1. Gaussowska funkcja RBF

$$\phi_\gamma(\mathbf{x}, \ell) = \exp(-\gamma \|\mathbf{x} - \ell\|^2)$$

Wykres tej funkcji ma dzwonowy kształt i przyjmuje wartości od 0 (daleko od punktu charakterystycznego) do 1 (dla punktu charakterystycznego). Teraz możemy obliczyć nowe cechy. Przykładowo, przyjrzyjmy się próbce $x_1 = -1$: dzieli ją odległość 1 od pierwszego punktu charakterystycznego i 2 od drugiego punktu charakterystycznego. Z tego powodu jego nowymi cechami są $x_2 = \exp(-0,3 \times 1^2) \approx 0,74$ i $x_3 = \exp(-0,3 \times 2^2) \approx 0,30$. Prawy wykres na rysunku 5.8 przedstawia już przekształcony zbiór danych (pozbawiony oryginalnych cech). Jak widać, teraz jest liniowo rozdzielny.



Rysunek 5.8. Cechy podobieństwa uzyskane za pomocą gaussowskiej funkcji RBF

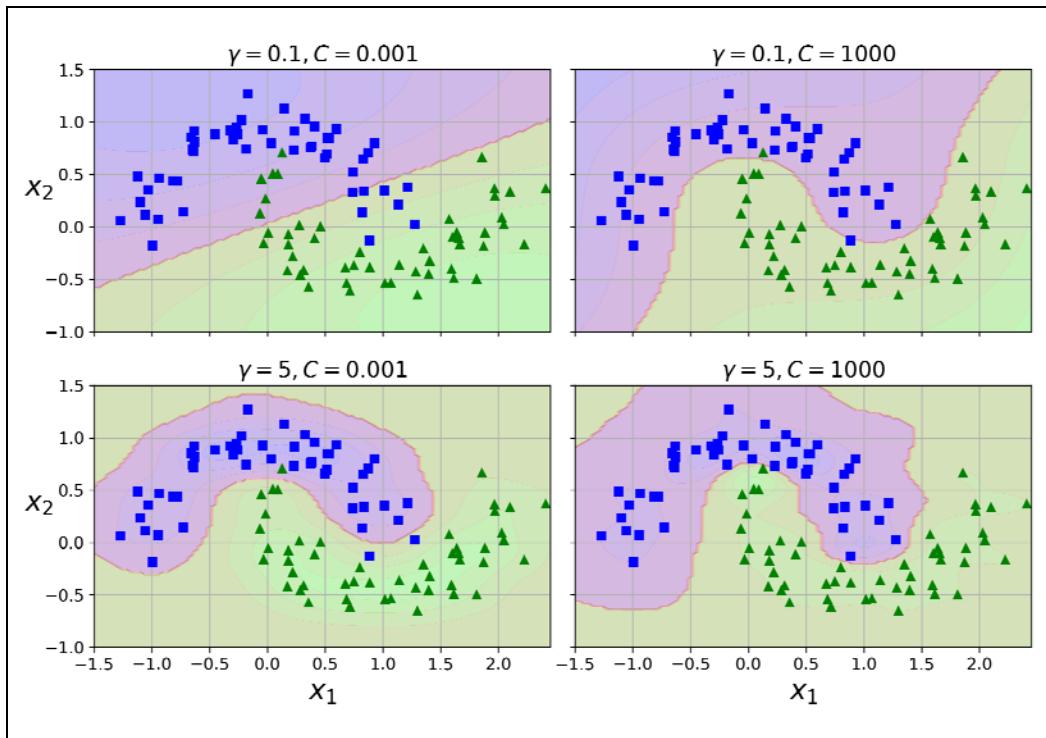
Zastanawiasz się pewnie, jak dobierać punkty charakterystyczne. Najprostszym rozwiązaniem jest ich utworzenie w lokacji każdego przykładu zbioru danych. W ten sposób uzyskujemy wiele wymiarów i zwiększamy szansę, że przekształcony zbiór danych będzie liniowo rozdzielny. Wadą tego rozwiązania jest fakt, że zbiór danych uczących zawierający m próbek i n cech zostaje przekształcony w zbiór uczący o m przykładach i m cechach (przy założeniu, że usuniemy pierwotne cechy). Jeśli korzystasz z bardzo dużego zestawu danych, wygenerujesz równie duży zbiór cech.

Gaussowskie jądro RBF

Podobnie jak w przypadku metody cech wielomianowych, technika cech podobieństwa okazuje się przydatna w każdym algorytmie uczenia maszynowego, ale wyliczenie wszystkich nowych cech może okazać się bardzo kosztowne obliczeniowo, zwłaszcza jeśli dysponujemy dużymi zbiorami danych. Również w tym przypadku sztuczka z jądrelem okazuje się niezastąpiona: uzyskujemy za jej pomocą podobne wyniki, jak po dodaniu wielu cech. Sprawdźmy, co uzyskamy, implementując gaussowskie jądro RBF za pomocą klasy 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)
```

Model ten jest ukazany na lewym dolnym wykresie na rysunku 5.9. Pozostałe wykresy uzyskaliśmy, ucząc modele zawierające różne wartości hiperparametrów gamma (γ) i C. Wraz ze wzrostem wartości hiperparametru gamma krzywa dzwonowa staje się węższa (prawy wykres na rysunku 5.8). W konsekwencji zmniejsza się zakres wpływów każdej próbki: wynikowa granica decyzyjna jest bardziej nieregularna i zagina się przy każdym przykładzie. Z kolei mniejsze wartości hiperparametru gamma powodują rozszerzanie krzywej dzwonowej, dlatego próbki mają większą strefę wpływów, a granica decyzyjna cechuje się gładszym przebiegiem. Zatem γ pełni rolę hiperparametru regularizacyjnego: jeżeli model ulega przetrenowaniu, należy zmniejszyć jego wartość, a jeśli jest niedotrenowany — zwiększyć (podobnie ma się rzecz w przypadku hiperparametru C).



Rysunek 5.9. Klasyfikatory SVM wykorzystujące jądro RBF



Przy takiej mnogości jąder jak należy wybrać właściwe? Zgodnie z niepisana zasadą należy zawsze wypróbować najpierw jądro liniowe (pamiętaj, że klasa `LinearSVC` jest znacznie szybsza od jądra `SVC(kernel="linear")`), zwłaszcza jeśli zbiór danych jest bardzo duży lub zawiera wiele cech. Przy mniejszych rozmiarach zbioru uczącego warto sprawdzić również gaussowskie jądro RBF; w większości przypadków sprawdza się całkiem nieźle. Jeśli masz nadmiar czasu i mocy obliczeniowej, możesz poeksperymentować z kilkoma innymi jądrami, stosując metody sprawdzianu krzyżowego i przeszukiwania siatki. Dotyczy to przede wszystkim jąder dostosowanych do używanych przez Ciebie struktur danych.

Istnieją również inne jądra, ale są stosowane znacznie rzadziej. Niektóre są dostosowane do określonych struktur danych. **Jądra łańcuchowe** (ang. *string kernels*) są czasami używane do klasyfikowania dokumentów tekstowych lub sekwencji DNA (np. przy użyciu **jądra podsekwencji łańcucha znaków** — ang. *string subsequence kernel* — lub jąder wykorzystujących **odległość Levenstheina**).

Złożoność obliczeniowa

Klasa LinearSVC bazuje na bibliotece LIBLINEAR, zawierającej zoptymalizowany algorytm (<https://www.csie.ntu.edu.tw/~cjlin/papers/cddual.pdf>) liniowych maszyn SVM¹. Nie obsługuje on sztuczki z jądrem, ale skaluje się niemal liniowo z liczbą przykładów uczących i liczbą cech. Jego złożoność czasu uczenia wynosi w przybliżeniu $O(m \times n)$.

Algorytm potrzebuje więcej czasu, jeśli wymagamy wyższej precyzyj. Określamy ją za pomocą hiperparametru ϵ (w module Scikit-Learn nosi on nazwę `tol`). Dla większości zadań klasyfikacji domyślna wartość tolerancji jest wystarczająca.

Z kolei klasa SVC wywodzi się z biblioteki LIBSVM, implementującej algorytm (<https://homl.info/13>) wykorzystujący sztuczkę z jądrem². Złożoność czasu uczenia mieści się tu zazwyczaj w przedziale pomiędzy $O(m^2 \times n)$ a $O(m^3 \times n)$. Niestety oznacza to, że algorytm przerazliwie zwalnia w przypadku dużej liczby próbek uczących (np. setek tysięcy). Algorytm ten nadaje się znakomicie do skomplikowanych, niewielkich bądź średnich zbiorów uczących. Skaluje się on nieźle z liczbą cech, zwłaszcza jeśli są **rzadkie** (ang. *sparse features*; tj. gdy każdy przykład zawiera kilka niezerowych cech). W takiej sytuacji algorytm skaluje się w przybliżeniu z uśrednioną liczbą niezerowych cech przypadających na każdą próbkę. W tabeli 5.1 porównuję klasyfikatory SVM dostępne w module Scikit-Learn.

Tabela 5.1. Porównanie maszyn wektorów nośnych w module Scikit-Learn

Klasa	Złożoność obliczeniowa	Uczenie pozakorowe	Wymagane skalowanie	Sztuczka z jądrem
LinearSVC	$O(m \times n)$	Nie	Tak	Nie
SGDClassifier	$O(m \times n)$	Tak	Tak	Nie
SVC	$O(m^2 \times n)$ do $O(m^3 \times n)$	Nie	Tak	Tak

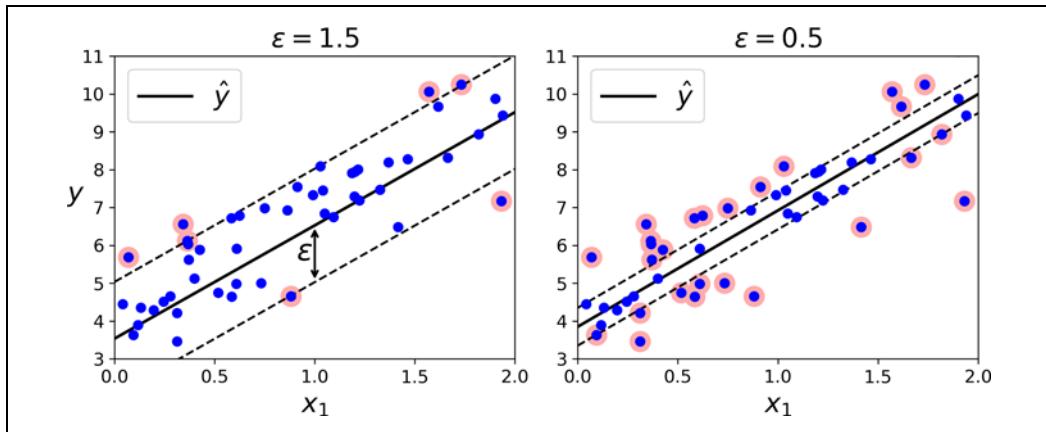
Regresja SVM

Jak już wspomniałem, algorytm SVM jest wszechstronny, ponieważ służy nie tylko do klasyfikacji liniowej i nieliniowej, ale obsługuje także regresję liniową/nieliniową. Tajemnica sukcesu wykorzystania maszyny SVM w zadaniach regresji, a nie klasyfikacji, tkwi w odwróceniu celu: w regresji SVM nie próbujemy uzyskać jak najszerszej „ulicy” pomiędzy dwiema klasami przy jednoczesnym ograniczeniu odległości do każdej z nich.

¹ Chih-Jen Lin i in., *A Dual Coordinate Descent Method for Large-Scale Linear SVM*, „Proceedings of the 25th International Conference on Machine Learning” (2008), s. 408 – 415.

² John Platt, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*, „Microsoft Research technical report”, April 21, 1998, <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf>.

niczeniu naruszeń marginesu, lecz zmieścić jak najwięcej przykładów wewnątrz „ulicy” przy jak najmniejszej liczbie naruszeń marginesu (próbkach wykraczających poza jego granice). Szerokość marginesu jest regulowana za pomocą hiperparametru ϵ . Na rysunku 5.10 widzimy dwa liniowe modele regresji SVM wyuczone za pomocą losowych danych linowych; jeden model ma szeroki margines ($\epsilon = 1,5$), natomiast drugi ma niewielką tolerancję ($\epsilon = 0,5$).



Rysunek 5.10. Regresja SVM

Dodawanie przykładów wewnątrz marginesu nie wpływa na wyliczane prognozy; dlatego mówimy, że model ten jest ϵ -nieczuły (ang. ϵ -insensitive).

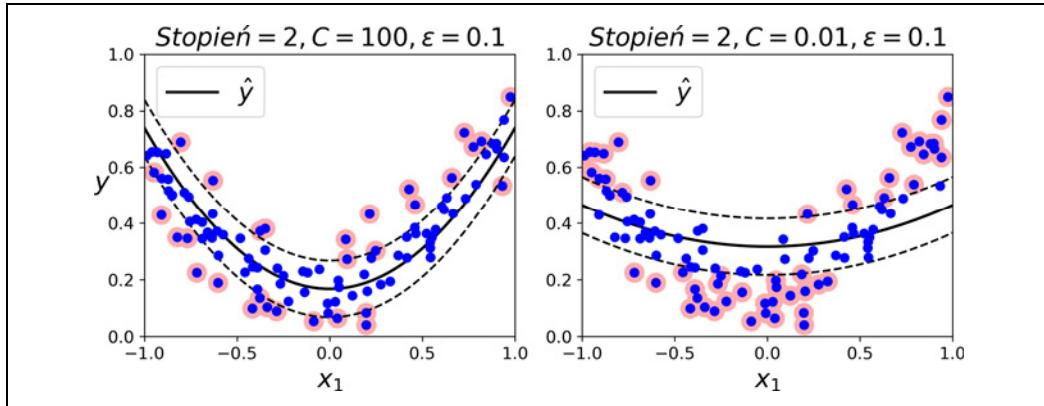
Mozemy wykorzystać klasę LinearSVR do przeprowadzenia regresji SVM. Dzięki poniższemu kodowi uzyskujemy model zaprezentowany na lewym wykresie (rysunek 5.10; dane uczące powinny zostać najpierw wyskalowane i wyśrodkowane):

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Aby przetwarzać zadania regresji nieliniowej, možemy skorzystać z kernelizowanego modelu SVM. Na rysunku 5.11 ukazano model regresji SVM wobec losowego zbioru danych wielomianowych, wykorzystujący wielomianowe jadro drugiego stopnia. Na lewym wykresie wprowadzamy niewielką regularyzację (tj. dużą wartość hiperparametru C), natomiast na prawym wykresie stopień regularizacji jest znacznie większy (mała wartość C).

Poniższy kod wykorzystuje klasę SVR (obsługującą sztuczkę z jądrem) do uzyskania modelu zaprezentowanego po lewej stronie na rysunku 5.11:

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



Rysunek 5.11. Regresja SVM przy użyciu wielomianowego jądra drugiego stopnia

Klasa SVR stanowi regresyjny równoważnik klasy SVC, podobnie jak klasa LinearSVR jest regresyjnym ekwiwalentem klasy LinearSVC. Klasa LinearSVR skaluje się liniowo z rozmiarem zbioru uczącego (podobnie jak klasa LinearSVC), podczas gdy klasa SVR/SVC znacznie spowalnia działanie wraz ze wzrostem rozmiarów zestawu danych uczących.



Możemy również używać maszyn SVM do wykrywania elementów odstających; szczegóły znajdziesz w dokumentacji modułu Scikit-Learn.

Mechanizm działania

W tym podrozdziale wyjaśnię mechanizm wyliczania prognoz przez maszyny SVM oraz zasadę działania ich algorytmów uczących, począwszy od liniowych klasyfikatorów SVM. Jeżeli dopiero zaczynasz przygodę z uczeniem maszynowym, możesz od razu przejść do podrozdziału z ćwiczeniami i wrócić tu, gdy już lepiej zapoznasz się z maszynami wektorów nośnych.

Jeszcze słowo wyjaśnienia na temat notacji. W rozdziale 4. korzystaliśmy z konwencji umieszczania wszystkich parametrów modelu w jednym wektorze θ , w tym punkt obciążenia θ_0 i początkowe wagę cech od θ_1 do θ_n , a następnie dodawania wejściowego obciążenia $x_0 = 1$ do wszystkich próbek. W tym rozdziale użyjemy wygodniejszej (i powszechniejszej) konwencji dotyczącej maszyn SVM: punkt obciążenia będzie symbolizowany literą b , natomiast wektor wag cech przyjmie postać litery w . Nie będziemy dodawać cechy obciążenia do wektora cech wejściowych.

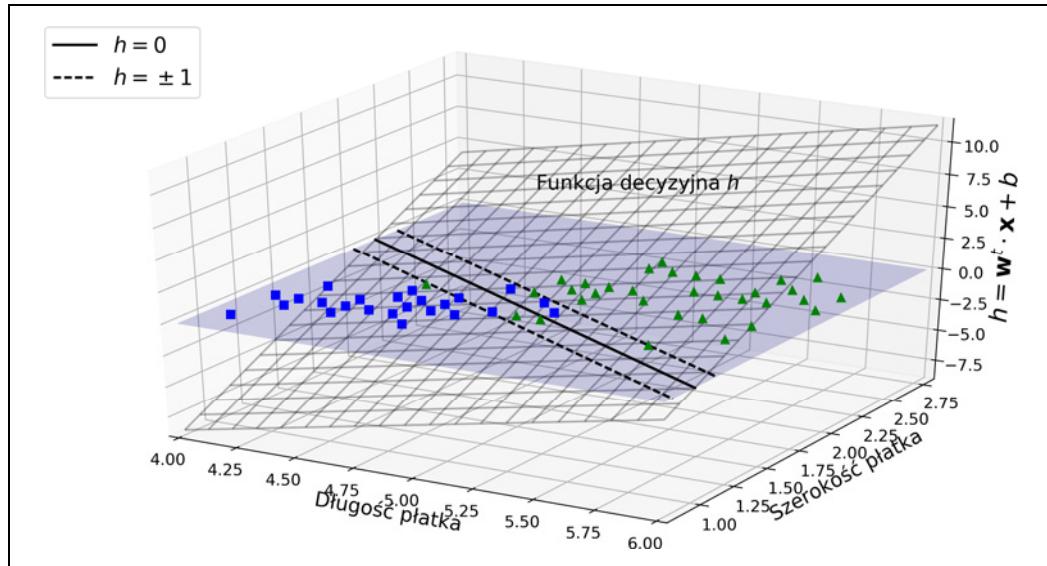
Funkcja decyzyjna i prognozy

Model liniowego klasyfikatora SVM przewiduje klasę nowego przykładu x poprzez obliczenie funkcji decyzyjnej $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + \dots + w_n x_n + b$. Jeżeli wynik jest dodatni, przewidywana klasa \hat{y} będzie pozytywna (1), w przeciwnym razie zostanie prognozowana klasa negatywna (0) (równanie 5.2).

Równanie 5.2. Prognoza liniowego klasyfikatora SVM

$$\hat{y} = \begin{cases} 0 & \text{jeśli } \mathbf{w}^T \mathbf{x} + b < 0 \\ 1 & \text{jeśli } \mathbf{w}^T \mathbf{x} + b \geq 0 \end{cases}$$

Rysunek 5.12 prezentuje funkcję decyzyjną odpowiadającą modelowi ukazanemu na lewym wykresie na rysunku 5.4: jest to płaszczyzna dwuwymiarowa, ponieważ zbiór danych zawiera dwie cechy (długość i szerokość płatka). Granicę decyzyjną stanowi zbiór punktów, dla których wartość funkcji decyzyjnej wynosi 0: mieści się ona na przecięciu dwóch płaszczyzn i ma postać linii prostej (na rysunku jest ona symbolizowana przez pogrubioną linię ciągłą)³.



Rysunek 5.12. Funkcja decyzyjna dla zbioru danych Iris

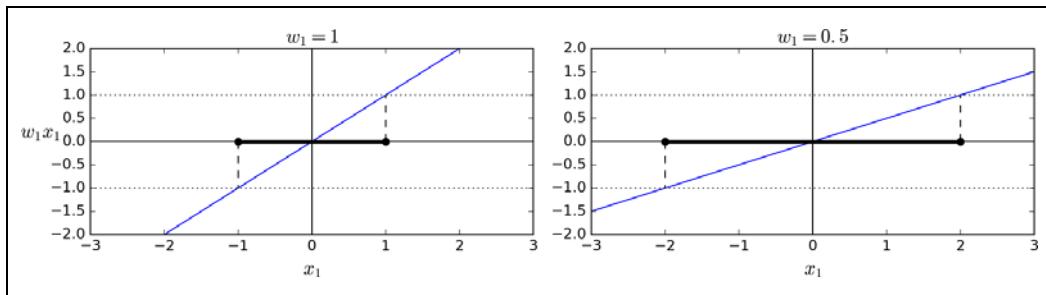
Linie przerywane ukazują punkty, w których funkcja decyzyjna przyjmuje wartości 1 lub -1: są one ułożone równolegle wobec granicy decyzyjnej i oddalone od niej o taką samą odległość — jest to nasze margines. Uczenie klasyfikatora SVM oznacza znalezienie wartości w i b pozwalających uzyskać jak najszerzy margines przy unikaniu jego naruszeń (twardy margines) lub ich ograniczeniu (miękkim margines).

Cel uczenia

Rozważmy nachylenie funkcji decyzyjnej: jest ona równa normie wektora wag — $\|w\|$. Jeżeli podzielimy wartości nachylenia przez 2, punkty, dla których wartości funkcji decyzyjnej będą równe ±1, będą znajdować się dwukrotnie dalej od granicy decyzyjnej. Innymi słowy, podzielenie nachylenia

³ W bardziej ogólnym ujęciu, jeżeli mamy n cech, funkcja decyzyjna przyjmuje postać n -wymiarowej **hiperplaszczyzny**, natomiast granica decyzyjna — hiperplaszczyzny ($n-1$)-wymiarowej.

przez 2 spowoduje dwukrotne rozszerzenie marginesu. Zjawisko to zostało zilustrowane na rysunku 5.13. Im mniejszy wektor wag w , tym uzyskujemy szerszy margines.



Rysunek 5.13. Zmniejszenie wartości wektora wag powoduje rozszerzenie marginesu

Chcemy zatem zminimalizować $\|w\|$, aby uzyskać duży margines. Jeśli pragniemy uniknąć jego naruszeń (twardy margines), musimy sprawić, żeby funkcja decyzyjna miała wartości większe od 1 dla wszystkich pozytywnych próbek uczących, a mniejsze od -1 dla przykładów negatywnych. Jeśli zdefiniujemy $t^{(i)} = -1$ dla próbek negatywnych ($y^{(i)} = 0$), a $t^{(i)} = 1$ dla przykładów pozytywnych ($y^{(i)} = 1$), to możemy wyrazić to ograniczenie jako $t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1$ dla wszystkich przykładów.

Możemy zatem odzwierciedlić cel liniowego klasyfikatora SVM wykorzystującego twardy margines jako problem **ograniczonej optymalizacji** (ang. *constrained optimization*) zaprezentowany w równaniu 5.3.

Równanie 5.3. Cel liniowego klasyfikatora SVM wykorzystującego twardy margines

$$\text{minimalizuj } \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

$$\text{pod warunkiem, że } t^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 \quad \text{dla } i = 1, 2, \dots, n$$



Nie minimalizujemy wyrażenia $\|\mathbf{w}\|$, lecz $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$, które jest równe $\frac{1}{2} \|\mathbf{w}\|^2$. Istotnie, wyrażenie $\frac{1}{2} \|\mathbf{w}\|^2$ ma całkiem prostą i elegancką pochodną (po prostu w), podczas gdy $\|\mathbf{w}\|$ nie jest różniczkowalne w punkcie $w = 0$. Algorytmy optymalizacyjne działają znacznie sprawniej wobec różniczkowalnych funkcji.

Aby uzyskać cel miękkiego marginesu, musimy dla każdej próbki wprowadzić **zmienną swobodną** (ang. *slack variable*) $\zeta^{(i)} \geq 0$:⁴ $\zeta^{(i)}$ definiuje stopień dopuszczalnego naruszenia marginesu przez i -ty przykład. Mamy teraz do czynienia z dwoma sprzecznymi celami: wyznaczeniem jak najmniejszych zmiennych swobodnych w celu ograniczenia naruszeń marginesu oraz zminimalizowaniem $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$ po to, aby rozszerzyć margines. Właśnie tutaj przydaje się hiperparametr C : za jego pomocą jesteśmy

⁴ Zeta (ζ) jest ósmą literą alfabetu greckiego.

w stanie wyznaczyć kompromis pomiędzy obydwooma celami. Teraz możemy zapisać problem ograniczonej optymalizacji (równanie 5.4).

Równanie 5.4. Cel liniowego klasyfikatora SVM wykorzystującego miękkie margines

$$\underset{\mathbf{w}, b, \zeta}{\text{minimalizuj}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)}$$

pod warunkiem, że $t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \zeta^{(i)} \geq 0 \quad \text{dla } i = 1, 2, \dots, m$

Programowanie kwadratowe

Problemy twardego i miękkiego marginesu należą do dziedziny problemów optymalizacji funkcji kwadratowych z ograniczeniami liniowymi. Problemy tego są znane jako problemy **programowania kwadratowego** (ang. *quadratic programming* — QP). Istnieje wiele mechanizmów ich rozwiązywania wykorzystujących techniki, których omówienie wykracza poza zakres niniejszej książki⁵. Ogólny problem został sformułowany w równaniu 5.5.

Równanie 5.5. Problem programowania kwadratowego

$$\underset{\mathbf{p}}{\text{minimalizuj}} \quad \frac{1}{2} \mathbf{p}^T \mathbf{H} \mathbf{p} + \mathbf{f}^T \mathbf{p}$$

pod warunkiem, że $\mathbf{A} \mathbf{p} \leq \mathbf{b}$

gdzie:

\mathbf{p} jest n_p -wymiarowym wektorem (n_p = liczba parametrów),

\mathbf{H} jest macierzą $n_p \times n_p$,

\mathbf{f} jest n_p -wymiarowym wektorem,

\mathbf{A} jest macierzą $n_c \times n_p$ (n_c = liczba ograniczeń),

\mathbf{b} jest n_c -wymiarowym wektorem.

Zwróć uwagę, że wyrażenie $\mathbf{A} \mathbf{p} \leq \mathbf{b}$ w rzeczywistości definiuje ograniczenia n_c : $\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}$ dla $i = 1, 2, \dots, n_c$, gdzie $\mathbf{a}^{(i)}$ stanowi wektor zawierający elementy i-tego rzędu macierzy \mathbf{A} , natomiast $b^{(i)}$ jest i-tym elementem wektora \mathbf{b} .

Możemy łatwo sprawdzić, że po wyznaczeniu parametrów QP w poniższy sposób otrzymamy cel liniowego klasyfikatora SVM wykorzystującego twardy margines:

⁵ Naukę programowania kwadratowego możesz rozpocząć od przeczytania książki *Convex Optimization* (https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf) autorstwa Stephena Boyda i Lievena Vandenberghe'a (Cambridge University Press, 2004) lub obejrzenia serii wykładów Richarda Browna (https://www.youtube.com/watch?v=elw1z1L_KJc&index=1&list=PLh464gFUoJW0mBYla3zbZbc4nv2AXez6X). Po polsku natomiast warto zapoznać się z pozycjami: *Analiza wypukła*, L. Kowalski i A. Piskorek (Wojskowa Akademia Techniczna) oraz *Wstęp do analizy wypukłej*, A. Piskorek (Wojskowa Akademia Techniczna).

- $n_p = n+1$, gdzie n stanowi liczbę cech (wartość +1 definiuje nasz punkt obciążenia),
- $n_c = m$, gdzie m stanowi liczbę próbek uczących,
- H stanowi macierz jednostkową $n_p \times n_p$, jedyne 0 znajduje się w pierwszej komórce (ignorujemy punkt obciążenia),
- $f = 0$ jest n_p -wymiarowym wektorem wypełnionym zerami,
- $b = -1$ jest n_c -wymiarowym wektorem wypełnionym ujemnymi jedynkami,
- $\mathbf{a}^{(i)} = -\mathbf{t}^{(i)}\dot{\mathbf{x}}^{(i)}$, gdzie $\dot{\mathbf{x}}^{(i)}$ jest równoważny wektorowi $\mathbf{x}^{(i)}$ zawierającemu dodatkową cechę obciążenia $\dot{\mathbf{x}}_0 = 1$.

Jednym ze sposobów trenowania liniowego klasyfikatora SVM wykorzystującego twardy margines jest zastosowanie dostępnego mechanizmu rozwiązywania QP i przekazanie mu powyższych parametrów. Wynikowy wektor \mathbf{p} będzie zawierał punkt obciążenia $b = p_0$ oraz wagi cech $w_i = p_i$ dla $i = 1, 2, \dots, n$. W analogiczny sposób możemy rozwiązywać problem miękkiego marginesu (zob. ćwiczenia na końcu rozdziału).

Żeby móc korzystać ze sztuczki z jądrem, musimy przeanalizować inny problem ograniczonej optymalizacji.

Problem dualny

Jeżeli znany jest problem ograniczonej optymalizacji, tzw. **problem pierwotny** (ang. *primal problem*), możliwe staje się wyrażenie odmiennego, ale blisko związanego zagadnienia, zwanego **problemem dualnym** (ang. *dual problem*). Rozwiązywanie problemu dualnego zazwyczaj stanowi ograniczenie dolne rozwiązania problemu pierwotnego, jednak w pewnych warunkach rozwiązanie obydwu problemów może być takie samo. Na szczęście maszyny SVM spełniają te warunki⁶, zatem możesz wybrać, czy chcesz rozwiązać problem pierwotny, czy dualny; w każdym przypadku uzyskasz to samo rozwiązanie. Równanie 5.6 przedstawia postać dualną celu liniowej maszyny SVM (jeżeli interesuje Cię wyprowadzenie formy dualnej z postaci pierwotnej, zajrzyj do dodatku C).

Równanie 5.6. Postać dualna celu liniowej maszyny SVM

$$\text{minimalizuj}_{\alpha} = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

pod warunkiem, że $\alpha^{(i)} \geq 0$ dla $i = 1, 2, \dots, m$

Po wyznaczeniu wektora $\hat{\alpha}$ minimalizującego wynik powyższego równania (za pomocą mechanizmu rozwiązywającego QP) wylicz za pomocą równania 5.7 wartości $\hat{\mathbf{w}}$ i \hat{b} minimalizujące problem pierwotny.

⁶ Funkcja celu jest wypukła, a ograniczenia nierówności są różniczkowalne w sposób ciągły i również stanowią funkcje wypukłe.

Równanie 5.7. Przejście od problemu dualnego do pierwotnego

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \right)$$

Jeżeli liczba przykładów uczących jest mniejsza od liczby cech, problem dualny jest rozwiązywany szybciej od problemu pierwotnego. Co więcej, w przeciwieństwie do problemu pierwotnego, problem dualny pozwala nam korzystać ze sztuczki z jądrem. Czym więc jest właściwie ta sztuczka?

Kernelizowane maszyny SVM

Załóżmy, że chcemy zastosować transformację wielomianową drugiego stopnia wobec dwuwymiarowego zbioru danych uczących (np. zbioru danych księzcówatych), a następnie wytrenować na nim liniowy klasyfikator SVM. Równanie 5.8 ukazuje wielomianową funkcję drugiego stopnia ϕ , którą chcemy wykorzystać.

Równanie 5.8. Odwzorowanie wielomianowe drugiego stopnia

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

Zwróc uwagę, że przekształcony wektor stał się trójwymiarowy. Zobaczmy teraz, co się stanie, jeśli przekształcimy w ten sposób dwa dwuwymiarowe wektory, \mathbf{a} i \mathbf{b} , a następnie wyliczymy ich iloczyn skalarny⁷ (równanie 5.9).

Równanie 5.9. Sztuczka z jądrem dla odwzorowania wielomianowego drugiego stopnia

$$\phi(\mathbf{a})^T \phi(\mathbf{b}) = \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 =$$
$$= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2$$

⁷ Jak już wyjaśniłem w rozdziale 4., iloczyn skalarny wektorów \mathbf{a} i \mathbf{b} zapisujemy zazwyczaj jako $\mathbf{a} \cdot \mathbf{b}$. Jednakże w uczeniu maszynowym wektory są często reprezentowane w postaci kolumnowej (tzn. jako macierze jednokolumnowe), dlatego iloczyn skalarny przyjmuje postać $\mathbf{a}^T \mathbf{b}$. Dla zachowania spójności z resztą książki będziemy korzystać tu ze wspomnianej notacji i przymknijmy oko na fakt, że z technicznego punktu widzenia otrzymujemy w ten sposób macierz jednoelementową, a nie wartość skalarną.

I jak? Iloczyn skalarny dwóch przekształconych wektorów jest równy kwadratowi iloczynu skalarnego dwóch pierwotnych wektorów: $\phi(\mathbf{a})^T \phi(\mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$.

Teraz nadszedł czas na ważne spostrzeżenie: jeżeli zastosujesz transformację ϕ wobec wszystkich próbek uczących, to problem dualny (równanie 5.6) będzie zawierał iloczyn skalarny $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. Jeżeli jednak funkcja ϕ stanowi transformację wielomianową drugiego stopnia (równanie 5.8), to możemy zastąpić iloczyn skalarny tych przekształconych wektorów wyrażeniem $(\mathbf{x}^{(i)T} \mathbf{x}^{(j)})^2$. Zatem nie musimy wcale przekształcać przykładów uczących: wystarczy w równaniu 5.6 zastąpić iloczyn skalarny jego kwadratem. Otrzymasz dokładnie taki sam wynik jak w przypadku mozolnego przekształcania danych uczących i uczenia modelu za pomocą liniowego algorytmu SVM, jednak sztuczka ta jest znacznie oszczędniejsza obliczeniowo.

Funkcja $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ jest nazywana **jądrem wielomianowym drugiego stopnia** (ang. *2nd-degree polynomial kernel*). W świecie uczenia maszynowego jądrem jest funkcja zdolna do obliczania iloczynu skalarnego $\phi(\mathbf{a})^T \phi(\mathbf{b})$ wyłącznie przy użyciu pierwotnych wektorów \mathbf{a} i \mathbf{b} , w najmniejszym nawet stopniu niewykorzystującą przekształcenia ϕ . Równanie 5.10 zawiera listę najczęściej stosowanych jąder.

Równanie 5.10. Najpopularniejsze jądra

Liniowe: $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$

Wielomianowe: $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$

Gaussowskie RBF: $K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$

Sigmoidalne: $K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$

Twierdzenie Mercera

Zgodnie z **twierdzeniem Mercera** jeżeli funkcja $K(\mathbf{a}, \mathbf{b})$ spełnia kilka matematycznych warunków, zwanych **warunkami Mercera** (tzn. funkcja K musi być ciągła i symetryczna tak, że $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$ itd.), to istnieje dla niej funkcja ϕ mapująca wektory a i b na inną przestrzeń (może mieć ona większą liczbę wymiarów) tak, że $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$. Zatem możemy korzystać z funkcji K jak z jądra, skoro wiemy, że istnieje dla niej przekształcenie ϕ , chociaż nawet nie musimy go znać. W przypadku gaussowskiego jądra RBF można udowodnić, że funkcja ϕ odwzorowuje każdą próbkę uczącą na przestrzeni o nieskończonej liczbie wymiarów, zatem bardzo dobrze, że nie musimy przeprowadzać takiego mapowania!

Zwróć uwagę, że pewne powszechnie stosowane jądra (np. sigmoidalne) nie spełniają wszystkich warunków Mercera, co nie przeszkadza im w większości przypadków w prawidłowym działaniu.

Pozostała nam jeszcze jedna sprawa do wyjaśnienia. Równanie 5.7 przedstawia sposób przejścia od rozwiązania dualnego do pierwotnego w przypadku liniowego klasyfikatora SVM. Jeśli jednak zastosujemy sztuczkę z jądrem, otrzymamy równania zawierające przekształcenie $\phi(\mathbf{x}^{(i)})$. Wektor $\hat{\mathbf{w}}$ musi zawierać taką liczbę wymiarów, jak funkcja $\phi(\mathbf{x}^{(i)})$, a może ona być olbrzymia albo wręcz nieskończona, zatem nie będziemy w stanie go wyliczyć. Jak jednak możemy obliczać prognozy bez wektora $\hat{\mathbf{w}}$? Dobra wiadomość jest taka, że możemy dołączyć wzór na wektor $\hat{\mathbf{w}}$ z równania 5.7 do funkcji decyzyjnej dla nowego przykładu $\mathbf{x}^{(n)}$, dzięki czemu otrzymamy równanie zawierające wyłącznie iloczyny skalarne wektorów wejściowych. W ten sposób możemy skorzystać ze sztuczki z jądrem (równanie 5.11).

Równanie 5.11. Wyliczanie prognoz za pomocą kernelizowanej maszyny SVM

$$\begin{aligned} h_{\hat{\mathbf{w}}, \hat{b}}\left(\phi\left(\mathbf{x}^{(n)}\right)\right) &= \hat{\mathbf{w}}^T \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \left(\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi\left(\mathbf{x}^{(i)}\right)\right)^T \phi\left(\mathbf{x}^{(n)}\right) + \hat{b} = \\ &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \left(\phi\left(\mathbf{x}^{(i)}\right)^T \phi\left(\mathbf{x}^{(n)}\right)\right) + \hat{b} = \\ &= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}\right) + \hat{b} \end{aligned}$$

Zauważ, że skoro $\alpha^{(i)} \neq 0$ wyłącznie dla wektorów nośnych, wyliczanie prognoz jest przeprowadzane za pomocą iloczynu skalarnego nowego wektora wejściowego $\mathbf{x}^{(n)}$ wraz z wyłącznie wektorami nośnymi, a nie wszystkimi próbками uczącymi. Oczywiście za pomocą tej samej sztuczki musimy obliczyć punkt obciążenia \hat{b} (równanie 5.12).

Równanie 5.12. Obliczanie punktu obciążenia za pomocą sztuczki z jądrem

$$\begin{aligned} \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \hat{\mathbf{w}}^T \phi\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi\left(\mathbf{x}^{(j)}\right)\right)^T \phi\left(\mathbf{x}^{(i)}\right)\right) = \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(t^{(i)} - \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K\left(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\right)\right) \end{aligned}$$

Nie masz co się dziwić, jeżeli zaczyna boleć Cię głowa: taki jest nieszczęśliwy skutek ubocznego sztuczki z jądrem.

Przyrostowe maszyny SVM

Zanim przejdziemy do następnego rozdziału, przyjrzymy się jeszcze pokróćce przyrostowym klasyfikatorom SVM (czyli takim, w których nauka odbywa się w czasie rzeczywistym, wraz z pojawianiem się nowych przykładów uczących).

W przypadku liniowych klasyfikatorów SVM jedną z metod implementacji przyrostowego klasyfikatora SVM jest użycie algorytmu gradientu prostego (np. za pomocą klasy `SGDClassifier`) w celu minimalizacji funkcji kosztu (równanie 5.13) — funkcja ta jest pochodną otrzymaną z problemu pierwotnego. Niestety algorytm gradientu prostego staje się zbieżny znacznie wolniej niż przy użyciu technik programowania kwadratowego.

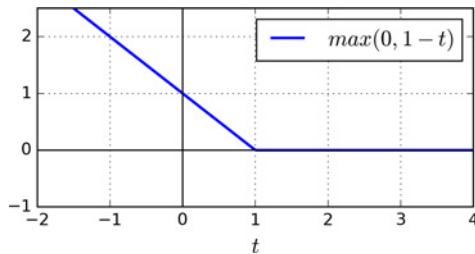
Równanie 5.13. Funkcja kosztu liniowego klasyfikatora SVM

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b))$$

Pierwsza suma w funkcji kosztu sprawia, że model będzie miał niewielki wektor wag w , co prowadzi do szerszego marginesu. Druga suma oblicza całkowitą wartość wszystkich naruszeń marginesu. Wartość naruszenia marginesu przez próbkę jest równa 0, jeżeli znajduje się poza nim i po właściwej stronie, w przeciwnym wypadku jest proporcjonalna do odległości od prawidłowej strony „ulicy”. Poprzez minimalizowanie tego członu staramy się, aby naruszenia marginesu były jak najmniejsze i najrzadsze.

Zawiasowa funkcja straty

Funkcja $\max(0, 1-t)$ jest nazywana **zawiasową funkcją straty** (została ukazana na poniższym wykresie). Jest równa 0, gdy $t \geq 1$. Jej pochodna (nachylenie) wynosi -1, jeśli $t < 1$, a 0, jeśli $t > 1$. Nie jest różniczkowalna w $t = 1$, ale podobnie jak w regresji metodą LASSO (zob. „Regresja metodą LASSO” w rozdziale 4.) jesteś w stanie korzystać z algorytmu gradientu prostego za pomocą dowolnej **subróżniczki** w $t = 1$ (tzn. dowolnej wartości pomiędzy -1 a 0).



Możliwa jest również implementacja innych przyrostowych, kernelizowanych maszyn SVM — co zostało opisane w publikacjach *Incremental and Decremental SVM Learning*⁸ (<https://papers.nips.cc/paper/1654-incremental-and-decremental-support-vector-machine-learning.pdf>).

⁸ Gert Cauwenberghs i Tomaso Poggio, *Incremental and Decremental Support Vector Machine Learning*, „Proceedings of the 13th International Conference on Neural Information Processing Systems” (2000), s. 388 – 394.

[cc/paper/1814-incremental-and-decremental-support-vector-machine-learning.pdf](http://www.csie.ntu.edu.tw/~cjlin/paper/1814-incremental-and-decremental-support-vector-machine-learning.pdf) i *Fast Kernel Classifiers with Online and Active Learning*⁹ (<http://www.jmlr.org/papers/volume6/bordes05a/bordes05a.pdf>). Wymienione modele są jednak zaimplementowane w środowiskach Matlab i C++. W przypadku wielkoskalowych problemów nieliniowych lepiej skorzystać z sieci neuronowych (część II).

Ćwiczenia

1. Jaka jest podstawowa koncepcja maszyn wektorów nośnych?
2. Czym jest wektor nośny?
3. Dlaczego należy skalować dane wejściowe podczas korzystania z maszyn SVM?
4. Czy klasyfikator SVM może obliczać wynik pewności podczas klasyfikowania przykładu? Czy może wyliczać prawdopodobieństwo przynależności próbki do danej klasy?
5. Z jakiego rozwiązania (pierwotnego czy dualnego) należy korzystać podczas uczenia modelu maszyny SVM wobec zestawu danych składającego się z milionów przykładów i setek cech?
6. Założymy, że wytrenowaliśmy klasyfikator SVM za pomocą jądra RBF, ale wydaje się, że jest niedotrenowany. Powinniśmy zwiększyć czy zmniejszyć wartość hiperparametru γ (gamma)? Co z hiperparametrem C ?
7. Jak należy skonfigurować parametry programowania kwadratowego (H, f, A i b) w celu użycia liniowego klasyfikatora SVM wykorzystującego miękkie margines za pomocą dostępnego mechanizmu QP?
8. Wyucz klasę LinearSVC wobec zbioru liniowo rozdzielnego danych. Następnie wytrenuj na nim również klasy SVC i SGDClassifier. Spróbuj uzyskać za ich pomocą podobny model.
9. Wyucz klasyfikator SVM wobec zbioru danych MNIST. Klasyfikatory SVM są binarne, zatem musisz zastosować strategię OvR, żeby były rozpoznawane wszystkie cyfry. Możesz chcieć dostroić hiperparametry za pomocą niewielkich zestawów walidacyjnych, aby przyśpieszyć cały proces. Jaką dokładność jesteś w stanie osiągnąć?
10. Wytrenuj regresor SVM wobec zbioru danych *California housing*.

Rozwiązań i odpowiedzi znajdziesz w dodatku A

⁹ Antoine Bordes i in., *Fast Kernel Classifiers with Online and Active Learning*, „Journal of Machine Learning Research” 6 (2005), s. 1579 – 1619.

Drzewa decyzyjne

Drzewa decyzyjne (ang. *decision trees*), podobnie jak maszyny wektorów nośnych, stanowią wszechstronne algorytmy uczenia maszynowego, służące zarówno do zadań klasyfikacji, jak i regresji, a nawet do operacji wielowyciśiowych. Uzyskujemy za ich pomocą potężne modele zdolne do uczenia się wobec złożonych zbiorów danych. Na przykład w rozdziale 2. wyuczylismy model `DecisionTreeRegressor` wobec zbioru danych California Housing i uzyskaliśmy doskonale wyniki (w rzeczywistości wręcz przetrenowaliśmy ten model).

Drzewa decyzyjne są również elementami składowymi losowych lasów (zob. rozdział 7.), czyli obecnie jednych z najlepszych algorytmów uczenia maszynowego.

W tym rozdziale zaczniemy od omówienia procesu uczenia drzew decyzyjnych, wyliczania prognoz i wizualizowania wyników. Następnie zajmiemy się algorytmem uczącym CART dostępnym w module Scikit-Learn, a także nauczymy się regularyzować drzewa i wykorzystywać je w zadaniach regresji. Na koniec przyjrzymy się niektórym ograniczeniom drzew decyzyjnych.

Uczenie i wizualizowanie drzewa decyzyjnego

Aby zrozumieć koncepcję drzew decyzyjnych, stworzmy jedno i zobaczymy, w jaki sposób wylicza prognozy. Za pomocą poniższego kodu wyuczylismy model `DecisionTreeClassifier` wobec zbioru danych Iris (zob. rozdział 4.):

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

iris = load_iris()
X = iris.data[:, 2:] # Długość i szerokość płatka
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Możemy zwizualizować wyuczone drzewo decyzyjne, używając najpierw metody `export_graphviz()`, aby stworzyć plik definicji grafu, nazwany `iris_drzewo.dot`:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
```

```

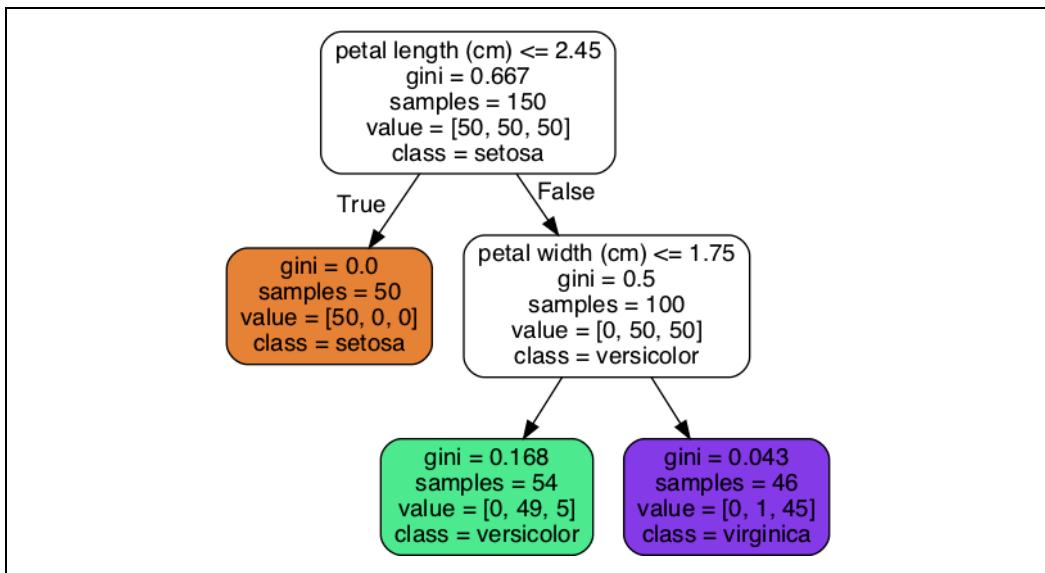
        out_file=image_path("iris_drzewo.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )

```

Możemy teraz przekonwertować wynikowy plik *.dot* na różne formaty (np. *.pdf* lub *.png*) za pomocą narzędzia wiersza poleceń dot, dostępnego w pakiecie *graphviz*¹. Poniższe polecenie przekształca plik *.dot* do formatu obrazu *.png*:

```
$ dot -Tpng iris_drzewo.dot -o iris_drzewo.png
```

Nasze pierwsze drzewo decyzyjne zostało pokazane na rysunku 6.1.



Rysunek 6.1. Drzewo decyzyjne wyuczone na zbiorze danych Iris

Wyliczanie prognoz

Zastanówmy się, w jaki sposób drzewo widoczne na rysunku 6.1 przewiduje wyniki. Założymy, że znaleźliśmy kwiat kosaćca i chcemy go sklasyfikować. Zaczynamy od **korzenia** (ang. *root node*; wysokość 0 — węzeł na samej górze grafu): zadajemy w tym węźle pytanie, czy długość płatka jest mniejsza niż 2,45 cm. Jeśli jest, przechodzimy do lewego węzła potomnego (wysokość 1, po lewej). W takim przypadku docieramy do **liścia** (ang. *leaf node*; nie wychodzą z niego kolejne węzły potomne), zatem nie zadajemy tu już więcej pytań. Wystarczy teraz spojrzeć na przewidywaną klasę w tym węźle — drzewo decyzyjne przewiduje, że mamy do czynienia z gatunkiem *Iris setosa* (*class=setosa*).

¹ Graphviz jest otwartym pakietem oprogramowania służącego do wizualizowania grafów, dostępnym pod adresem <http://www.graphviz.org/>.

Załóżmy teraz, że znajdujemy kolejny kwiat kosańca, którego długość płatka przekracza teraz 2,45 cm. Musimy teraz skierować się ku prawemu węzlowi potomnemu (wysokość 1, po prawej), który nie jest liściem, dlatego zostaje postawione kolejne pytanie: czy szerokość płatka jest mniejsza niż 1,75 cm? Jeśli tak, to prawdopodobnie znaleziony kwiat należy do gatunku *Iris versicolor* (wysokość 2, po lewej). W przeciwnym razie możemy mieć do czynienia z gatunkiem *Iris virginica* (wysokość 2, po prawej). To naprawdę jest takie proste.



Jedną z licznych zalet drzew decyzyjnych jest fakt, że prawie nie wymagają one przygotowywania danych. W istocie nie jest potrzebne skalowanie ani śródkowianie cech.

Atrybut `sample` węzła zlicza liczbę wyznaczonych do niego próbek uczących. Na przykład, 100 próbek ma długość płatka przekraczającą 2,45 cm (wysokość 1, po prawej), spośród których 54 mają szerokość płatka nieprzekraczającą 1,75 cm (wysokość 2, po lewej). Dzięki atrybutowi `value` dowiadujemy się, jak wiele przykładów uczących z każdej klasy przynależy do danego węzła: np. węzeł znajdujący się na dole po prawej stronie zawiera 0 próbek *Iris setosa*, 1 *Iris versicolor* i 45 *Iris virginica*. Z kolei atrybut `gini` stanowi miarę **zanieczyszczenia** (ang. *impurity*) węzła: węzeł jest „czysty” (`gini=0`), jeżeli wszystkie znajdujące się w nim próbki uczące należą do tej samej klasy. Przykładem jest węzeł na wysokości 1 po lewej stronie, ponieważ zawiera on tylko przykłady uczące *Iris setosa*; jego **wskaźnik Giniego** (ang. *Gini impurity*) jest równy 0. Równanie 6.1 pokazuje, w jaki sposób algorytm wylicza wskaźnik Giniego G_i dla i-tego węzła. Węzeł znajdujący się na wysokości 2 po lewej stronie ma wskaźnik Giniego o wartości $1 - (0.54)^2 - (49.54)^2 / (5.54)^2 \approx 0,168$.

Równanie 6.1. Wskaźnik Giniego

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

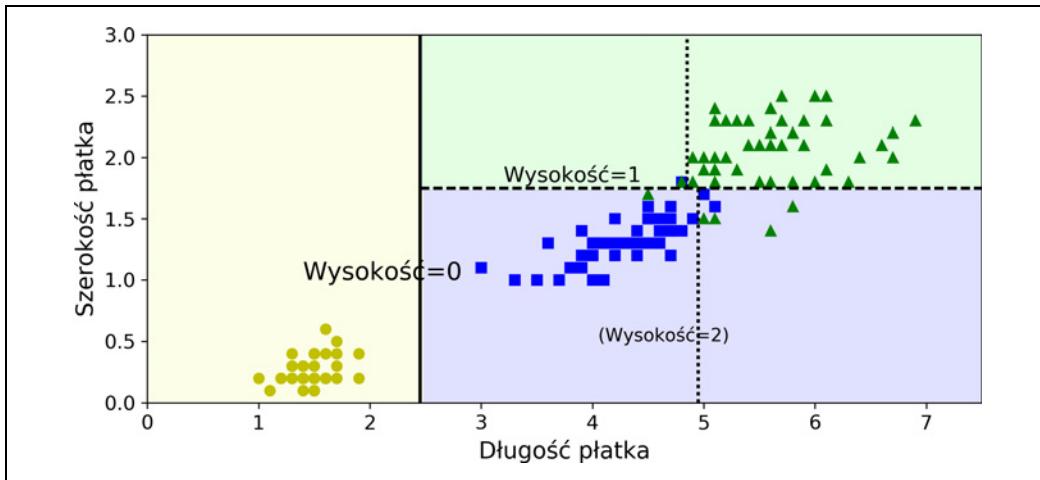
W tym równaniu:

- $p_{i,k}$ — współczynnik występowania klas k wśród próbek uczących w i -tym węźle.



Moduł Scikit-Learn wykorzystuje algorytm CART generujący wyłącznie **drzewa binarne** (ang. *binary trees*): węzły niebędące liśćmi zawsze mają dwoje potomków (tj. odpowiedziami na pytania są „tak” albo „nie”). Jednak inne algorytmy, takie jak ID3, mogą tworzyć drzewa decyzyjne, w których węzły mogą mieć większą liczbę potomków.

Na rysunku 6.2 widzimy granice decyzyjne drzewa decyzyjnego. Pogrubiona linia pionowa przedstawia granicę decyzyjną korzenia (wysokość 0): długość płatka = 2,45 cm. Obszar po lewej stronie jest czysty (występuje tu wyłącznie klasa *Iris setosa*), dlatego nie da się go bardziej podzielić. Jednakże obszar po prawej stronie pozostaje zanieczyszczony, zatem prawy węzeł na wysokości 1 rozdziela się na węzły potomne przy szerokości płatka = 1,75 cm (linia kreskowana). Wartość parametru `max_depth` wynosi 2, dlatego na takiej wysokości zatrzymuje się drzewo decyzyjne. Jeśli wyznaczymy wartość 3 parametru `max_depth`, to dwa węzły na wysokości 2 wprowadzilyby kolejną granicę decyzyjną (reprezentowaną przez linię kropkowaną).



Rysunek 6.2. Granice decyzyjne drzewa decyzyjnego

Interpretacja modelu: „czarne skrzynki” i „białe skrzynki”

Algorytm drzew decyzyjnych jest bardzo intuicyjny i możemy go z łatwością interpretować. Tego typu modele są często nazywane **modelami „białej skrzynki”** (ang. *white box models*). Jak się przekonamy, algorytmy losowego lasu lub sieci neuronowe należą do zgoła odmiennej kategorii, **modeli „czarnej skrzynki”** (ang. *black box models*). Ich przewidywania są znakomite i bez trudu możemy sprawdzić, jakie zostały przeprowadzone obliczenia do ich uzyskania; nie zmienia to faktu, że nierzaz ciężko wytłumaczyć prostymi słowami, skąd się te predykcje wzięły. Na przykład, jeśli sieć neuronowa stwierdzi, że na danym zdjęciu znajduje się taka to, a taka osoba, ciężko stwierdzić, jakie czynniki wpłynęły na ten rezultat: czy model rozpoznał daną osobę po oczach? Po ustach? Nosie? Butach? A może po kanapie, na której ta osoba siedzi? Z drugiej strony, drzewo decyzyjne zawiera szereg przejrzystych i dobrze zdefiniowanych reguł klasyfikacji, za pomocą których w razie potrzeby można nawet ręcznie wyliczać prognozy (np. w przypadku klasyfikowania kwiatów).

Szacowanie prawdopodobieństw przynależności do klas

Drzewo decyzyjne może również szacować prawdopodobieństwo przynależności danej próbki do określonej klasy k . Najpierw jest wyszukiwany liść, w którym dana próbka się znajduje, po czym zostaje zwrócony odsetek przykładów uczących w tym węźle należących do klasy k . Założymy, na przykład, że znaleźliśmy kwiat, którego płatki mają 5 cm długości i 1,5 cm szerokości. Próbka symbolizująca ten kwiat znajduje się w lewym liściu na wysokości 2 drzewa, zatem algorytm wyliczy następujące prawdopodobieństwa: 0% przynależności do gatunku *Iris setosa* (0/54), 90,7% dla *Iris versicolor* (49/54) i 9,3% dla *Iris virginica* (5/54). Zostanie dla tego kwiatu przewidziana klasa *Iris versicolor* (klasa 1.), ponieważ uzyskała ona największe prawdopodobieństwo. Sprawdzmy:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[0., 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Doskonale! Zwróć uwagę, że szacowane prawdopodobieństwa będą identyczne w dowolnym obszarze prawego dolnego prostokąta widocznym na rysunku 6.2 — np. dla płatków o długości 6 cm i szerokości 1,5 cm (nawet jeśli jest dla nas oczywiste, że w tym przypadku płatki te należałyby najprawdopodobniej do gatunku *Iris virginica*).

Algorytm uczący CART

Moduł Scikit-Learn wykorzystuje algorytm **drzew klasyfikacyjnych i regresyjnych** (ang. *classification and regression tree* — CART) do uczenia drzew decyzyjnych (ich „wzrostu”). Algorytm rozdziela najpierw dane uczące na dwa podzbiory przy użyciu pojedynczej cechy k i progu t_k (np., „długość płatka $\leq 2,45$ cm”). W jaki sposób są dobierane wartości k i t_k ? Wyszukiwana jest para parametrów (k, t_k) generująca najczystsze podzbiory (ważone pod względem rozmiaru). Funkcja kosztu, jaką algorytm stara się minimalizować, została przedstawiona w równaniu 6.2.

Równanie 6.2. Funkcja kosztu algorytmu CART używana w zadaniach klasyfikacji

$$J(k, t_k) = \frac{m_{\text{lewy}}}{m} G_{\text{lewy}} + \frac{m_{\text{prawy}}}{m} G_{\text{prawy}}$$

gdzie:

- $G_{\text{lewy/prawy}}$ — miara zanieczyszczenia lewego/prawego podzbioru,
- $m_{\text{lewy/prawy}}$ — liczba próbek w lewym/prawym podzbiorze.

Gdy algorytm CART rozdzieli zestaw danych uczących na dwa podzbiory, są one dalej dzielone na tej samej zasadzie aż do osiągnięcia maksymalnej wysokości (zdefiniowanej za pomocą hiperparametru `max_depth`) lub jeśli nie uda się określić takiego podziału, który zmniejszałby zanieczyszczenie. Kilka innych hiperparametrów (omówimy je niebawem) określa dodatkowe warunki zatrzymania budowy drzewa (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` i `max_leaf_nodes`).



Jak widać, CART jest **algorytmem zachłannym** (ang. *greedy algorithm*): zachłannie poszukuje optymalnego podziału na najniższym poziomie, a następnie na każdym kolejnym powtarza tę czynność. Nie sprawdza on, czy dany podział będzie prowadził kilka poziomów wyżej do najmniejszego możliwego zanieczyszczenia. Algorytmy zachłanne często dają dobre wyniki, nie muszą być one jednak optymalne.

Niestety szukanie optymalnego drzewa jest klasyfikowane jako **problem NP-zupełny**². Należy poświecić $O(\exp(m))$ czasu na jego rozwiązanie, przez co problem staje się trudny nawet dla małych zbiorów uczących. Dlatego musi nam wystarczyć poszukiwanie „w miarę dobrego” rozwiązania.

² P stanowi zbiór problemów, jakie mogą zostać rozwiązane w wielomianowym czasie, z kolei zbiór NP zawiera problemy, których rozwiązania mogą zostać zweryfikowane w wielomianowym czasie. Problem NP-trudny to każdy problem, do którego problem NP może zostać zredukowany w czasie wielomianowym. Do problemu NP-zupełnego zaliczają się problemy NP i NP-trudne. Jednym z głównych pytań natury matematycznej, niemających do tej pory odpowiedzi, pozostaje, czy $P = NP$. Jeżeli $P \neq NP$ (co jest bardzo prawdopodobne), to nigdy nie zostanie odkryty wielomianowy algorytm dla dowolnego problemu NP-zupełnego (może zmieni się to w przypadku komputerów kwantowych).

Złożoność obliczeniowa

Wyliczanie prognoz wymaga poruszania się po drzewie decyzyjnym od korzenia aż do liścia. Generalnie drzewa decyzyjne są w miarę zrównoważone, zatem poruszanie się po drzewie decyzyjnym wymaga jedynie odwiedzenia mniej więcej $O(\log_2(m))$ węzłów³. W każdym węźle wymagane jest jedynie sprawdzenie wartości jednej cechy, zatem całkowita złożoność prognoz to $O(\log_2(m))$, niezależnie od liczby cech. Wyliczanie przewidywań jest zatem bardzo szybkie nawet w przypadku bardzo dużych zbiorów uczących.

Algorytm uczący porównuje wszystkie cechy (mniej, jeśli wyznaczymy wartość parametru `max_features`) ze wszystkimi próbками znajdującymi się w danym węźle. Porównanie wszystkich cech we wszystkich przykładach w każdym węźle skutkuje złożonością uczenia na poziomie $O(n \times m \log_2(m))$. W przypadku niewielkich zbiorów uczących (zawierających do kilku tysięcy próbek) moduł Scikit-Learn może przyspieszyć proces uczenia za pomocą wstępnego sortowania danych (`presort=True`), jednak wyłączenie tej funkcji powoduje znaczne spowolnienie trenowania za pomocą dużych zestawów uczących.

Wskaźnik Giniego czy entropia?

Domyślnie używany jest wskaźnik Giniego, ale możemy wybrać również **entropię** jako miarę zanieczyszczenia, wprowadzając wartość `entropy` dla hiperparametru `criterion`. Pojęcie entropii wywodzi się z termodynamiki, gdzie służy do opisu miary nieuporządkowania cząsteczek: gdy cząsteczki są nieruchome i uporządkowane w przestrzeni, to wartość entropii wynosi 0. Koncepcja entropii wkradła się również w różne inne dziedziny naukowe, w tym również do **teorii informacji** Shannona, gdzie służy do pomiaru średniej zawartości informacji w wiadomości⁴: entropia wynosi 0, gdy wszystkie wiadomości są takie same. W świecie uczenia maszynowego za pomocą entropii często mierzy się zanieczyszczenie: entropia zbioru jest równa zeru, gdy mieszczą się w nim wyłącznie próbki należące do jednej klasy. Równanie 6.3 ukazuje definicję entropii i -tego węzła. Przykładowo

entropia lewego węzła na wysokości 2 (rysunek 6.1) wynosi $-\frac{49}{54} \log_2\left(\frac{49}{54}\right) - \frac{5}{54} \log_2\left(\frac{5}{54}\right) \approx 0,1$.

Równanie 6.3. Entropia

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log_2(p_{i,k})$$

Powinniśmy zatem korzystać ze wskaźnika Giniego czy z entropii? Prawdę mówiąc, w większości przypadków nie ma to większego znaczenia, gdyż uzyskujemy za ich pomocą podobne drzewa. Wskaźnik Giniego jest obliczany nieco szybciej, więc stanowi dobrą wartość domyślną. Jednak w sytu-

³ Zapis \log_2 oznacza logarytm binarny (dwójkowy). Jest on równy $\log_2(m) = \log(m)/\log(2)$.

⁴ Proces zmniejszania entropii często jest określany mianem **przyrostu informacji** (ang. *information gain* — IG).

acjach, w których obydwa wskaźniki się różnią, wskaźnikowi Giniego zdarza się izolować najczęściej występującą klasę w osobnej gałęzi, natomiast entropia generuje nieco bardziej zrównoważone drzewa⁵.

Hiperparametry regularyzacyjne

Algorytm drzew decyzyjnych nie przyjmuje niemal żadnych założeń dotyczących danych uczących (w przeciwieństwie na przykład do modeli liniowych, które zakładają, że operują na danych liniowych). Jeżeli nie nałożymy żadnych ograniczeń, struktura drzewa samoistnie dostosuje się do danych uczących i zrobi to prawie idealnie, niemal z pewnością ulegając przetrenowaniu. Jest to tak zwany **model nieparametryczny** (ang. *nonparametric model*) — nie dlatego, że nie zawiera parametrów (często ma ich znaczną liczbę), lecz dlatego, że liczba tych parametrów nie jest ustalana przed rozpoczęciem trenowania, zatem struktura modelu jest w stanie ściśle dopasować się do danych. Z drugiej strony, mamy do czynienia z **modelami parametrycznymi** (ang. *parametric models*; reprezentowanymi m.in. przez model liniowy), w których występuje ustalona liczba parametrów, zatem cechuje je ograniczenie stopni swobody, dzięki czemu zmniejszamy ryzyko przetrenowania (ale jednocześnie zwiększamy możliwość niedotrenowania).

Aby uniknąć przetrenowania modelu, musimy ograniczyć swobodę algorytmu drzewa decyzyjnego podczas uczenia. Wiemy już, że ten proces nosi nazwę regularizacji. Hiperparametry regularyzacyjne zależą od stosowanego algorytmu, zazwyczaj jednak możemy ograniczyć przynajmniej maksymalną wysokość drzewa. W module Scikit-Learn odpowiada za to hiperparametr `max_depth` (jego wartość domyślna, `None`, powoduje tworzenie drzew o nieograniczonej wysokości). Podanie wartości liczbowej w hiperparametrze `max_depth` spowoduje regularyzację modelu i zmniejszenie ryzyka przetrenowania.

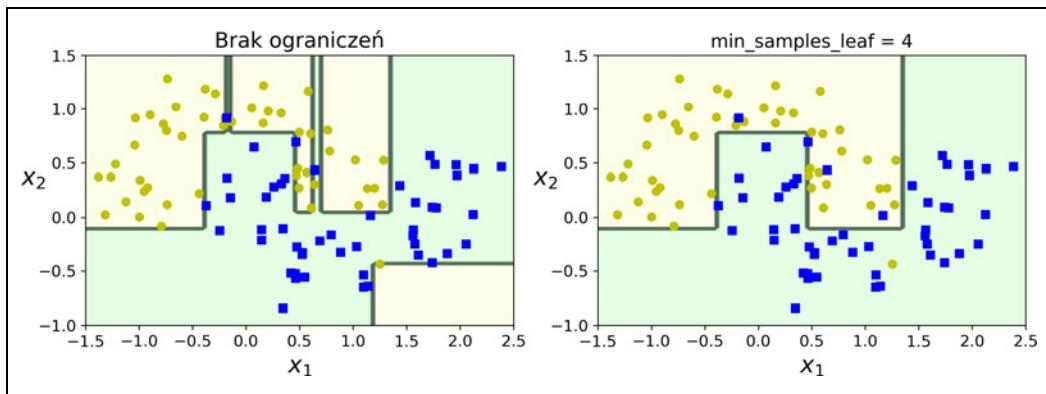
Klasa `DecisionTreeClassifier` zawiera także kilka innych parametrów ograniczających kształt drzewa decyzyjnego: `min_samples_split` (minimalna liczba próbek, jakie muszą znajdować się w węźle, zanim zostanie podzielony), `min_samples_leaf` (minimalna liczba próbek, jakie muszą znajdować się w liściu), `min_weight_fraction_leaf` (podobny do parametru `min_samples_leaf`, tu jednak wartością jest ułamek całkowitej liczby ważonych próbek), `max_leaf_nodes` (maksymalna liczba liści) oraz `max_features` (maksymalna liczba cech używanych do dzielenia poszczególnych węzłów). Zwiększenie wartości hiperparametrów `min_*` lub zmniejszanie `max_*` powoduje regularyzację modelu.



Inne algorytmy najpierw trenują model drzewa decyzyjnego bez żadnych ograniczeń, a następnie **przycinają** (ang. *prune*; usuwają) niepotrzebne liście. Węzeł zawierający same liście jest uznawany za niepotrzebny, jeśli zapewniana przez niego redukcja zanieczyszczenia okazuje się **nieistotna statystycznie**. Standardowe testy statystyczne, takie jak test chi kwadrat (test χ^2), służą do oszacowania prawdopodobieństwa, że zmniejszenie zanieczyszczenia stanowi wyłącznie wynik przypadku (jest to tzw. **hipoteza zerowa**). Jeżeli to prawdopodobieństwo, zwane **p-wartością**, przekroczy pewien określony próg (zazwyczaj 5% — definiujemy go za pomocą hiperparametru), to węzeł jest uznawany za niepotrzebny, a jego liście zostają usunięte. Proces przycinania trwa, dopóki nie zostaną usunięte wszystkie niepotrzebne węzły.

⁵ Więcej szczegółów znajdziesz w interesującej analizie (<https://sebastianraschka.com/faq/docs/decision-tree-binary.html>), której autorem jest Sebastian Raschka.

Rysunek 6.3 przedstawia dwa drzewa decyzyjne wyuczone wobec zestawu danych sierpowatych (mieliśmy z nimi do czynienia w rozdziale 5.). Drzewo po lewej stronie zostało wytrenowane za pomocą standardowych hiperparametrów (tj. bez ograniczeń), natomiast w drzewie po prawej wyznaczyliśmy hiperparametr `min_samples_leaf=4`. Wyróżnia się, że lewy model jest przetrenowany, natomiast prawy będzie prawdopodobnie lepiej przeprowadzał uogólnianie.



Rysunek 6.3. Regularyzacja za pomocą hiperparametru `min_samples_leaf`

Regresja

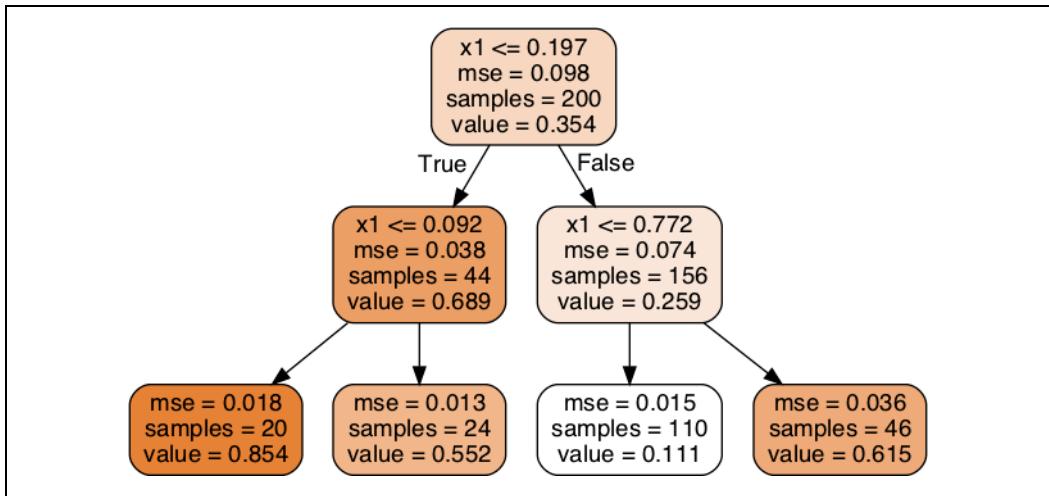
Drzewa decyzyjne mogą również wykonywać zadania regresyjne. Stworzymy takie drzewo regresyjne za pomocą klasy `DecisionTreeRegressor`; wyuczmy je wobec zaszumionego, kwadratowego zbioru danych, a jego maksymalną wysokość ustalmy na poziomie `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor(max_depth=2)  
tree_reg.fit(X, y)
```

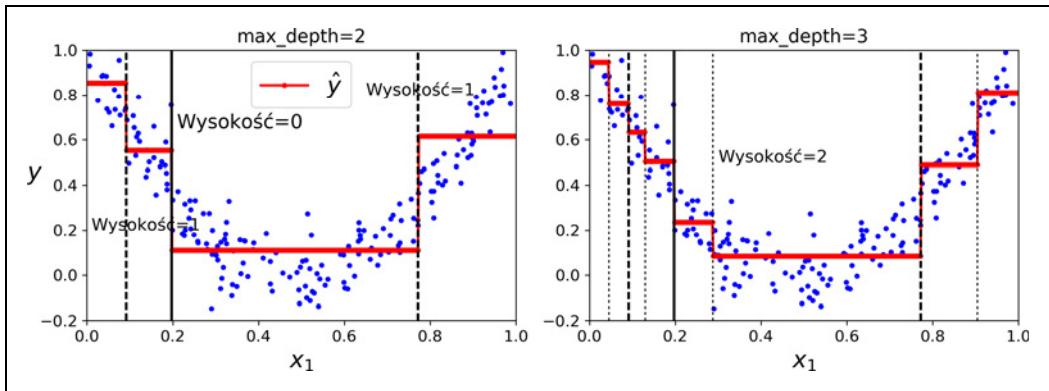
Rysunek 6.4 przedstawia wygenerowane drzewo decyzyjne.

Model ten bardzo przypomina stworzone przez nas wcześniej drzewo klasyfikacyjne. Główna różnica polega na tym, że w każdym węźle jest prognozowana nie klasa, lecz wartość. Założymy przykładowo, że chcemy wyliczyć prognozę dla nowej próbki $x_1 = 0.6$. Poruszamy się po drzewie, poczawszy od korzenia, i w końcu docieramy do liścia przewidującego `value=0.111`. Predykcja ta stanowi średnią wartość docelową 110 próbek uczących powiązanych z tym liściem, a w wyniku tej prognozy otrzymujemy wartość błędu MSE wynoszącą 0,015 dla tych 110 próbek.

Przewidywania tego modelu zostały zaprezentowane po lewej stronie na rysunku 6.5. Jeżeli wyznaczmy hiperparametr `max_depth=3`, prognozy będą wyglądały tak, jak na prawym wykresie. Zwrót uwagę, że prognozowana wartość dla każdego obszaru stanowi zawsze średnią wartość docelową próbek znajdujących się na tym obszarze. Algorytm rozdziela każdy obszar w taki sposób, żeby jak najwięcej próbek uczących znajdowało się możliwie blisko tej przewidywanej wartości.



Rysunek 6.4. Regresyjne drzewo decyzyjne



Rysunek 6.5. Prognozy dwóch modeli regresyjnych drzew decyzyjnych

Algorytm CART działa niemal tak samo, jak w czasie klasyfikacji, teraz jednak stara się rozdzielać zbiór danych uczących w sposób minimalizujący błąd MSE (a nie zanieczyszczenie). Równanie 6.4 zawiera funkcję kosztu minimalizowaną przez ten algorytm.

Równanie 6.4. Funkcja kosztu CART stosowana w regresji

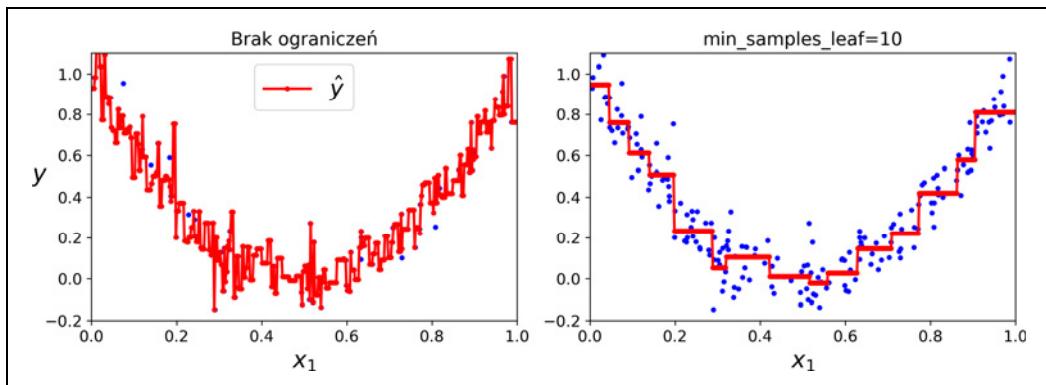
$$J(k, t_k) = \frac{m_{\text{lewy}}}{m} MSE_{\text{lewy}} + \frac{m_{\text{prawy}}}{m} MSE_{\text{prawy}}$$

gdzie:

$$MSE_{\text{w\u0142e\u0144}} = \sum_{i \in \text{w\u0142e\u0144}} \left(\hat{y}_{\text{w\u0142e\u0144}} - y^{(i)} \right)^2$$

$$\hat{y}_{\text{w\u0142e\u0144}} = \frac{1}{m_{\text{w\u0142e\u0144}}} \sum_{i \in \text{w\u0142e\u0144}} y^{(i)}$$

Podobnie jak w przypadku zadań klasyfikacji, drzewa decyzyjne są podatne na przetrenowanie w modelach regresji. Jeśli nie będziemy stosować żadnej formy regularizacji (np. korzystając wyłącznie z domyślnych wartości hiperparametrów), uzyskamy prognozy widoczne na lewym wykresie rysunku 6.6. Jak widać, model jest znacznie przetrenowany. Wystarczy jednak, że wyznaczymy hiperparametr `min_samples_leaf=10`, aby uzyskać znacznie lepszy model, widoczny na prawym wykresie.



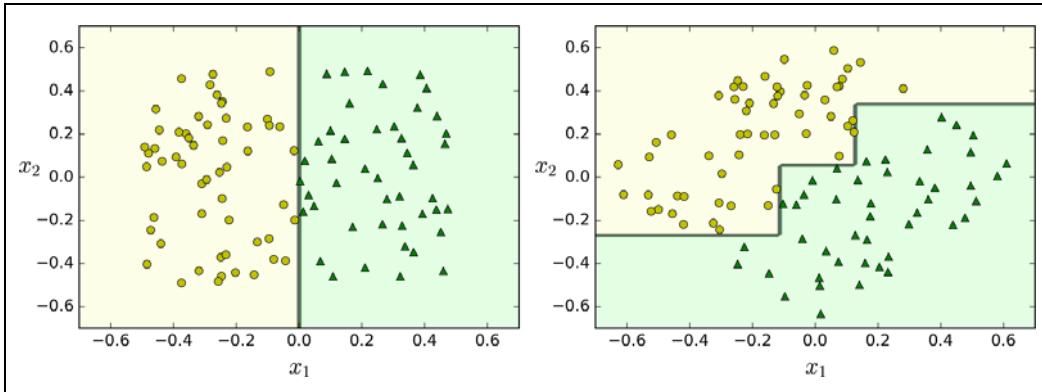
Rysunek 6.6. Regularizacja regresyjnego drzewa decyzyjnego

Niestabilność

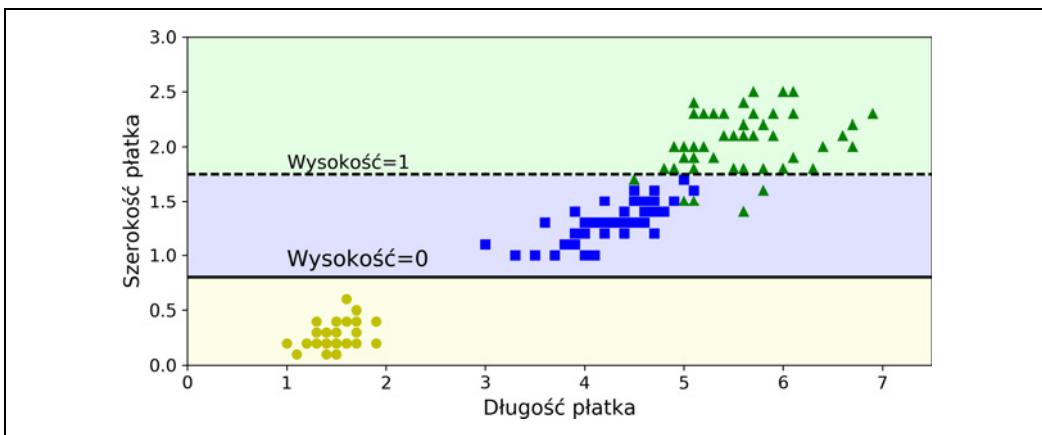
Mam nadzieję, że dostrzegasz już olbrzymi potencjał drzew decyzyjnych: są zrozumiałe i łatwe do interpretacji, przystępne, wszechstronne i wydajne. Mają jednak kilka ograniczeń. Po pierwsze, jak pewnie zdążyła/zdążyłeś zauważyc, algorytm drzewa decyzyjnego uwielbia ortogonalne granice decyzyjne (wszystkie podziały są przeprowadzane prostopadle do osi odciętych), przez co model ten jest wrażliwy na rotację zbioru uczącego. Na przykład rysunek 6.7 przedstawia prosty, liniowo rozdzielony zestaw danych: na lewym wykresie próbki są z łatwością rozdzielane, natomiast na prawym wykresie te same dane zostały obrócone o 45° i granica decyzyjna jest tu niepotrzebnie skomplikowana. Pomimo tego, że drzewo decyzyjne zostało perfekcyjnie dopasowane do danych, istnieje duże ryzyko, że model ten nie będzie zbyt dobrze przeprowadzał uogólniania. Jednym ze sposobów ograniczenia tego problemu jest wprowadzenie algorytmu analizy głównych składowych (PCA; zob. rozdział 8.), dzięki któremu model staje się lepiej zorientowany wobec danych uczących.

W bardziej ogólnym ujęciu główny problem drzew decyzyjnych polega na ich olbrzymiej czułości na niewielką wariancję w zestawie danych uczących. Na przykład, jeśli usuniesz tylko najszerszy płatek kosaćca *Iris versicolor* ze zbioru danych *Iris* (jego płatki mają długość 4,8 cm i szerokość 1,8 cm) i wyuczysz nowe drzewo decyzyjne, możesz uzyskać model zaprezentowany na rysunku 6.8. Jak widać, różni się on znacznie od wygenerowanego wcześniej drzewa (rysunek 6.2). W rzeczywistości algorytm uczący używany przez moduł Scikit-Learn jest stochastyczny⁶, zatem możesz uzyskiwać zupełnie odmienne modele wobec tego samego zestawu danych uczących (chyba że używasz hiperparametru `random_state`).

⁶ Losowo dobiera zestaw cech do oceniania każdego węzła.



Rysunek 6.7. Wrażliwość na rotację zbioru uczącego



Rysunek 6.8. Wrażliwość na niuanse zbioru danych uczących

Jak się przekonamy w następnym rozdziale, losowe lasy mogą ograniczać tę niestabilność poprzez uśrednianie prognoz uzyskiwanych z wielu drzew.

Ćwiczenia

1. Jaką przybliżoną wysokość osiągnie drzewo decyzyjne uczone (bez ograniczeń) wobec zestawu danych składającego się z miliona próbek?
2. Czy wskaźnik Giniego węzła podrzędnego jest zazwyczaj większy lub mniejszy od tego wskaźnika w węźle nadrzędnym? Czy jest on *zazwyczaj*, czy też *zawsze* mniejszy/większy?
3. Czy jeżeli drzewo decyzyjne ulega przetrenowaniu, warto próbować zmniejszyć wartość hiperparametru `max_depth`?
4. Czy dobrym pomysłem jest próba skalowania cech wejściowych, jeżeli drzewo decyzyjne wykazuje oznaki niedotrenowania?

5. Jeżeli wyuczenie drzewa decyzyjnego wobec zbioru danych składającego się z miliona próbek zajmuje godzinę, w przybliżeniu jak wiele czasu należy poświęcić na wytrenowanie drzewa przy użyciu zestawu składającego się z 10 milionów przykładów?
6. Jeżeli zbiór danych zawiera 100 000 próbek, to czy wyznaczenie parametru `presort=True` przyspieszy proces uczenia?
7. Wytrenuj i dostrój model drzewa decyzyjnego wobec danych sierpowatych korzystając z następujących kroków:
 - a. Stwórz zbiór danych za pomocą funkcji `make_moons(n_samples=10000, noise=0.4)`.
 - b. Rozdziel uzyskany zestaw danych na podzbiory uczący i testowy przy użyciu metody `train_test_split()`.
 - c. Wykorzystaj przeszukiwanie siatki wraz ze sprawdzianem krzyżowym (przyda się klasa `GridSearchCV`), aby znaleźć dobre wartości hiperparametrów dla klasy `DecisionTreeClassifier`. Podpowiedź: wypróbuj różne wartości hiperparametru `max_leaf_nodes`.
 - d. Wytrenuj ten model wobec pełnego zbioru uczącego, korzystając z uzyskanych wartości hiperparametrów, a następnie sprawdź wydajność modelu wobec zestawu testowego. Powinnaś/powinieneś uzyskać wyniki rzędu 85 – 87%.
8. Posadź las za pomocą następujących kroków:
 - a. Korzystając z poprzedniego ćwiczenia, wygeneruj 1000 podzbiorów zestawu uczącego, każdy zawierający 100 losowo dobranych próbek. Podpowiedź: możesz w tym celu skorzystać z klasy `ShuffleSplit`.
 - b. Wytrenuj po jednym drzewie decyzyjnym dla każdego podzbioru, korzystając z najlepszych wartości hiperparametrów odkrytych w poprzednim ćwiczeniu. Oceń wydajność tego tysiąca drzew decyzyjnych na zestawie testowym. Drzewa te zostały wyuczone przy użyciu mniejszych zbiorów danych, dlatego prawdopodobnie będą miały gorszą dokładność od pierwotnego drzewa decyzyjnego, oscylującą w granicach 80%.
 - c. Czas na odrobinę magii. Dla każdej próbki zbioru testowego wygeneruj prognozy wyliczane przez wszystkie 1000 drzew i zachowaj jedynie najczęściej powtarzający się wynik (możesz użyć do tego metody `mode()`, stanowiącej część modułu SciPy). Uzyskujesz w ten sposób **prognozy metodą głosowania większościowego** dla zbioru testowego.
 - d. Oceń te przewidywania wobec zbioru testowego: powinnaś/powinieneś uzyskać nieco większą dokładność niż w przypadku pierwotnego modelu (wyższą o 0,5% – 1,5%). Gratulacje, właśnie wytrenowałaś/wytrenowałeś swój pierwszy klasyfikator losowego lasu!

Rozwiązań tych ćwiczeń znajdziesz w dodatku A.

Uczenie zespołowe i losowe lasy

Założymy, że zadajesz skomplikowane pytanie tysiącom losowo dobranych osób, a następnie łączysz uzyskane odpowiedzi. W wielu przypadkach okazuje się, że taka zbiorcza odpowiedź jest lepsza od opinii eksperta. Jest to tak zwana **mądrość tłumu**. Podobne zjawisko następuje, gdy połączymy przewidywania grupy predyktorów (np. klasyfikatorów albo regresorów) — często otrzymamy w ten sposób lepsze prognozy niż uzyskane przez najlepszy pojedynczy model. Grupę predyktorów nazywamy **zespołem** (ang. *ensemble*), zatem omawiana technika nosi miano **uczenia zespołowego** (ang. *ensemble learning*), a używany w niej algorytm — **metody uczenia zespołowego** (ang. *ensemble learning method*).

W ramach przykładowej metody uczenia zespołowego możesz wytrenować grupę klasyfikatorów drzew decyzyjnych, każdy wobec odrębnego podzbioru zbioru uczącego. Aby uzyskać prognozy, wystarczy je zebrać z pojedynczych drzew, a następnie wybrać klasę, która otrzymała najwięcej głosów (zob. ostatnie ćwiczenie w rozdziale 6.). Taki zespół drzew decyzyjnych nazywamy **losowym lasem** (ang. *random forest*). Pomimo złożoności stanowi on jeden z najpotężniejszych współczesnych algorytmów uczenia maszynowego.

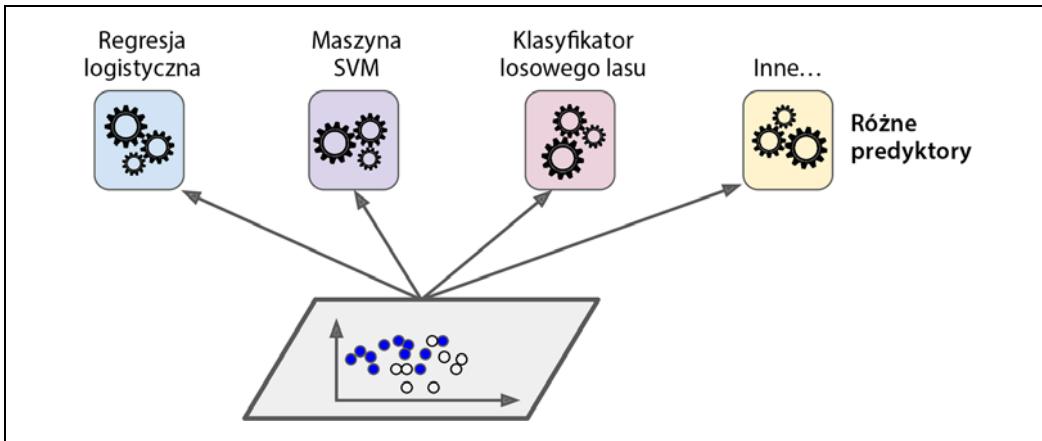
Jak już wspomniałem w rozdziale 2., będziemy często korzystać z metod zespołowych pod koniec projektu, już po utworzeniu kilku dobrych predyktorów — będziemy je łączyć w jeden jeszcze lepszy model. Rzeczywiście, w różnych konkursach zwyciężają najczęściej rozwiązania wykorzystujące kilka metod zespołowych (naj słynniejszy z nich jest konkurs Netflix Prize, <https://netflixprize.com/>).

W tym rozdziale przyjrzymy się najpopularniejszym metodom zespołowym, takim jak **agregacja** (ang. *bagging*), **wzmacnianie** (ang. *boosting*) i **kontaminacja** (ang. *stacking*). Przeanalizujemy również losowe lasy.

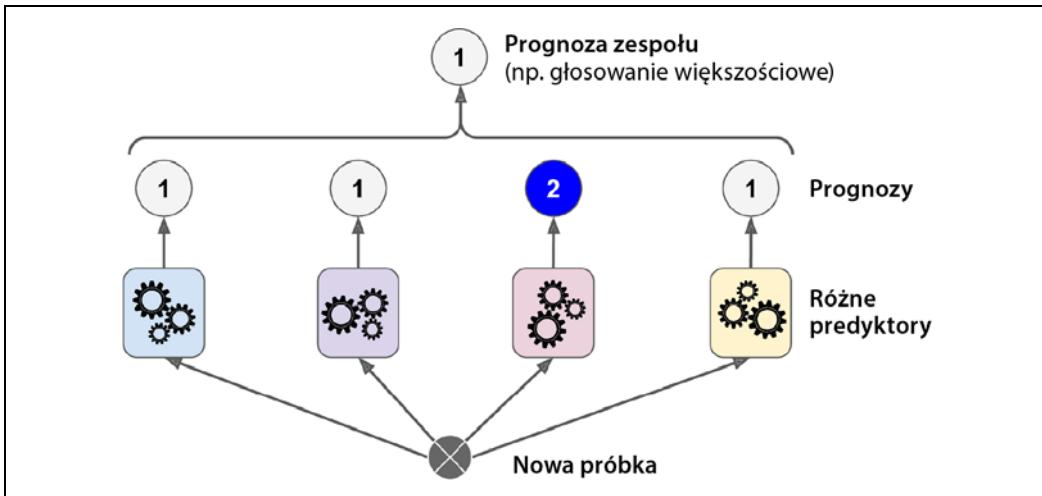
Klasyfikatory głosujące

Założymy, że wytrenowaliśmy kilka klasyfikatorów, z których każdy osiąga dokładność rzędu 80%. Może to być klasyfikator regresji logistycznej, maszyna wektorów nośnych, losowy las albo model k-najbliższych sąsiadów, a także kilka innych (rysunek 7.1).

Bardzo prostą metodą uzyskania jeszcze lepszego klasyfikatora jest połączenie prognoz wyliczanych przez poszczególne klasyfikatory i wybranie klasy, która uzyskała najwięcej głosów. Klasyfikator wykorzystujący mechanizm głosowania większościowego jest nazywany **klasyfikatorem głosującym większościowo** (ang. *hard voting classifier*; rysunek 7.2).



Rysunek 7.1. Uczenie różnych klasyfikatorów

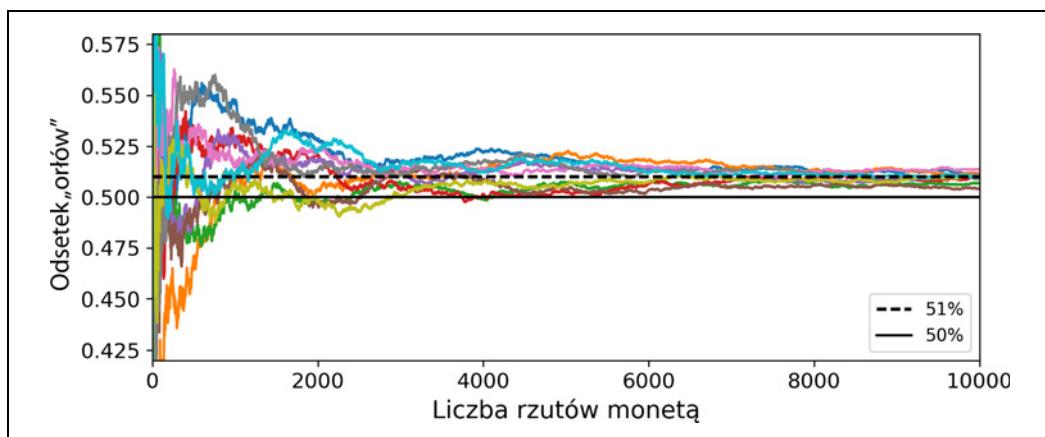


Rysunek 7.2. Prognozy klasyfikatora głosowania większościowego

Co ciekawe, taki klasyfikator głosujący często osiąga większą dokładność od najlepszego klasyfikatora w zespole. W istocie, nawet jeśli mamy do czynienia z **klasyfikatorem słabym** (ang. *weak learner*; tzn. że radzi sobie tylko nieco lepiej od przypadkowego zgadywania), cały zespół może mimo wszystko stanowić **klasyfikator silny** (ang. *strong learner*; osiągający dużą dokładność) przy założeniu, że dysponujemy odpowiednimi liczbą i zróżnicowaniem tych słabych klasyfikatorów.

Jak to możliwe? Poniższa analogia pomoże nam rozwikłać tę tajemnicę. Powiedzmy, że mamy nieco „oszukaną” monetę, gwarantującą 51% prawdopodobieństwa, że wypadnie orzeł, i 49%, że trafimy na reszkę. Jeżeli tysiąckrotnie rzucimy tą monetą, mniej więcej 510 razy wypadnie orzeł i 490 reszka, zatem orłów będzie więcej. Jeżeli pobawimy się w liczenie, okazuje się, że prawdopodobieństwo wystąpienia większości orłów po tysiącu rzutów jest bliskie 75%. Im więcej razy rzucisz monetą, tym bardziej zwiększasz to prawdopodobieństwo (np. po 10 000 rzutów wynosi ono już ponad 97%).

Jest to wynik **prawa wielkich liczb**: jeśli będziesz ciągle rzucać monetą, odsetek orłów będzie zbliżał się do prawdopodobieństwa ich wyrzucenia (51%). Na rysunku 7.3 widzimy 10 serii rzutów taką „oszukaną” monetą. Jak widać, wraz z liczbą rzutów odsetek wyrzuconych orłów zbliża się do 51%. Ostatecznie wszystkie 10 serii kończy się tak blisko odsetka 51%, że ani razu nie przekraczają progu 50%.



Rysunek 7.3. Prawo wielkich liczb

Załóżmy na zasadzie analogii, że tworzymy zespół składający się z klasyfikatorów, z których dokładność każdego oscyluje zaledwie na poziomie 51% (tylko nieznacznie lepiej od losowego zgadywania). Jeżeli przewidzimy klasę za pomocą głosowania większościowego, uzyskamy dokładność rzędu 75%! Jest to jednak prawdę jedynie wtedy, gdy wszystkie klasyfikatory są doskonale niezależne i popełniają nieskorelowane błędy, co w naszym przypadku jest niemożliwe, gdyż są trenowane za pomocą tego samego zbioru uczącego. Najprawdopodobniej będą popełniać takie same błędy, przez co głosowanie większościowe będzie nieraz dotyczyło nieprawidłowej klasy, co spowoduje zmniejszenie dokładności zespołu.



Metody zespołowe sprawują się najlepiej, gdy poszczególne predyktory są od siebie w jak największym stopniu uniezależnione. Jednym ze sposobów osiągnięcia tego stanu jest uczenie klasyfikatorów za pomocą odmiennych algorytmów. Zwiększymy w ten sposób prawdopodobieństwo, że klasyfikatory te będą popełniały zupełnie różne rodzaje błędów, co prowadzi do poprawienia dokładności zespołu.

Za pomocą poniższego kodu stworzymy i wyuczymy klasyfikator głosujący w module Scikit-Learn zawierający trzy różne klasyfikatory (zestaw uczący składa się z danych sierpowatych, zaprezentowanych już w rozdziale 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)
```

Przyjrzyjmy się dokładności tego klasyfikatora wobec zbioru testowego:

```
>>> 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 0.896  
SVC 0.888  
VotingClassifier 0.904
```

No proszę! Klasyfikator głosujący osiąga nieco lepsze wyniki od jego elementów składowych.

Jeżeli wszystkie klasyfikatory są w stanie przewidywać prawdopodobieństwo przynależności do klasy (np. wszystkie one korzystają z metody `predict_proba()`), to możemy sprawić, że moduł Scikit-Learn będzie przewidywał klasę o największym prawdopodobieństwie poprzez uśrednienie wyników uzyskanych z poszczególnych klasyfikatorów. Jest to tak zwane **głosowanie miękkie** (ang. *soft voting*). Często uzyskujemy za jego pomocą lepszą wydajność niż w przypadku głosowania większościowego, ponieważ głosy o większej pewności mają nadawaną wyższą wagę. Wystarczy zastąpić parametr `voting="hard"` wartością `voting="soft"` i upewnić się, że wszystkie klasyfikatory są w stanie szacować prawdopodobieństwo przynależności do klasy. Wyjątek stanowi klasa SVC, w której należy wyznaczyć wartość True hiperparametru `probability` (zacznie być wtedy używany sprawdzian krzyżowy do szacowania prawdopodobieństwa, co spowolni cały proces uczenia, oraz zostanie dodana metoda `predict_proba()`). Jeśli dostosujesz powyższy kod do głosowania miękkiego, zauważysz, że klasyfikator głosujący osiąga ponad 91,2% dokładności!

Agregacja i wklejanie

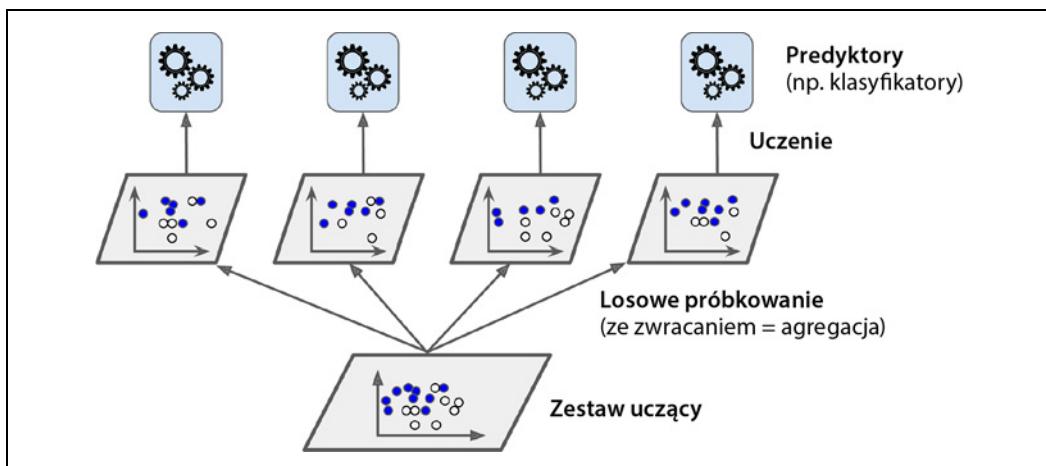
Jak już wiemy, jednym ze sposobów na uzyskanie różnorodnych typów klasyfikatorów jest używanie odmiennych algorytmów uczących. Innym rozwiązaniem jest trenowanie każdego predyktora za pomocą tego samego algorytmu przy użyciu różnych, losowych podzbiorów zbioru uczącego. Gdy próbkowanie jest przeprowadzane **ze zwracaniem** (ang. *sampling with replacement*), mechanizm ten jest nazywany **agregacją**¹ (ang. *bagging*; nazwa jest skrótem angielskiego pojęcia *bootstrap aggregating*², czyli **agregacja przykładów wstępnych**). Z kolei jeśli próbkowanie jest przeprowadzane **bez zwracania** (ang. *sampling without replacement*), to mamy do czynienia z **wklejaniem**³ (ang. *pasting*).

¹ Leo Breiman, *Bagging Predictors*, „Machine Learning” 24, no. 2 (1996), s. 123 – 140, <https://link.springer.com/article/10.1007/BF00058655>.

² W statystyce metody przepróbkowania ze zwracaniem noszą miano metod samowspornych (ang. *bootstrapping*).

³ Leo Breiman, *Pasting Small Votes for Classification in Large Databases and On-Line*, „Machine Learning” 36, no. 1 – 2 (1999), s. 85 – 103, <https://link.springer.com/article/10.1023/A:1007563306331>

Innymi słowy, zarówno agregacja, jak i wklejanie umożliwiają próbkowanie przykładów uczących kilka razy wobec wszystkich predyktorów, ale tylko w tym pierwszym przypadku możemy kilka razy przekazywać przykłady temu samemu predyktorowi. Omawiany proces próbkowania i uczenia został указанny na rysunku 7.4.



Rysunek 7.4. Zarówno podczas agregacji, jak i wklejania następuje uczenie kilku predyktorów za pomocą kilku różnych, losowych przykładów z zestawu danych uczących

Po wyuczeniu wszystkich predyktorów zespół wylicza prognozę dla nowej próbki, zbierając przewidywania wygenerowane przez poszczególne elementy składowe. Funkcję aggregującą najczęściej stanowi **dominanta** (np. najczęściej wyliczana prognoza, podobnie jak w przypadku głosowania większościowego) w zadaniach klasyfikacji lub średnia w przypadku regresji. Każdy pojedynczy predyktor cechuje się większym obciążeniem, niż gdyby był uczony wobec pierwotnego zbioru danych, ale agregacja zmniejsza zarówno obciążenie, jak i wariancję⁴. Najczęściej zespół uzyskuje podobne obciążenie, jak pojedynczy predyktor wyuczony na zbiorze danych uczących, ale niższą wariancję.

Jak widać na rysunku 7.4, wszystkie predyktory mogą być uczone równolegle, przy użyciu osobnych rdzeni procesora, a nawet serwerów. W podobny sposób możemy również wyliczać prognozy. Jest to jeden z powodów wielkiej popularności agregacji i wklejania: obydwie metody są bardzo skalowalne.

Agregacja i wklejanie w module Scikit-Learn

Moduł Scikit-Learn zawiera prosty interfejs API obsługujący zarówno agregację, jak i wklejanie przy użyciu klasy BaggingClassifier (odpowiednikiem regresyjnym jest BaggingRegressor). Za pomocą poniższego kodu uczymy zespół 500 klasyfikatorów drzew decyzyjnych⁵: każde z nich jest treinowane przy użyciu podzbioru 100 przykładów losowo dobieranych ze zwracaniem (prezentuję tu przykład agregacji; jeżeli chcesz sprawdzić możliwości wklejania, zmień wartość parametru bootstrap

⁴ Pojęcia obciążenia i wariancji zostały wyjaśnione w rozdziale 4.

⁵ Możemy ewentualnie wyznaczyć wartość zmiennoprzecinkową (w zakresie pomiędzy 0.0 a 1.0) w parametrze max_samples — w tym przypadku maksymalna liczba próbkowanych przykładów jest równa rozmiarowi zestawu uczącego pomnożonemu przez wartość max_samples.

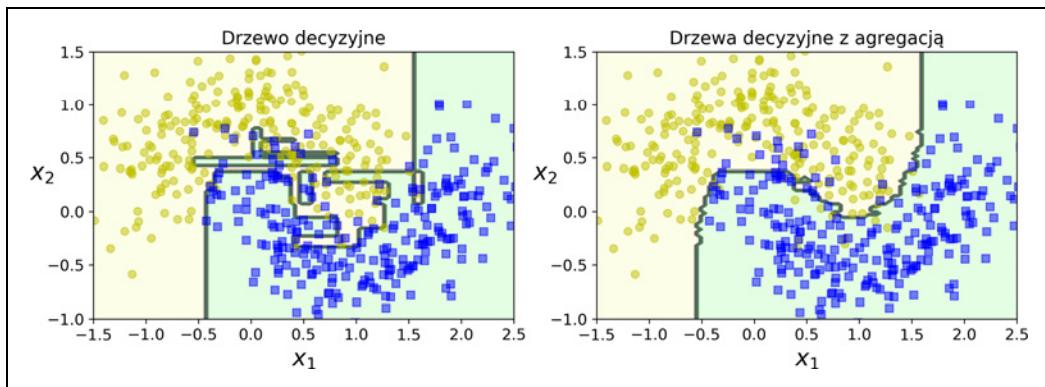
na `False`). W parametrze `n_jobs` wyznaczamy liczbę rdzeni procesora wykorzystywanych do uczenia i wyliczania prognoz (wartość `-1` oznacza wykorzystanie wszystkich dostępnych rdzeni):

```
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)
```



Klasa `BaggingClassifier` automatycznie przeprowadza głosowanie miękkie, jeżeli bazowy klasyfikator jest w stanie oszacować prawdopodobieństwo przynależności do klasy (tj. jeśli korzysta z metody `predict_proba()`), o czym należy pamiętać podczas korzystania z klasyfikatorów drzewa decyzyjnego.

Na rysunku 7.5 porównujemy granicę decyzyjną pojedynczego drzewa decyzyjnego z granicą decyzyjną generowaną przez agregacyjny zespół 500 drzew (zob. powyższy kod); obydwa modele zostały wyuczone wobec sierpowatego zbioru danych. Jak widać, przewidywania zespołu prawdopodobnie będą znacznie lepiej generalizować wyniki do nowych próbek niż w przypadku pojedynczego drzewa: zespół ten cechuje podobne obciążenie i mniejsza wariancja (popołnia w przybliżeniu tyle samo błędów na zbiorze uczącym, ale jego granica decyzyjna ma bardziej regularny kształt).



Rysunek 7.5. Porównanie pojedynczego drzewa decyzyjnego (po lewej) z agregacją zespołu 500 drzew decyzyjnych (po prawej)

Metody samowsporne wprowadzają nieco większe zróżnicowanie do podzbiorów uczących, dlatego agregacja ma nieco większe obciążenie od wklejania, taka dodatkowa różnorodność oznacza jednak również, że predyktory są ze sobą mniej skorelowane, co skutkuje zmniejszeniem wariancji. Generalnie dzięki agregacji zyskujemy często lepsze modele, dlatego to raczej ona jest bardziej polecana. Jeżeli jednak masz czas i dużą moc obliczeniową, możesz ocenić wydajność agregacji i wklejania za pomocą sprawdzianu krzyżowego, po czym wybrać lepiej sprawujący się model.

Ocena OOB

W przypadku agregacji część przykładów może być kilkukrotnie przypisywana jednemu predyktoriowi, natomiast niektóre próbki mogą nie być wcale wykorzystane. Domyślnie klasa BaggingClassifier próbuje m przykładów uczących ze zwracaniem (bootstrap=True), gdzie m oznacza rozmiar zbioru uczącego. Oznacza to, że średnio zaledwie ok. 63% danych uczących jest próbkiwanych dla każdego predyktora⁶. Pozostałe 37% niewykorzystanych próbek nosi nazwę **przykładów pozatreningowych** (ang. *out-of-bag instances* — OOB). Pamiętaj, że na te 37% przykładów pozatreningowych może składać się inny zbiór przykładów dla każdego predyktora.

Predyktor nigdy nie widzi danych OOB podczas nauki, dlatego możemy za ich pomocą ocenić jego wydajność bez konieczności tworzenia osobnego zbioru walidacyjnego. Jesteśmy również w stanie ocenić cały zespół, uśredniając wyniki OOB pochodzące z poszczególnych predyktorów.

Możemy w module Scikit-Learn wyznaczyć parametr oob_score=True podczas tworzenia klasy BaggingClassifier, aby automatycznie uzyskiwać ocenę OOB po przeprowadzeniu treningu. Przykład tego rozwiązania został zaprezentowany na poniższym fragmencie kodu. Wynik oceny jest dostępny w zmiennej 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_  
0.901333333333332
```

Według tej oceny OOB nasz klasyfikator BaggingClassifier prawdopodobnie uzyska wynik dokładności rzędu 90,1% wobec zestawu testowego. Sprawdźmy:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred = bag_clf.predict(X_test)  
>>> accuracy_score(y_test, y_pred)  
0.9120000000000003
```

Dokładność wynosi 91,2% wobec zbioru testowego — oszacowanie OOB myli się zatem w bardzo niewielkim stopniu!

Wynik funkcji decyzyjnej OOB dla każdego przykładu uczącego jest również dostępny w zmiennej oob_decision_function_. W naszym przypadku (bazowy estymator zawiera metodę predict_proba()) funkcja decyzyjna zwraca prawdopodobieństwo przynależności do każdej klasy dla każdej próbki uczącej. Na przykład ocena OOB szacuje, że pierwszy przykład uczący ma 68,25% szans na przynależność do klasy pozytywnej (czyli 31,75% szans na przynależność do klasy negatywnej):

```
>>> 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]])
```

⁶ Wraz ze wzrostem m wartość tego współczynnika dąży do $1 - \exp(-1) \approx 63,212\%$.

Rejony losowe i podprzestrzenie losowe

Klasa BaggingClassifier obsługuje również próbkowanie cech. Kontrolujemy tę funkcję za pomocą hiperparametrów `max_features` i `bootstrap_features`. Działają one na takiej samej zasadzie, jak hiperparametry `max_samples` i `bootstrap`, dotyczą jednak cech, a nie próbek. Zatem dzięki nim każdy predyktor zostanie wyuczony wobec losowego podzbioru cech wejściowych.

Technika ta okazuje się szczególnie przydatna, gdy zajmujemy się wielowymiarowymi zbiorami danych (np. obrazami). Proces próbkowania zarówno przykładów uczących, jak i cech nosi nazwę **metydry rejonów losowych**⁷ (ang. *random patches method*). Z kolei pozostawienie przykładów uczących (poprzez wyznaczenie parametrów `bootstrap=False` i `max_samples=1.0`) przy jednoczesnym próbkowaniu samych cech (za pomocą parametrów `bootstrap_features=True` i/lub wartości `max_features` mniejszej od 1.0) to tak zwana **metoda podprzestrzeni losowych**⁸ (ang. *random subspaces method*).

Dzięki próbkowaniu cech zyskujemy jeszcze większe zróżnicowanie predyktorów — zwiększamy nieco obciążenie, ale redukujemy wariancję.

Losowe lasy

Wiemy już, że losowy las⁹ stanowi zespół drzew decyzyjnych, uczonych najczęściej metodą agregacji (czasami wklejania), dla których wartość hiperparametru `max_samples` jest równa rozmiarowi zbioru uczącego. Zamiast konstruować klasę BaggingClassifier i przekazywać ją klasyfikatorowi DecisionTreeClassifier, możemy skorzystać z klasy RandomForestClassifier, która jest wygodniejsza i zoptymalizowana pod względem algorytmu drzew decyzyjnych¹⁰ (istnieje również regresyjna odmiana tej klasy — RandomForestRegressor). Za pomocą poniższego kodu wyuczymy losowy las składający się z 500 drzew (każde ograniczamy do 16 węzłów) za pomocą wszystkich dostępnych rdzeni procesora:

```
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)
```

⁷ Gilles Louppe i Pierre Geurts, *Ensembles on Random Patches*, „Lecture Notes in Computer Science” 7523 (2012), s. 346 – 361, <https://www.semanticscholar.org/paper/Ensembles-on-Random-Patches-Louppe-Geurts/f5ae0b1b48a21ae083f670d7043495f067064e4>.

⁸ Tin Kam Ho, *The Random Subspace Method for Constructing Decision Forests*, „IEEE Transactions on Pattern Analysis and Machine Intelligence” 20, no. 8 (1998), s. 832 – 844, https://www.researchgate.net/publication/3192880_The_Random_Subspace_Method_for_Constructing_Decision_Forests.

⁹ Tin Kam Ho, *Random Decision Forests*, „Proceedings of the Third International Conference on Document Analysis and Recognition” 1 (1995), s. 278, <https://ieeexplore.ieee.org/abstract/document/598994>.

¹⁰ Klasa BaggingClassifier pozostaje przydatna, jeżeli chcemy korzystać z modelu innego niż drzewa decyzyjne.

Oprócz kilku wyjątków klasa RandomForestClassifier zawiera wszystkie hiperparametry klasyfikatora DecisionTreeClassifier (regulujące sposób wzrostu drzew) oraz klasyfikatora BaggingClassifier (sterujące działaniem samego zespołu)¹¹.

Algorytm losowego lasu wprowadza dodatkową losowość do wzrostu drzew; zamiast wyszukiwania najlepszej cechy podczas podziału na węzły podrzędne (zob. rozdział 6.) szuka on najlepszej cechy wśród losowego podzbioru cech. W ten sposób algorytm skutkuje większym zróżnicowaniem drzew, które okupujemy (znowu) większym obciążeniem w zamian za niższą wariancję, co najczęściej oznacza lepszy model. Poniżej zaprezentowana klasa BaggingClassifier stanowi z grubsza odpowiednik poprzednio zdefiniowanego klasyfikatora RandomForestClassifier:

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),  
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Zespół Extra-Trees

Podczas wzrostu drzewa w modelu losowego lasu w każdym węzle jedynie losowy podzbior cech jest brany pod uwagę podczas podziału na węzły potomne (omówiliśmy to już wcześniej). Jesteśmy w stanie generować jeszcze bardziej przypadkowe drzewa poprzez wprowadzenie losowych progów dla każdej cechy zamiast poszukiwania najlepszych możliwych progów (jak w modelu drzewa decyzyjnego).

Taki maksymalnie przypadkowy las jest nazywany zespołem **ekstremalnie losowych drzew**¹² (ang. *extremely randomized trees*; w skrócie **Extra-Trees**). Także w tej technice obniżamy wariancję kosztem większego obciążenia. Uczenie modelu Extra-Trees jest również znacznie szybsze od trenowania standardowego losowego lasu, gdyż znalezienie najlepszego możliwego progu dla każdej cechy w każdym węźle stanowi jedno z najbardziej czasochłonnych zadań podczas wzrostu drzewa.

Możemy utworzyć klasyfikator Extra-Trees za pomocą klasy ExtraTreesClassifier. Jej interfejs jest taki sam, jak klasyfikatora RandomForestClassifier. Na zasadzie analogii klasa ExtraTreesRegressor ma identyczną strukturę jak klasa RandomForestRegressor.



Trudno z góry przewidzieć, czy klasa RandomForestClassifier będzie sprawowała się lepiej, czy gorzej od klasyfikatora ExtraTreesClassifier. Zazwyczaj jedynym sposobem jest wypróbowanie obydwu modeli i porównanie ich za pomocą sprawdzianu krzyzowego (dostrojenie hiperparametrów przy użyciu przeszukiwania siatki).

Istotność cech

Kolejną znakomitą cechą losowych lasów jest łatwość pomiaru względnej istotności każdej cechy. Moduł Scikit-Learn mierzy istotność cechy poprzez sprawdzenie, w jakim stopniu węzły (we wszystkich

¹¹ Należy wymienić kilka godnych uwagi wyjątków: brakuje hiperparametrów splitter (ma wymuszoną wartość "random"), presort (wymuszona wartość False), max_samples (wymuszona wartość 1.0) i base_estimator (wymuszona wartość DecisionTreeClassifier wraz z podanymi hiperparametrami).

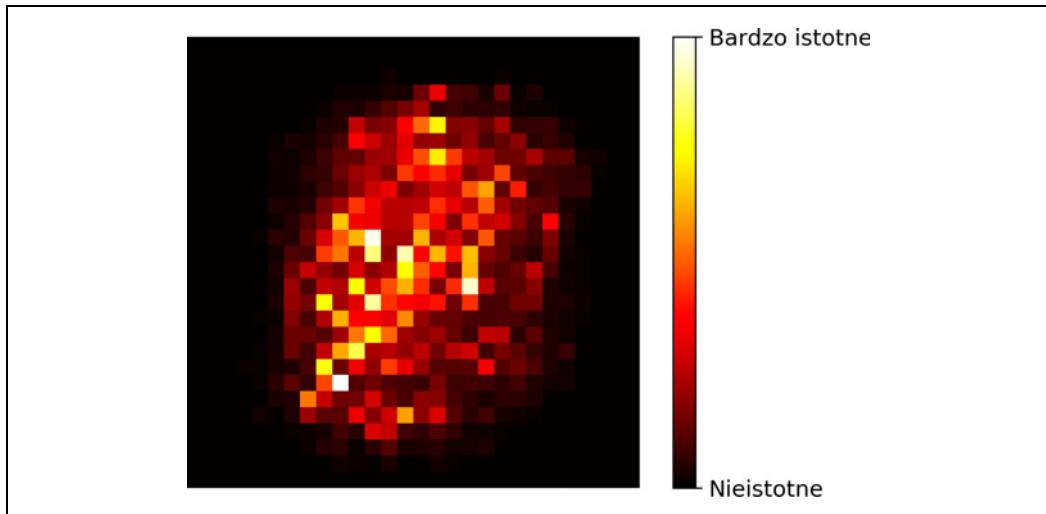
¹² Pierre Geurts i in., *Extremely Randomized Trees*, „Machine Learning” 63, no. 1 (2006), s. 3 – 42, <https://link.springer.com/article/10.1007/s10994-006-6226-1>.

drzewach lasu) wykorzystujące tę cechę zmniejszają zanieczyszczenie. Mówiąc dokładniej, mamy tu do czynienia ze średnią ważoną, gdzie waga każdego węzła jest równa liczbie powiązanych z nim próbek uczących (zob. rozdział 6.).

Moduł Scikit-Learn po treningu automatycznie oblicza wynik dla każdej cechy, a następnie skaluje rezultaty tak, że suma wszystkich istotności wynosi 1. Wyniki te są przechowywane w zmiennej `feature_importances_`. Przykładowo, poniższy kod służy do uczenia klasyfikatora RandomForestClassifier wobec zbioru danych Iris (korzystaliśmy już z niego w rozdziale 4.), po czym wyświetla istotność każdej cechy. Wygląda na to, że najistotniejsze są długość płatka (44%) i jego szerokość (42%), natomiast w stosunku do nich długość i szerokość działki nie grają większej roli (ich wartości to, odpowiednio, 11% i 2%):

```
>>> 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
```

W podobny sposób możemy wytrenować klasyfikator losowego lasu wobec zestawu danych MNIST (używanego przez nas w rozdziale 3.) i stworzyć wykres istotności każdego piksela — efekt został zaprezentowany na rysunku 7.6.



Rysunek 7.6. Istotność pikseli ze zbioru danych MNIST (według klasyfikatora losowego lasu)

Losowe lasy bardzo pomagają w określeniu, które cechy mają naprawdę znaczenie, zwłaszcza jeśli musisz przeprowadzić dobór cech.

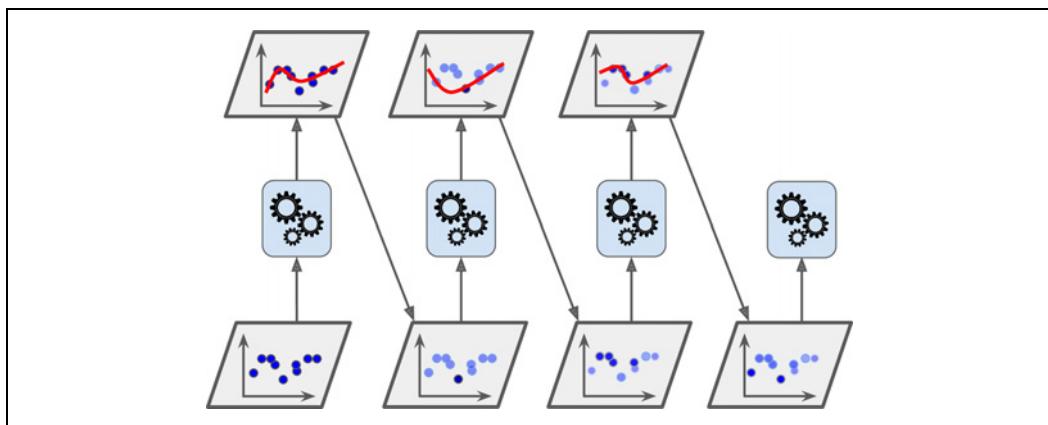
Wzmacnianie

Wzmacnianie (ang. *boosting*; pierwotnie zwane **wzmacnianiem hipotezy** — ang. *hypothesis boosting*) dotyczy każdej metody zespołowej łączącej kilka klasyfikatorów słabych w klasyfikator silny. Podstawową koncepcją w większości metod wzmacniania jest sekwencyjne uczenie predyktorów w taki sposób, że każdy następny próbuje korygować poprzednika. Istnieje wiele metod wzmacniania, do tej pory jednak najpopularniejsze pozostają algorytmy **AdaBoost**¹³ (skróć angielskiego pojęcia *Adaptive Boosting*, czyli **wzmacnianie adaptacyjne**) i **wzmacnianie gradientowe** (ang. *gradient boosting*). Zaczniemy od metody AdaBoost.

AdaBoost

Nowy predyktor może korygować swojego poprzednika poprzez zwracanie większej uwagi na przykłady uczące, dla których poprzedni algorytm pozostał niedotrenowany. W ten sposób kolejne predyktory koncentrują się coraz bardziej na najtrudniejszych przypadkach. Właśnie ta technika jest wdrażana przez algorytm AdaBoost.

Przykładowo podczas uczenia klasyfikatora AdaBoost najpierw zostaje wytrenowany klasyfikator bazowy (taki jak drzewo decyzyjne), a potem jest on użyty do uzyskania prognoz dla zbioru uczącego. Następnie algorytm zwiększa względną wagę nieprawidłowo sklasyfikowanych próbek uczących. Teraz uczony jest drugi klasyfikator za pomocą zaktualizowanych wag, znowu następuje wyliczenie predykcji dla zbioru uczącego, wagi zostają zaktualizowane itd. (rysunek 7.7).

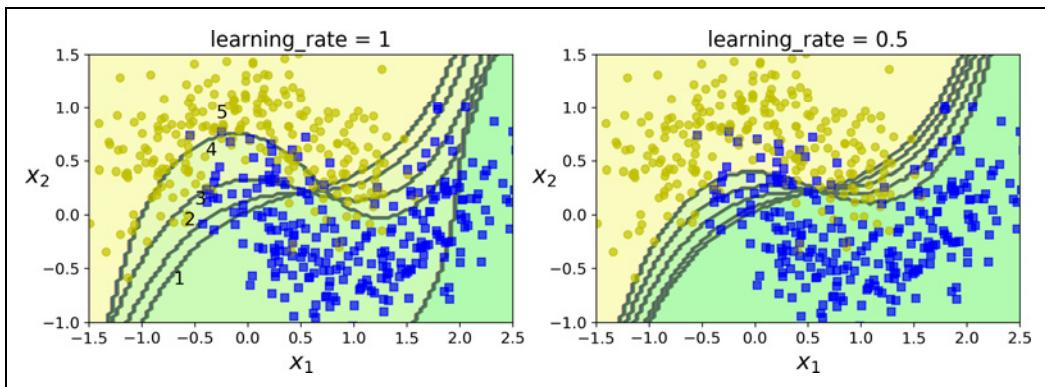


Rysunek 7.7. Mechanizm AdaBoost — uczenie sekwencyjne przy użyciu aktualizowanych wag przykładów

Rysunek 7.8 prezentuje granice decyzyjne pięciu kolejnych predyktorów dla zbioru danych sierpowatych (w tym przykładzie każdym predyktorem jest mocno regularyzowany klasyfikator SVM wykorzy-

¹³ Yoav Freund i Robert E. Schapire, A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting, „Journal of Computer and System Sciences” 55, no. 1 (1997), s. 119 – 139, <https://www.sciencedirect.com/science/article/pii/S00220009791504X>.

stującym jądro RBF¹⁴). Pierwszy klasyfikator myli się co do wielu próbek, dlatego ich wagi zostają podniesione. Drugi klasyfikator radzi sobie z nimi już lepiej itd. Na prawym wykresie ukazana jest ta sama sekwencja predyktorów z wartością współczynnika uczenia zmniejszoną o połowę (tzn. wagi nieprawidłowo sklasyfikowanych przykładów są wzmacniane o połowę słabiej). Jak widać, ta technika sekwencyjnego uczenia wykazuje pewne podobieństwa do metody gradientu prostego — różnica polega na tym, że nie poprawiamy parametrów pojedynczego predyktora w celu zminimalizowania funkcji kosztu, lecz dodajemy predyktory do zespołu, przez co stopniowo go usprawniamy.



Rysunek 7.8. Granice decyzyjne sekwencyjnie ułożonych predyktorów

Po wyuczeniu wszystkich predyktorów zespół wylicza prognozy w sposób bardzo zbliżonych do agregacji lub wklejania, jednak tutaj każdy predyktor ma inną wagę, w zależności od jego dokładności wobec ważonego zbioru uczącego.



Technika ta ma jedno poważne ograniczenie: nie można równolegle uczyć predyktorów (a przynajmniej nie całkowicie), ponieważ każdy z nich może zostać wytrenowany dopiero po wyuczeniu i ocenieniu poprzednika. W związku z tym rozwiązanie to nie jest tak skalowalne, jak agregacja czy wklejanie.

Przyjrzymy się uważniej algorytmowi AdaBoost. Waga każdej próbki $w^{(i)}$ ma początkowo wyznaczoną wartość $\frac{1}{m}$. Pierwszy predyktor zostaje wytrenowany, a następnie zostaje wyliczony jego ważony współczynnik błędu r_1 (zob. równanie 7.1) (pamiętaj, że wartość mianownika zawsze jest tu równa 1).

Równanie 7.1. Ważony współczynnik błędu dla j -tego predyktora

$$r_j = \frac{\sum_{i=1}^m w^{(i)}_{j(i) \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}}$$

¹⁴ Użyłem tego rozwiązania jedynie w celach poglądowych. Maszyny SVM zazwyczaj nie są dobrymi predyktorami bazowymi dla algorytmu AdaBoost, ponieważ w takiej kombinacji okazują się powolne i niestabilne.

gdzie $\hat{y}_j^{(i)}$ jest prognozą j -tego predyktora dla i -tego przykładu.

Następnie za pomocą równania 7.2 wyliczamy wagę predyktora α_j , gdzie η stanowi współczynnik uczenia (domyślnie jego wartość wynosi 1)¹⁵. Im dokładniejszy predyktor, tym zostaje mu nadana wyższa waga. W przypadku niemal losowego zgadywania waga predyktora będzie zbliżona do zera. Jeśli jednak będzie się mylił w większości przypadków (będzie miał dokładność jeszcze mniejszą od losowego zgadywania), to uzyska negatywną wagę.

Równanie 7.2. Waga predyktora

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

Kolejnym etapem jest aktualizowanie wag próbek przy użyciu równania 7.3: zostają wzmacnione nieprawidłowo sklasyfikowane przykłady.

Równanie 7.3. Reguła aktualizowania wag

dla $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{jeśli } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{jeśli } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

Teraz wszystkie wagi przykładów zostają znormalizowane (tj. podzielone przez $\sum_{i=1}^m w^{(i)}$).

Na koniec uczymy nowy predyktor za pomocą zaktualizowanych wag i cały proces zostaje powtórzony (wyliczenie wagi nowego predyktora, aktualizacja wag próbek, trenowanie nowego predyktora itd.). Algorytm przestaje działać, gdy osiągniemy założoną liczbę predyktorów lub gdy otrzymamy doskonały predyktor.

W celu uzyskania prognoz algorytm AdaBoost oblicza po prostu przewidywania wszystkich predyktorów i sprawdza ich wagę za pomocą współczynnika α_j . Prognozowana klasa zostaje wybrana w drodze głosowania większościowego (równanie 7.4).

Równanie 7.4. Prognozy algorytmu AdaBoost

$$\hat{y}(\mathbf{x}) = \arg \max_k \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^N \alpha_j$$

gdzie N oznacza liczbę predyktorów.

Moduł Scikit-Learn wykorzystuje wieloklasową wersję algorytmu AdaBoost o nazwie **SAMME**¹⁶ (ang. *Stagewise Additive Modeling using a Multiclass Exponential loss function* — sekwencyjne modelowanie

¹⁵ Oryginalny algorytm AdaBoost nie zawiera współczynnika uczenia.

¹⁶ Więcej szczegółów znajdziesz w: Ji Zhu i in., *Multi-Class AdaBoost*, „Statistics and Its Interface” 2, no. 3 (2009), s. 349 – 360, <https://www.intlpress.com/site/pub/pages/journals/items/sii/content/vols/0002/0003/a008/>.

addytywne przy użyciu wieloklasowej, wykładniczej funkcji straty). Gdy mamy do czynienia tylko z dwiema klasami, algorytm SAMME staje się równoważny metodzie AdaBoost. Jeśli predyktory są w stanie szacować prawdopodobieństwo przynależności do danej klasy (tj. zawierają metodę `predict_proba()`), to moduł Scikit-Learn może korzystać z wariantu **SAMME.R** (skrót R oznacza angielski wyraz *real*, czyli „rzeczywisty”), w którym wyliczane są właśnie prawdopodobieństwa, a nie prognozy, dzięki czemu cechują się one większą wydajnością.

Poniższy kod wyucza klasyfikator AdaBoost na podstawie 200 **pieńków decyzyjnych** (ang. *decision stump*) przy użyciu klasy `AdaBoostClassifier` (jak łatwo przewidzieć, istnieje też klasa `AdaBoostRegressor`). Pieńkiem decyzyjnym nazywamy drzewo decyzyjne o wysokości `max_depth=1` — innymi słowy, jest to drzewo składające się z jednego węzła decyzyjnego i dwóch liści. Jest to domyślny estymator bazowy klasy `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)
```



Jeżeli zespół AdaBoost ulega przetrenowaniu wobec danych uczących, możesz spróbować zmniejszyć liczbę estymatorów lub mocniej regularyzować bazową klasę oszacowującą.

Wzmacnianie gradientowe

Inną bardzo popularną techniką wzmacniania jest **wzmacnianie gradientowe**¹⁷. Podobnie jak algorytm AdaBoost, wzmacnianie gradientowe dodaje kolejne predyktory do zespołu w sposób sekwencyjny, gdzie każdy następny poprawia poprzednika. Jednak nie aktualizujemy tu wag przykładów po każdym przebiegu, lecz próbujemy dopasować predyktor do **błędu resztowego** (ang. *residual error*) popełnionego przez poprzedni predyktor.

Przeanalizujmy prosty przykład regresji, w którym bazowymi predyktorami będą drzewa decyzyjne (oczywiście wzmacnianie gradientowe działa również znakomicie z zadaniami klasyfikacji). Mechanizm ten nazywamy **gradientowym wzmacnianiem drzew** (ang. *gradient tree boosting*) lub **drzewami regresyjnymi wzmacnianymi gradientowo** (ang. *gradient boosted regression trees* — GBRT). Wyuczmy najpierw klasę `DecisionTreeRegressor` wobec zestawu uczącego (np. zaszumionych danych kwadratowych):

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg1 = DecisionTreeRegressor(max_depth=2)  
tree_reg1.fit(X, y)
```

¹⁷ Pierwszy raz zostało zaprezentowane w artykule Leo Breimana *Arcing the Edge* (<https://pdfs.semanticscholar.org/8162/f9036f5b7a2a05fed1148cb04d5355c0f213.pdf>), a w artykule z 1999 r. *Greedy Function Approximation: A Gradient Boosting Machine* (https://www.jstor.org/stable/2699986?seq=1#page_scan_tab_contents) Jerome H. Friedman dalej je rozwinął.

Następnie wytrenujemy drugą klasę `DecisionTreeRegressor` wobec błędów resztowych popełnionych przez pierwszy predyktor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Teraz uczymy trzeci regresor przy użyciu błędów resztowych popełnionych przez poprzednika:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Mamy teraz zespół składający się z trzech drzew. Może on wyliczać prognozy dla nowych próbek poprzez zsumowanie predykcji uzyskanych przez wszystkie trzy klasyfikatory:

```
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
```

Rysunek 7.9 prezentuje po lewej stronie prognozy tych trzech drzew, natomiast po prawej widzimy przewidywania całego zespołu. W pierwszym rzędzie zespół składa się tylko z jednego drzewa, zatem prognozy pierwszego drzewa i zespołu są na tym etapie identyczne. W drugim rzędzie nowe drzewo zostało wyuczone wobec błędów resztowych pierwszego drzewa. Łatwo zauważać, że prognozy zespołu stanowią sumę przewidywań wyliczonych przez obydwa drzewa. Z kolei w trzecim rzędzie wyuczyliśmy trzeci predyktor za pomocą błędów resztowych drugiego drzewa. Widzimy, że prognozy zespołu stają się stopniowo coraz lepsze wraz z dołączaniem kolejnych drzew.

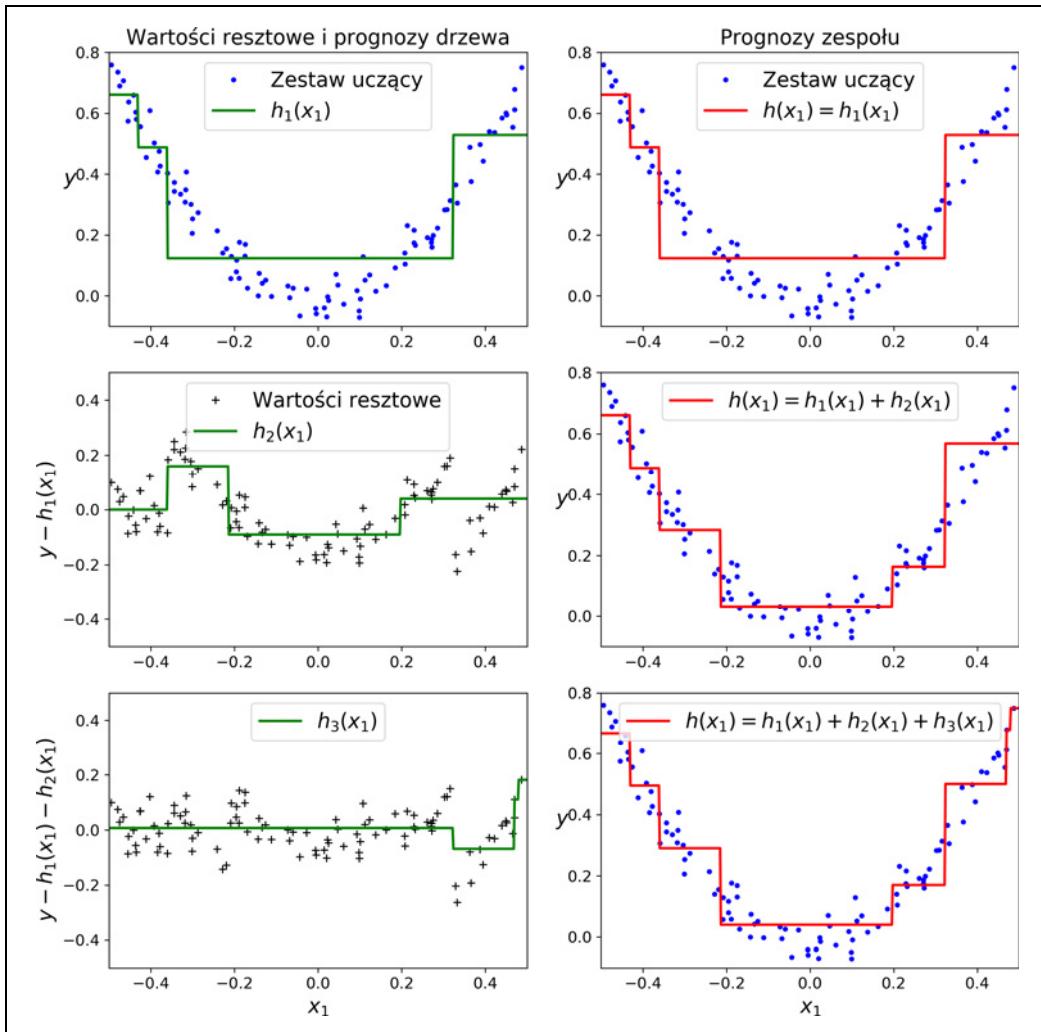
Prostszą metodą uzyskania zespołu GBRT jest użycie klasy `GradientBoostingRegressor`. Podobnie jak w przypadku klasy `RandomForestRegressor` dostępne są tu hiperparametry regulujące wzrost drzew decyzyjnych (np. `max_depth`, `min_samples_leaf`), a także kontrolujące proces uczenia zespołu, jak choćby liczbę drzew (`n_estimators`). Poniższy fragment kodu tworzy taki sam zespół, jak w poprzednim przykładzie:

```
from sklearn.ensemble import GradientBoostingRegressor

gbdt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0)
gbdt.fit(X, y)
```

Hiperparametr `learning_rate` określa udział każdego drzewa w zespole. Jeżeli wprowadzisz jego niewielką wartość, np. 0.1, będzie wymagana większa liczba drzew, ale model najczęściej będzie lepiej generalizował wyniki. Jest to technika regularyzacji zwana **kurczliwością** (ang. `shrinkage`). Na rysunku 7.10 zademonstrowałem dwa zespoły GBRT wytrenowane z niską wartością współczynnika uczenia: model po lewej zawiera zbyt mało drzew, aby móc się dopasować do danych, natomiast na prawym wykresie zespół zawiera zbyt dużo drzew i ulega przetrenowaniu.

W celu znalezienia optymalnej liczby możemy skorzystać z mechanizmu wcześniego zatrzymywania (zob. rozdział 4.). Możemy go z łatwością zaimplementować, wprowadzając metodę `staged_predict()`: zostaje zwrócony iterator zawierający prognozy wyliczone przez zespół na każdym etapie uczenia (po utworzeniu pierwszego drzewa, drugiego itd.). Poniższy kod służy do wytrenowania zespołu GBRT składającego się ze 120 drzew, następnie pomiaru błędu walidacji na każdym etapie w poszukiwaniu optymalnej liczby drzew, a także do wyuczenia kolejnego klasyfikatora GBRT przy użyciu optymalnej liczby drzew:

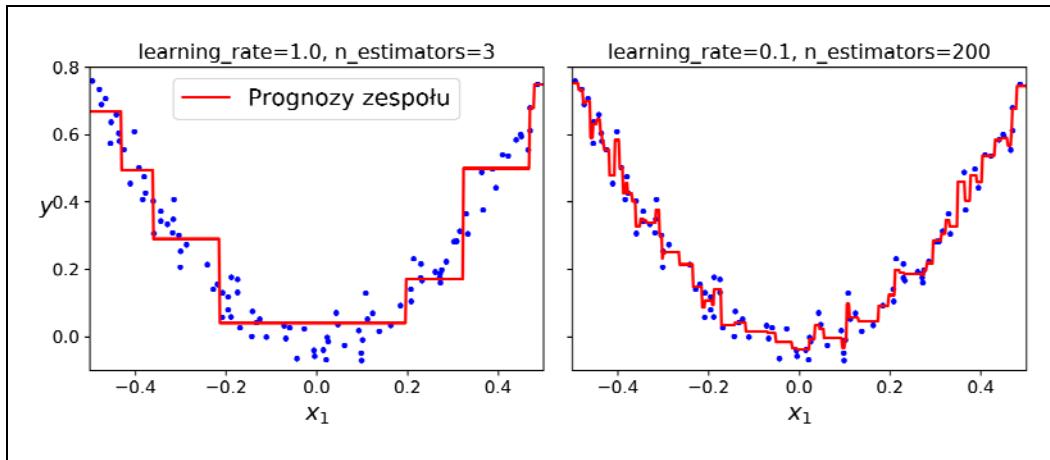


Rysunek 7.9. W tym przedstawieniu wzmacniania gradientowego pierwszy predyktor (lewy górny wykres) jest uczyony normalnie, a każdy następny (lewy środkowy i lewy dolny) zostaje wytrenowany za pomocą wartości resztowych poprzednika; prawa kolumna pokazuje prognozy uzyskane za pomocą zespołu

```
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)
```



Rysunek 7.10. Zespoły GBRT zawierające niedostatek (lewy wykres) i nadmiar predyktorów (prawy wykres)

```

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]  

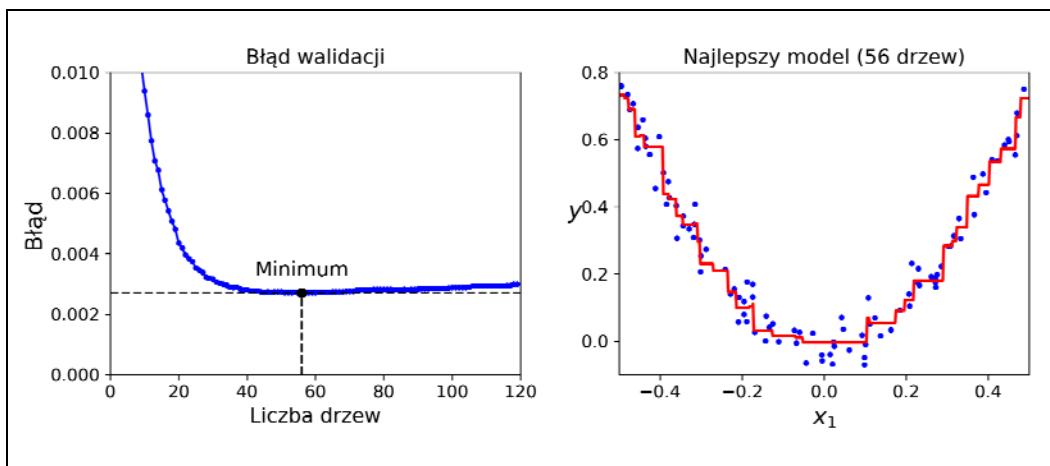
bst_n_estimators = np.argmin(errors) + 1  
  

gbrt_best = GradientBoostingRegressor(max_depth=2, n_estimators=bst_n_estimators)  

gbrt_best.fit(X_train, y_train)

```

Na lewym wykresie rysunku 7.11 widzimy błędy walidacji naszego modelu, natomiast prawy wykres przedstawia najlepsze prognozy zespołu.



Rysunek 7.11. Dobieranie liczby drzew za pomocą metody wczesnego zatrzymywania

Możemy również zaimplementować mechanizm rzeczywiście powodujący zatrzymanie algorytmu (zamiast uczenia dużej liczby drzew i następnie cofania się w poszukiwaniu optymalnego rozmiaru zespołu). Wystarczy wprowadzić parametr `warm_start=True`, dzięki czemu moduł Scikit-Learn zapamiętuje dotychczas istniejące drzewa w momencie wywołania metody `fit()`, co pozwala na uczenie

przyrostowe. Za pomocą poniższego kodu przerywamy proces uczenia w momencie, gdy błąd iteracji nie maleje w trakcie pięciu kolejnych przebiegów:

```
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 # Wczesne zatrzymywanie
```

Klasa GradientBoostingRegressor zawiera również hiperparametr subsample, wyznaczający odsetek przykładów uczących wykorzystywanych do uczenia każdego drzewa. Przykładowo, jeśli subsample=0.25, to każde drzewo będzie trenowane przy użyciu 25% losowo dobieranych próbek z całego zbioru uczącego. Jak już pewnie się domyślasz, zwiększymy w ten sposób obciążenie, ale zmniejszymy wariancję; do tego proces uczenia przebiega znacznie szybciej. Technika ta nosi nazwę **stochastycznego wzmacniania gradientowego** (ang. *stochastic gradient boosting*).



Możliwe jest również stosowanie innych funkcji kosztu we wzmacnianiu gradientowym. Wybieramy je za pomocą hiperparametru loss (więcej informacji znajdziesz w dokumentacji Scikit-Learn).

Warto zauważać, że zoptymalizowana wersja wzmacniania gradientowego jest dostępna w popularnej bibliotece XGBoost (<https://github.com/dmlc/xgboost>), której nazwę można rozszyfrować jako ekstremalne wzmacnianie gradientowe (ang. *Extreme Gradient Boosting*). Tianqi Chen stworzył pierwotnie ten pakiet jako część społeczności Distributed (Deep) Machine Learning Community (DMLC) z myślą o uzyskaniu skrajnej szybkości, skalowalności i przenośności. Rzeczywiście pakiet XGBoost często stanowi element zwycięskich projektów w różnych konkursach uczenia maszynowego. Jego API przypomina interfejs Scikit-Learn:

```
import xgboost

xgb_reg = xgboost.XGBRegressor()
xgb_reg.fit(X_train, y_train)
y_pred = xgb_reg.predict(X_val)
```

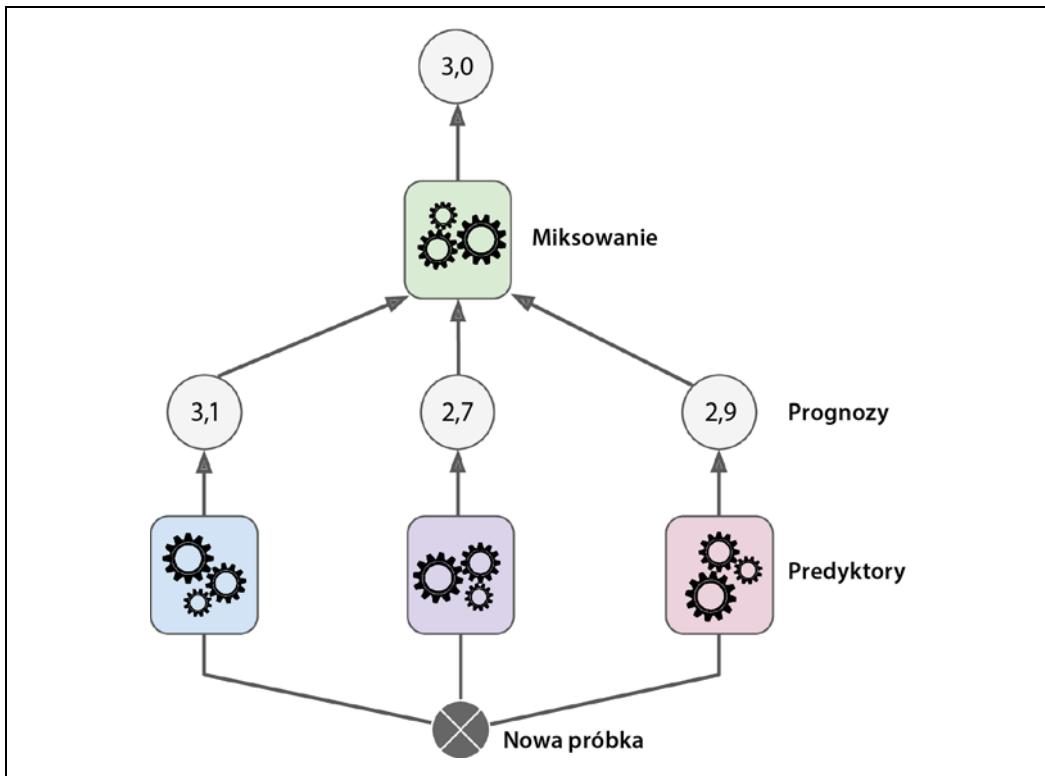
Pakiet XGBoost zawiera również kilka przydatnych funkcji, takich jak automatyzacja wczesnego zatrzymywania:

```
xgb_reg.fit(X_train, y_train,
             eval_set=[(X_val, y_val)], early_stopping_rounds=2)
y_pred = xgb_reg.predict(X_val)
```

Zdecydowanie warto się z nim zapoznać!

Kontaminacja

Ostatnia metoda zespołowa, jaką omówimy, nosi nazwę **kontaminacji** (ang. *stacking* — jest to skrót angielskiego pojęcia *stacked generalization*)¹⁸. Koncepcja jest prosta: zamiast korzystać z funkcji trywialnych (np. głosowania większościowego) w celu agregacji wszystkich prognoz predyktorów w zespole trenujemy tu model przeprowadzający tę agregację. Rysunek 7.12 pokazuje przykład takiego zespołu przeprowadzającego zadanie regresji wobec nowego przykładu. Każdy z trzech widocznych predyktorów wylicza inną prognozę (3,1, 2,7 i 2,9), a następnie ostateczny predyktor, zwany **miksrem, blenderem** (ang. *blender*) lub **metauczniem** (ang. *meta-learner*) przyjmuje te przewidywania jako dane wejściowe i za ich pomocą uzyskuje ostateczną prognozę (3,0).

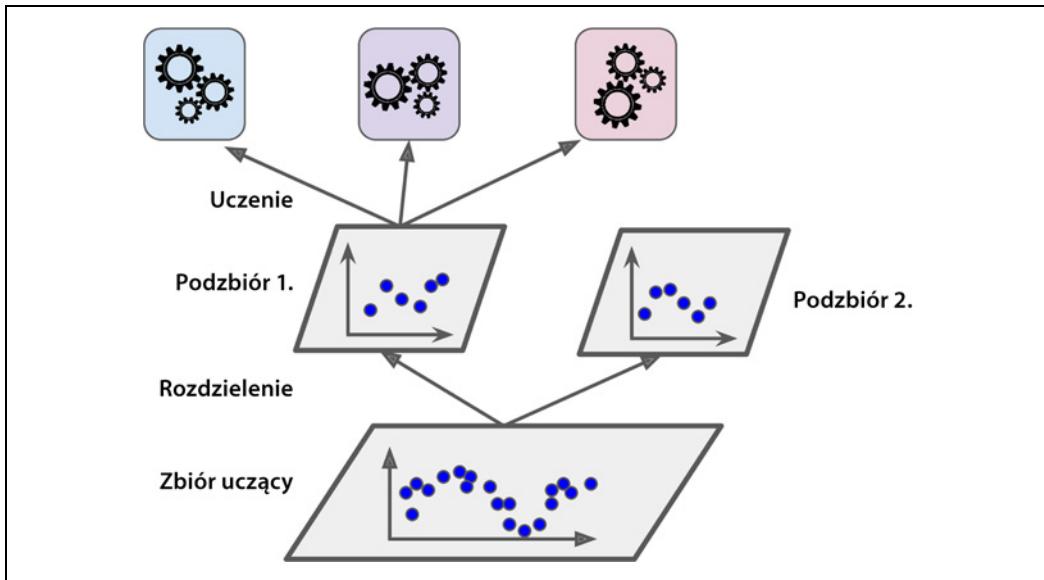


Rysunek 7.12. Łączenie prognoz za pomocą predyktora miksuującego

Powszechnie stosowaną metodą uczenia miksera jest użycie wydzielonego zbioru danych¹⁹. Przeanalizujmy jej mechanizm działania. Najpierw zbiór danych uczących zostaje podzielony na dwa podzbiory. Pierwszy podzbiór służy do uczenia pierwszej warstwy predyktorów (rysunek 7.13).

¹⁸ David H. Wolpert, *Stacked Generalization*, „Neural Networks” 5, no. 2 (1992), s. 241 – 259, <http://machine-learning.martinsewell.com/ensembles/stacking/Wolpert1992.pdf>.

¹⁹ Ewentualnie możemy skorzystać z wyliczonych prognoz. W pewnych kontekstach proces ten jest nazywany **kontaminacją**, zaś stosowanie wydzielonego zbioru danych — **miksowaniem**. Dla wielu osób jednak nazwy te są synonimiczne.

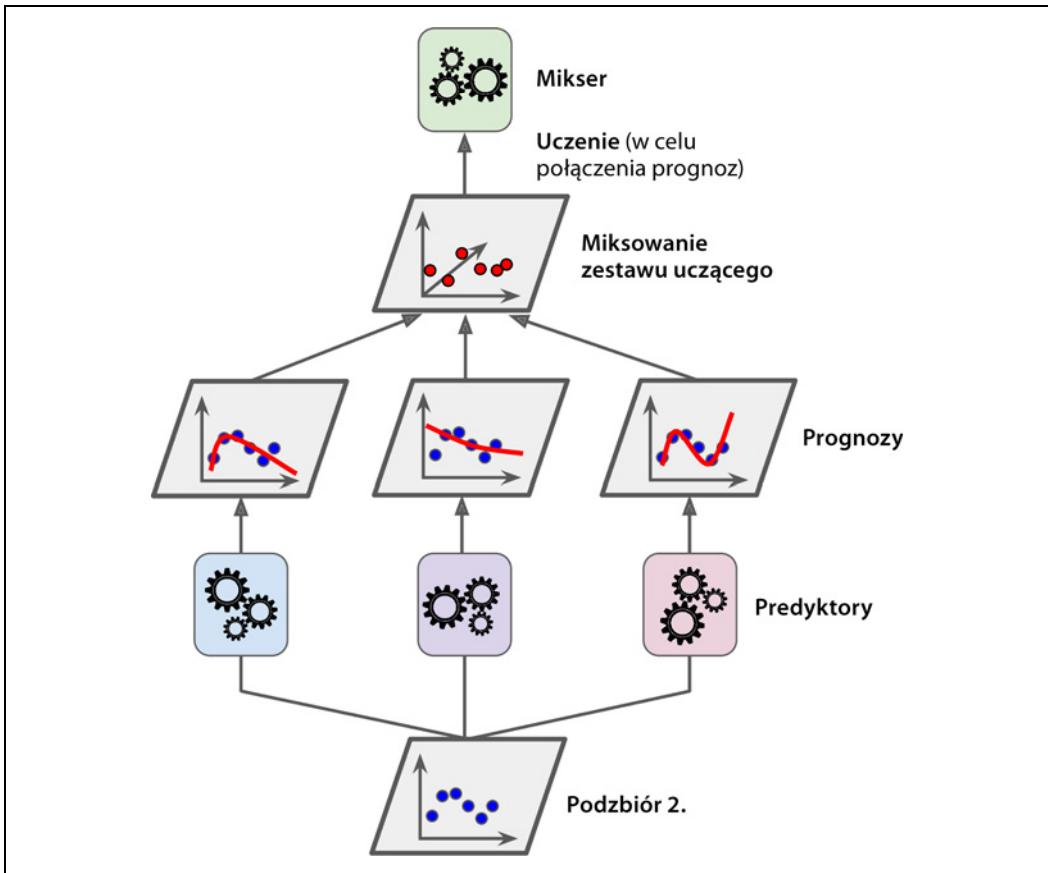


Rysunek 7.13. Uczenie pierwszej warstwy

Następnie pierwsza warstwa predyktorów wylicza prognozy dla drugiego (wydzielonego) podzbioru (rysunek 7.14). W ten sposób mamy pewność, że przewidywania będą „czyste”, gdyż predyktory nie widziały tych przykładów podczas nauki. Każda próbka z wydzielonego zbioru danych zawiera trzy prognozy. Możemy stworzyć nowy zestaw uczący, w którym te prognozy stanowić będą cechy wejściowe (przez co nowy zbiór stanie się trójwymiarowy) przy jednoczesnym zachowaniu wartości docelowych. Mikser zostaje wyuczony wobec nowego zbioru danych, dzięki czemu uczy się przewidywać wartość docelową, znając prognozy pierwszej warstwy predyktorów.

W rzeczywistości możliwe jest wyuczenie w ten sposób kilku różnych mikserów (np. jednego za pomocą regresji liniowej, innego przy użyciu regresji losowego lasu) w celu uzyskania całej warstwy mikserów. Sztuka polega na rozdzieleniu zbioru danych na trzy podzbiory: jeden posłuży do wytrenowania pierwszej warstwy, za pomocą drugiego tworzymy zestaw danych, dzięki któremu wyuczymy drugą warstwę (przy użyciu prognoz wyliczonych przez pierwszą warstwę), a trzeci podzbiór pozwoli na wytrenowanie trzeciej warstwy (za pomocą przewidywań uzyskanych w drugiej warstwie). Ostateczną prognozę dla nowego przykładu otrzymujemy, przechodząc sekwencyjnie przez każdą warstwę, co zostało ukazane na rysunku 7.15.

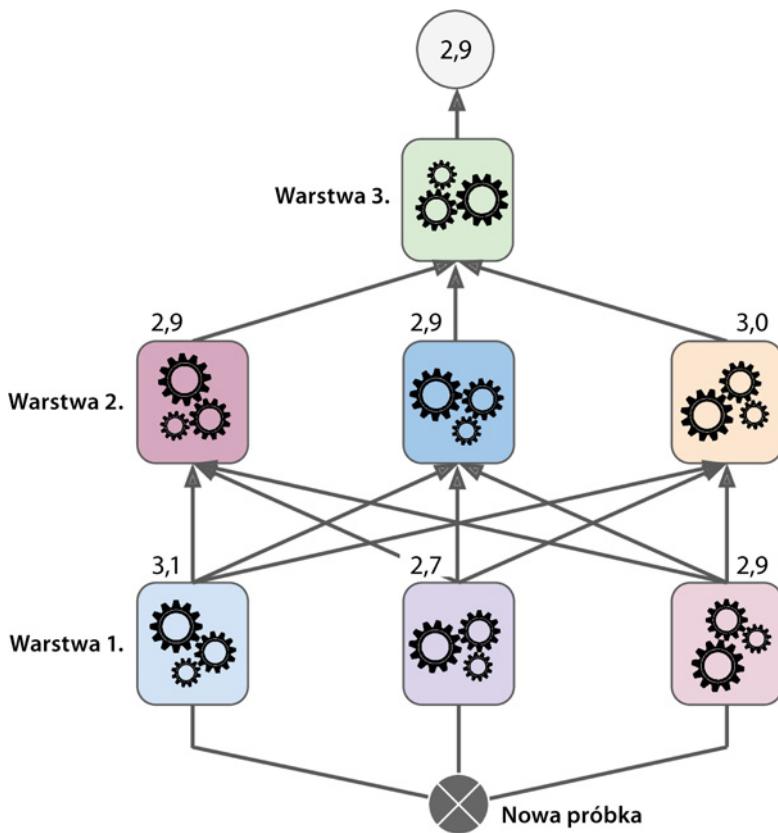
Niestety moduł Scikit-Learn nie obsługuje bezpośrednio kontaminacji, ale jej własnoręczna implementacja nie powinna nastręczać zbyt dużych problemów (zob. poniższe ćwiczenia). Ewentualnie możesz skorzystać z otwartej implementacji, takiej jak **brew** (<https://github.com/viisar/brew>).



Rysunek 7.14. Uczanie miksera

Ćwiczenia

1. Jeżeli wyuczyłaś/wyuczyłeś pięć różnych modeli wobec tego samego zbioru danych uczących i wszystkie uzyskały taką samą precyzję rzędu 95%, czy istnieje jakakolwiek szansa, że ich połączenie w zespół pozwoli uzyskać jeszcze lepsze wyniki? Jeśli tak, to w jaki sposób? Jeśli nie, to dlaczego?
2. Jaka jest różnica pomiędzy klasyfikatorami większościowym i miękkiego głosowania?
3. Czy jest możliwe przyśpieszenie uczenia zespołu agregacji poprzez rozdzielenie tego procesu pomiędzy kilka serwerów? Jak to wygląda w przypadku zespołów wklejania, wzmacniania, losowych lasów czy kontaminacji?
4. Jaka jest zaleta ewaluacji za pomocą próbek pozatreninowych?
5. Dlaczego drzewa uzyskane metodą Extra-Trees są bardziej losowe od standardowych losowych lasów? W jaki sposób ta dodatkowa losowość może nam się przysłużyć? Czy algorytm Extra-Trees jest wolniejszy od losowego lasu?



Rysunek 7.15. Prognozy uzyskiwane za pomocą wielowarstwowego zespołu kontaminacji

6. Jeżeli zespół AdaBoost ulega niedotrenowaniu, które hiperparametry i w jaki sposób należy dostroić?
7. Jeżeli zespół wzmacniania gradientowego ulega przetrenowaniu, to należy zwiększyć czy zmniejszyć wartość współczynnika uczenia?
8. Wczytaj zbiór danych MNIST (zob. rozdział 3.) i podziel go na podzbiory uczący, walidacyjny i testowy (w stosunku 50 000 : 10 000 : 10 000). Następnie wytrenuj różne klasyfikatory, np. losowego lasu, Extra-Trees czy SVM. Teraz spróbuj połączyć je w jeden zespół (korzystając z klasyfikatora głosowania większościowego lub miękkiego), który będzie osiągał lepsze wyniki wobec zestawu walidacyjnego w stosunku do pojedynczych klasyfikatorów. Po uzyskaniu takiego modelu wypróbuj go na zbiorze testowym. O ile lepiej sprawuje się ten zespół w porównaniu do poszczególnych klasyfikatorów?
9. Uruchom poszczególne klasyfikatory utworzone w poprzednim ćwiczeniu w celu wyliczenia prognoz dla zbioru walidacyjnego, a następnie utwórz nowy zbiór danych zawierający te predykcje: każdy przykład uczący ma stanowić wektor przechowujący zbiór prognoz pochodzących z pojedynczych klasyfikatorów dla obrazu, natomiast wartością docelową niech będzie klasa obrazu.

Wytrenuj klasyfikator na tym nowym zestawie danych uczących. Gratulacje! Właśnie udało Ci się wyuczyć mikser, który wraz z klasyfikatorami tworzy zespół kontaminacji. Sprawdź teraz wydajność tego zespołu wobec danych testowych. Dla każdego obrazu w zbiorze testowym wylicz prognozy za pomocą wszystkich klasyfikatorów, a następnie prześlij wyniki do miksera, aby uzyskać przewidywania zespołu. Porównaj wyniki z utworzonym wcześniej klasyfikatorem głosującym.

Rozwiązania tych ćwiczeń znajdziesz w dodatku A.

Redukcja wymiarowości

Wiele problemów uczenia maszynowego obejmuje tysiące, a nawet miliony cech opisujących każdy przykład uczący. Z tego powodu proces uczenia nie tylko przebiega bardzo powoli, ale, jak się wkrótce przekonamy, taka liczba cech utrudnia również znalezienie dobrego rozwiązania. Problem ten jest często nazywany **kławką wymiarowości** (ang. *curse of dimensionality*).

Na szczęście w przypadku rzeczywistych zadań nierzadko możliwe jest znaczne ograniczenie liczby cech, dzięki czemu nieroziwiąwalny problem nagle staje się możliwy do rozwiązania. Weźmy na przykład pod uwagę zbiór obrazów MNIST (po raz pierwszy użyty w rozdziale 3.): piksele na krańcach obrazu są niemal zawsze białe, dlatego moglibyśmy je usunąć ze zbioru danych przy minimalnej stracie informacji. Rysunek 7.6 udowadnia, że te piksele są zupełnie nieistotne z perspektywy zadania klasyfikacji. Ponadto dwa sąsiadujące piksele często są ze sobą silnie skorelowane: jeżeli połączymy je w jeden piksel (np. stanowiący średnią wartość poziomów szarości obydwu pikseli), również nie przepadnie wiele informacji.



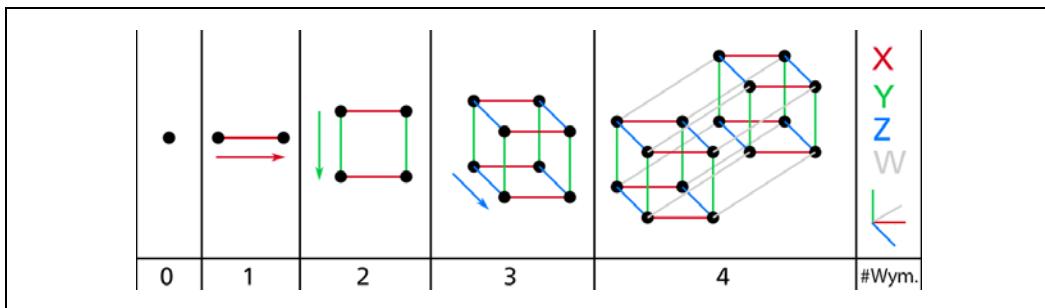
Redukcja wymiarowości mimo wszystko powoduje w pewnym stopniu utratę informacji (np. konwersja obrazu do formatu JPEG powoduje zmniejszenie jego jakości), zatem nawet jeśli przyspieszymy proces nauki, model może osiągać nieznacznie gorsze rezultaty. Poza tym potoki przyjmują nieco bardziej złożoną postać, przez co są trudniejsze do utrzymania. Dlatego przed przystąpieniem do redukowania wymiarowości należy najpierw spróbować wytrenować system za pomocą oryginalnego zbioru danych i skorzystać z tej możliwości dopiero w sytuacji, gdy nauka będzie trwała zbyt długo. W pewnych sytuacjach redukcja wymiarowości danych uczących pozwala wyeliminować pewne rodzaje szumów i nieistotne szczegóły, co może skutkować większą wydajnością, zazwyczaj jednak tak się nie dzieje i przyspieszamy jedynie proces uczenia.

Oprócz przyspieszania procesu nauki redukcja wymiarowości doskonale nadaje się również do wizualizowania danych (tzw. **DataViz**). Ograniczenie liczby wymiarów do dwóch (lub trzech) umożliwia generowanie zwartego wykresu wielowymiarowego zbioru danych nierzadko ukazującego przydatne informacje, np. wzorce skupień. Do tego technika DataViz stanowi kluczowy element w przekazywaniu wyników osobom niebędącym anietykami danych, zwłaszcza decydentom wykorzystującym Twoje rezultaty.

W niniejszym rozdziale przyjrzymy się uważniej kłatwie wymiarowości i postaramy się zrozumieć, co takiego dzieje się w wielowymiarowej przestrzeni. Następnie zajmiemy się dwiema głównymi strategiami redukcji wymiarowości (rzutowaniem i uczeniem rozmaitościowym — ang. *manifold learning*), po czym przejdziemy do trzech najpopularniejszych algorytmów: analizy PCA, jądrowej analizy PCA i LLE.

Kłatwa wymiarowości

Jesteśmy tak przyzwyczajeni do życia w trzech wymiarach¹, że nie potrafimy sobie wyobrazić wielowymiarowej przestrzeni. Mamy problem nawet z wizualizacją w głowie podstawowego czterowymiarowego hipersześcianu (rysunek 8.1), nie wspominając nawet o dwustuwympiarowej elipsoidzie zagiętej w tysiącwymiarowej przestrzeni.



Rysunek 8.1. Punkt, odcinek, kwadrat, sześcian i tesserakt (hipersześciany od zerowego do czwartego wymiaru)²

Okazuje się, że w wielowymiarowej przestrzeni wiele zjawisk wygląda inaczej niż w znanej nam rzeczywistości. Na przykład jeżeli wybierzemy losowy punkt w jednostce kwadratowej (kwadracie o rozmiarze 1×1), prawdopodobieństwo, że będzie znajdował się on w odległości mniejszej niż 0,001 od brzegu, wynosi zaledwie 0,4% (inaczej mówiąc, jest bardzo mało prawdopodobne, że taki losowy punkt będzie stanowił „ekstremum” w jakimkolwiek kierunku). Jednak w przypadku tysiącwymiarowego hipersześcianu prawdopodobieństwo to przekracza 99,999999%. Większość punktów w wielowymiarowym hipersześcianie znajduje się blisko jego brzegów³.

Nas powinna martwić inna różnica: jeżeli losowo wybierzesz dwa punkty w jednostce kwadratowej, odległość między nimi będzie średnio wynosiła 0,52. W przypadku dwóch punktów wybranych w sześcianie średnia odległość pomiędzy nimi wzrośnie do ok. 0,66. A co w przypadku dwóch losowo dobranych punktów w 1000000-wymiarowym hipersześcianie? Możesz wierzyć lub nie, ale średnia

¹ Dokładniej mówiąc, w czterowymiarowej przestrzeni, jeżeli uwzględnimy czas, a także zawierającej kilka dodatkowych wymiarów, jeśli zajmujesz się teorią strun.

² Pod adresem <https://www.youtube.com/watch?v=BVo2igbFSPE> możesz obejrzeć animację obracającego się tesseraktu rzutowanego na trójwymiarową przestrzeń. Rysunek wykonany przez użytkownika Wikipedii — NerdBoy1392 (Uznanie autorstwa — na tych samych warunkach 3.0, <https://creativecommons.org/licenses/by-sa/3.0/deed.pl>). Oryginał pochodzi ze strony <https://en.wikipedia.org/wiki/Tesseract>.

³ Ciekawostka: każda osoba, jaką znasz jest prawdopodobnie „ekstremistą” przynajmniej w jednym wymiarze (np. pod względem słodzenia kawy), jeżeli weźmiemy pod uwagę wystarczającą liczbę wymiarów.

odległość pomiędzy nimi wyniesie w przybliżeniu 408,25 (około $\sqrt{1000000 / 6}$)! Przeczy to zdrowemu rozsądkowi: jak dwa punkty mogą być od siebie tak oddalone, skoro mieszą się w tej samej jednostce hipersześciennej? W przypadku przestrzeni wielowymiarowych istnieje wystarczająco dużo miejsca. W konsekwencji wielowymiarowe zbiory danych są narażone na rozrzedzenie (rzadkość): większość przykładów uczących będzie od siebie znacznie oddalona. Oznacza to również, że nowa próbka będzie znajdowała się też daleko od przykładów uczących, przez co wyliczane prognozy będą znacznie mniej pewne niż uzyskane w przestrzeni o mniejszej liczbie wymiarów, ponieważ będą one bazować na znacznie większych ekstrapolacjach. Krótko mówiąc, wraz ze wzrostem liczby wymiarów zwiększa się ryzyko przetrenowania modelu.

W teorii jednym z rozwiązań kłatywy wymiarowości mogłyby być zwiększenie zbioru danych uczących w celu osiągnięcia wystarczającej gęstości przestrzeni danych. Niestety w praktyce oznacza to, że liczba przykładów wymaganych do osiągnięcia zakładanej gęstości wzrasta wykładniczo wraz z liczbą wymiarów. Przy 100 wymiarach (znacznie mniej, niż znajduje się w zbiorze danych MNIST) w celu wyuczenia próbek znajdujących się średnio w odległości 0,1 od siebie (przy założeniu, że są one równomiernie rozmieszczone we wszystkich wymiarach) wymagana byłaby liczba przykładów przekraczająca liczbę atomów znajdujących się w obserwowlanym Wszechświecie⁴.

Główne strategie redukcji wymiarowości

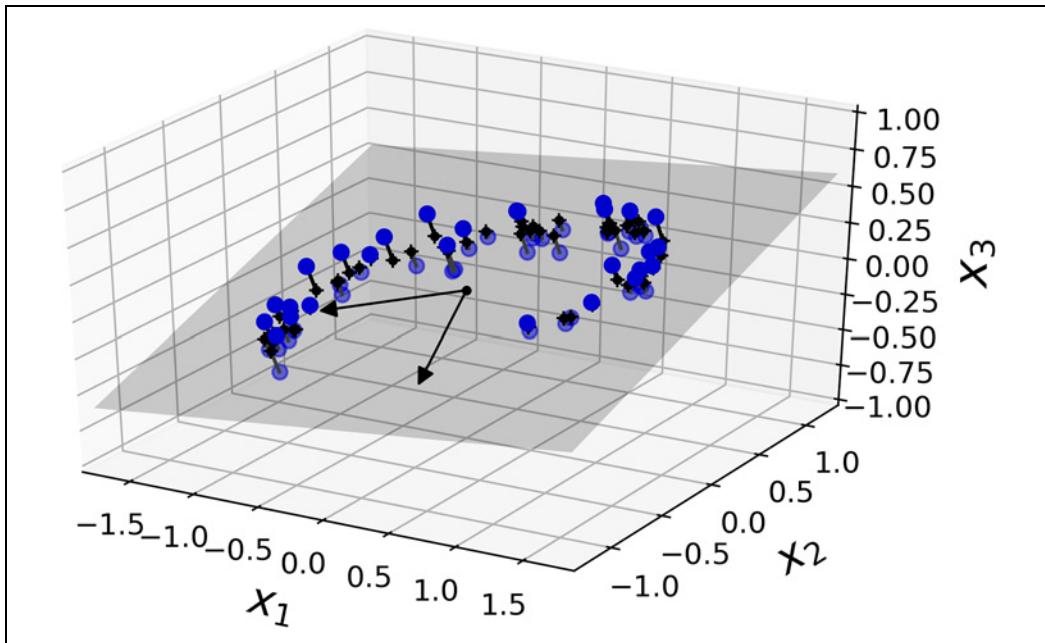
Zanim zajmiemy się poszczególnymi algorytmami redukcji wymiarowości, zastanówmy się nad dwiema głównymi strategiami jej zmniejszania: rzutowaniem i uczeniem rozmaitościowym.

Rzutowanie

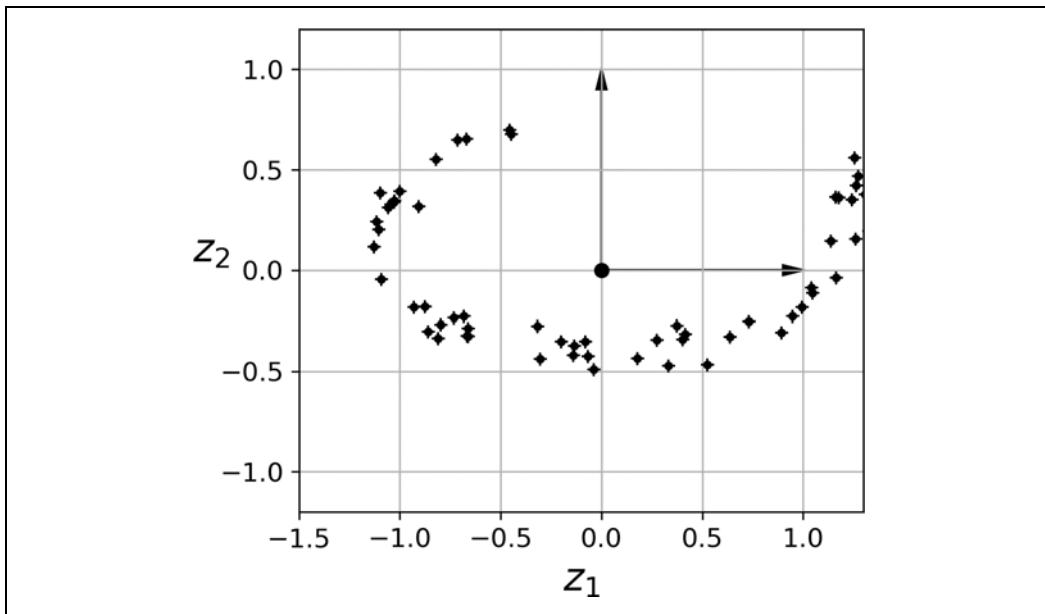
W większości praktycznych problemów próbki uczące *nie* są równomiernie rozmieszczone we wszystkich wymiarach. Wiele cech jest niemal stałych, natomiast inne są ze sobą silnie skorelowane (jak wiemy z dyskusji na temat zbioru danych MNIST). W rezultacie wszystkie przykłady uczące znajdują się wewnętrz (lub w pobliżu) **podprzestrzeni** składającej się z mniejszej liczby wymiarów od pierwotnej przestrzeni cech. Brzmi to bardzo abstrakcyjnie, dlatego pomożemy sobie przykładem. Na rysunku 8.2 widzimy trójwymiarowy zbiór danych reprezentowany przez kółka.

Zauważ, że wszystkie przykłady uczące znajdują się blisko jednej płaszczyzny: jest to dwuwymiarowa podprzestrzeń przestrzeni trójwymiarowej. Jeśli będziemy rzutować każdy przykład uczący prostopadle na tę podprzestrzeń (co zostało ukazane w postaci krótkich odcinków łączących punkty z płaszczyzną), otrzymamy nowy, dwuwymiarowy zestaw danych zaprezentowany na rysunku 8.3. Ta-da! Właśnie zredukowaliśmy wymiarowość z trzech do dwóch wymiarów! Zwróć uwagę, że osie reprezentują nowe cechy z_1 i z_2 (współrzędne rzutowania na płaszczyznę).

⁴ Szacuje się, że w obserwowlanym Wszechświecie znajduje się ok. 10^{80} samych atomów wodoru — *przyp. tłum.*

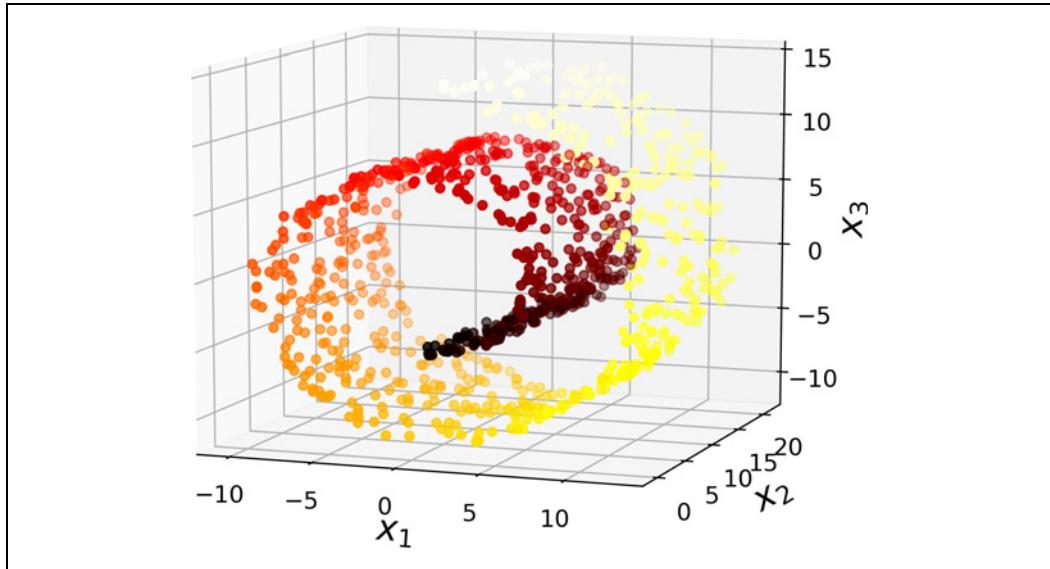


Rysunek 8.2. Trójwymiarowy zbiór danych znajdujący się blisko dwuwymiarowej podprzestrzeni



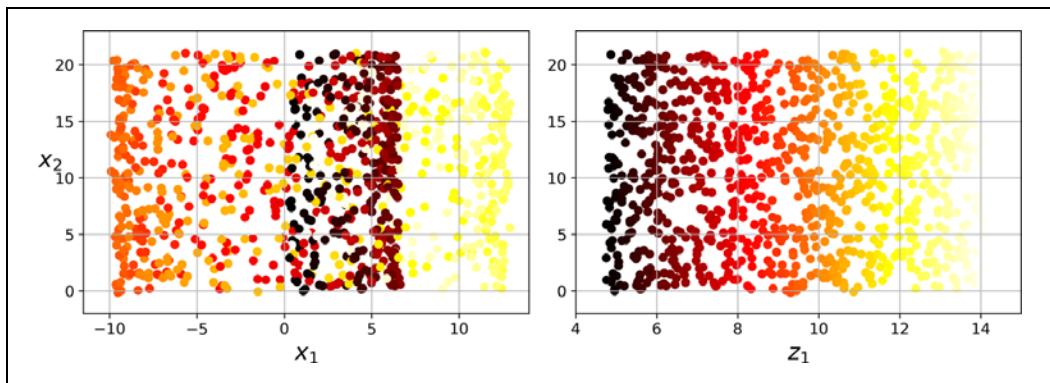
Rysunek 8.3. Nowy, dwuwymiarowy zbiór danych uzyskany w wyniku rzutowania

Jednak rzutowanie nie zawsze stanowi dobry sposób redukcji wymiarowości. W wielu przypadkach podprzestrzeń może być zawinięta i poskręcana, tak jak w słynnym zbiorze danych **Swiss roll** ukazanym na rysunku 8.4.



Rysunek 8.4. Zbiór danych Swiss roll

Zwykłe rzutowanie na płaszczyznę (np. poprzez usunięcie cechy x_3) spowoduje wymieszanie różnych warstw zbioru danych, co możemy zobaczyć na lewym wykresie rysunku 8.5. W rzeczywistości natomiast chcemy tak rozwinąć ten zbiór danych, aby uzyskać dwuwymiarowy wykres zaprezentowany po prawej stronie rysunku 8.5.



Rysunek 8.5. „Wtłoczenie” danych poprzez rzutowanie na płaszczyznę (lewy wykres) i rozwinięcie „roladki” (prawy wykres)

Uczenie rozmaitościowe

Zbiór danych Swiss roll stanowi przykład dwuwymiarowej **rozmaitości** (ang. *manifold*). Krótko mówiąc, rozmaitością dwuwymiarową nazywamy dwuwymiarowy kształt, który można wygiąć i wykręcać w wielowymiarowej przestrzeni. W bardziej ogólnym ujęciu d -wymiarowa rozmaistość stanowi

część n -wymiarowej przestrzeni (gdzie $d < n$), która lokalnie przypomina d -wymiarową hiperpłaszczyznę. W przypadku zestawu danych Swiss roll $d = 2$ i $n = 3$; lokalnie przypomina on dwuwymiarową płaszczyznę, ale zostaje zwinięty w trzech wymiarach.

Wiele algorytmów redukcji wymiarowości działa poprzez modelowanie rozmaitości, na której znajdują się próbki uczące; jest to proces **uczenia rozmaitościowego** (ang. *manifold learning*). Wykorzystywane jest w nim **założenie rozmaitości** (ang. *manifold assumption*), zwane także **hipotezą rozmaitości** (ang. *manifold hypothesis*), zgodnie z którą większość rzeczywistych, wielowymiarowych zbiorów danych znajduje się blisko rozmaitości składającej się ze znacznie mniejszej liczby wymiarów. Założenie to jest bardzo często obserwowane empirycznie.

Niech za przykład nam posłuży po raz kolejny zbiór danych MNIST: wszystkie obrazy odręcznie pisanych cyfr mają jakieś cechy wspólne. Cyfry te składają się z połączonych odcinków, obramowania są koloru białego, a cyfry są mniej więcej wyśrodkowane. W przypadku losowo wygenerowanych obrazów jedynie niezmiernie mały ich odsetek przypominałby odręcznie napisane znaki. Innymi słowy, istnieje znacznie mniejsza liczba stopni swobody pozwalających na utworzenie obrazu cyfry niż w przypadku dowolnego losowo wygenerowanego obrazu. Ograniczenia te najczęściej pozwalają umieścić zbiór danych w rozmaitości o mniejszej liczbie wymiarów.

Hipotezie rozmaitości towarzyszy często inne niejawne założenie: wykonywanie zadanie (np. klasyfikacja albo regresja) będzie prostsze, jeżeli zostanie wyrażone w podprzestrzeni rozmaitości. Przykładowo, w górnym rzędzie rysunku 8.6 zestaw danych Swiss roll został rozdzielony na dwie klasy: w przestrzeni trójwymiarowej (po lewej) granica decyzyjna przedstawia się dość skomplikowanie, jednak w dwuwymiarowej przestrzeni rozmaitości (po prawej) stanowi ona linię prostą.

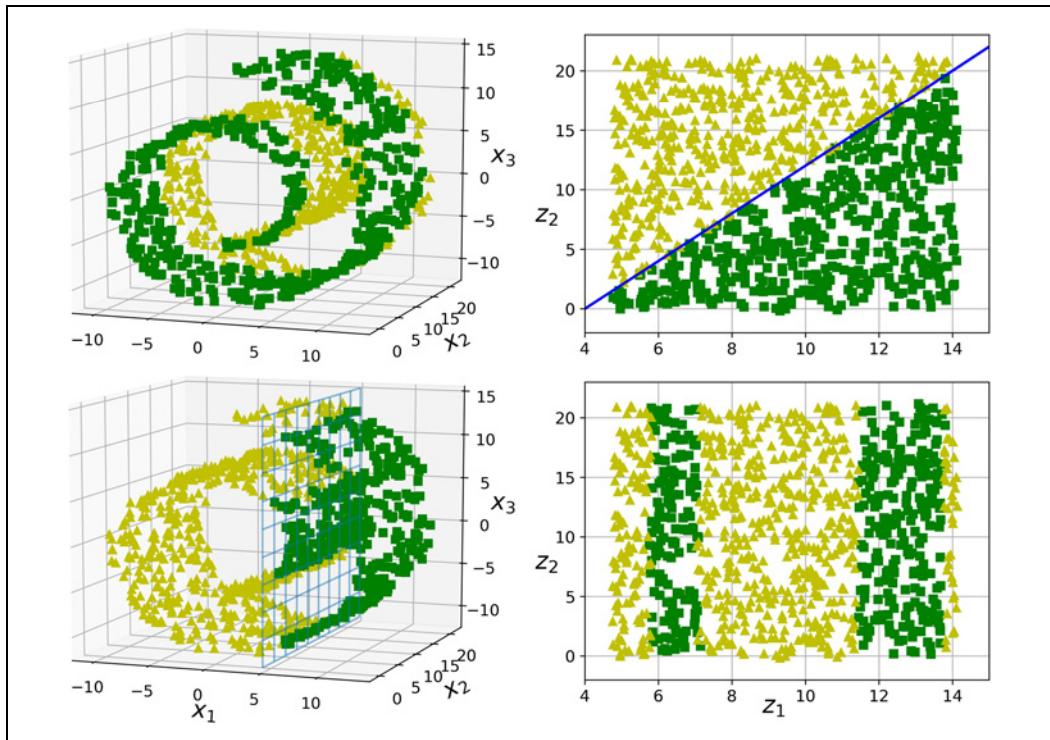
Jednak to niejawne założenie nie zawsze jest prawdziwe. Na przykład w dolnym rzędzie rysunku 8.6 granica decyzyjna mieści się w punkcie $x_1 = 5$. W oryginalnej, trójwymiarowej przestrzeni (pionowa płaszczyzna) ta granica decyzyjna wygląda bardzo prosto, jednak na rozwiniętej rozmaitości okazuje się znacznie bardziej złożona (cztery oddzielne obszary).

Krótko mówiąc, zredukowanie wymiarowości zbioru danych uczących przed wytrenowaniem modelu zazwyczaj powoduje przyspieszenie procesu uczenia, nie zawsze jednak prowadzi do lepszego lub prostszego rozwiązania — wszystko zależy od zbioru danych.

Mam nadzieję, że już dobrze rozumiesz koncepcję klatwy wymiarowości oraz sposobów jej ograniczania, zwłaszcza w sytuacjach, gdy założenie rozmaitości okazuje się prawdziwe. W dalszej części rozdziału zajmiemy się najpopularniejszymi algorytmami służącymi do redukowania wymiarowości.

Analiza PCA

Analiza głównych składowych (ang. *Principal Component Analysis* — PCA) stanowi obecnie najpopularniejszy algorytm redukcji wymiarowości. Najpierw określa on hiperpłaszczyznyznę znajdującą się najbliżej danych, po czym są one na nią rzutowane, zupełnie jak na rysunku 8.2.



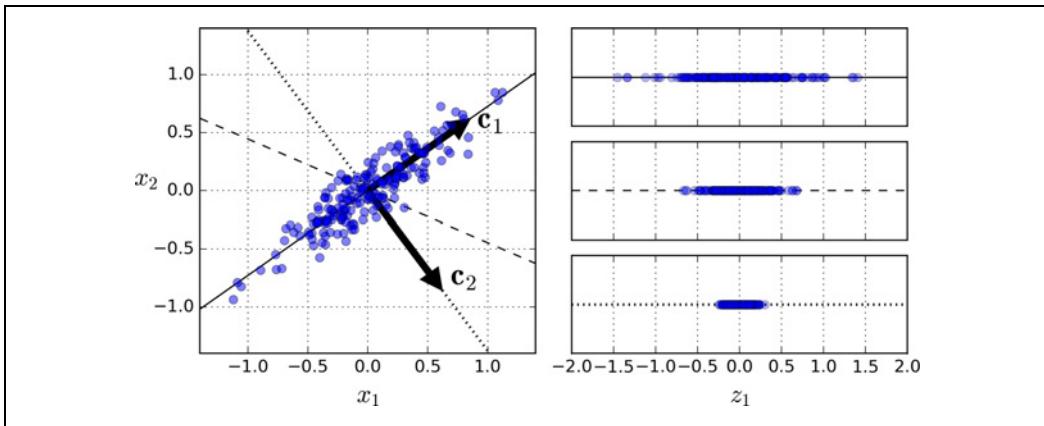
Rysunek 8.6. Granica decyzyjna nie zawsze musi ulec uproszczeniu po zredukowaniu wymiarowości

Zachowanie wariancji

Zanim będziemy mogli rzutować zbiór danych uczących na hiperpłaszczyznę o mniejszej liczbie wymiarów, musimy najpierw ją wybrać. Na przykład na lewym wykresie rysunku 8.7 widzimy prosty zestaw danych wraz z ukazanymi trzema różnymi osiami (tzn. jednowymiarowymi hiperpłaszczyznami). Po prawej stronie widzimy rezultaty rzutowania zbioru danych na każdą z tych osi. Jak łatwo zauważycy, rzutowanie na hiperpłaszczyznę reprezentowaną przez linię ciągłą pozwala zachować maksymalną wariancję, z kolei hiperpłaszczyzna symbolizowana linią kropkowaną zachowuje bardzo małą wariancję, a hiperpłaszczyzna wskazywana przez linię kreskowaną zachowuje pośrednią wartość wariancji.

Najrozsądzniejszy wydaje się wybór osi zachowującej największą wariancję, gdyż najprawdopodobniej straci ona najmniej informacji w stosunku do pozostałych rzutów. Możemy jeszcze inaczej wyjaśnić ten wybór: oś ta minimalizuje odległość średniokwadratową pomiędzy pierwotnym zbiorem danych a jego rzutem na tę oś. To właśnie ta prosta koncepcja kryjąca się za analizą PCA⁵.

⁵ Karl Pearson, *On Lines and Planes of Closest Fit to Systems of Points in Space*, „The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science” 2, no. 11 (1901), s. 559 – 572, <https://www.tandfonline.com/doi/10.1080/14786440109462720>.



Rysunek 8.7. Wybór podprzestrzeni rzutowania

Główne składowe

Analiza PCA służy do wyznaczania osi zachowującej największą wartość wariancji zbioru uczącego. Na rysunku 8.7 oś ta jest oznaczona linią ciągłą. Zostaje również znaleziona druga oś, prostopadła do pierwszej, która zachowuje największą wartość pozostałej wariancji. W naszym dwuwymiarowym przykładzie nie ma zbyt dużego wyboru: ta prostopadła oś jest oznaczona linią kropkowaną. W przypadku bardziej wielowymiarowego zbioru danych algorytm PCA znajdzie również trzecią oś, ortogonalną do dwóch wcześniejszych, następnie czwartą, piątą itd. — liczba osi jest taka sama, jak liczba wymiarów zbioru danych.

Oś i-ta nosi nazwę **i-tej głównej składowej** (ang. *principal component* — PC) danych. Na rysunku 8.7 pierwszą główną składową jest oś, na której znajduje się wektor c_1 , natomiast oś, na której mieści się wektor c_2 , stanowi drugą główną składową. Dwie pierwsze główne składowe są prostopadłymi osiami, na których znajdują się strzałki, natomiast trzecia główna składowa to oś umieszczona ortogonalnie do tej płaszczyzny.



Dla każdej głównej składowej algorytm PCA wyszukuje wychodzący ze środka układu współrzędnych wektor jednostkowy wyznaczający kierunek tej składowej. Dwa przeciwnie wektory jednostkowe znajdują się na jednej osi, dlatego kierunek wektorów jednostkowych zwracanych przez analizę PCA nie jest stabilny: jeśli trochę zmodyfikujesz zbiór danych uczących i ponownie wykonasz analizę PCA, otrzymane wektory jednostkowe mogą być skierowane w przeciwną stronę niż za pierwszym razem. Zazwyczaj jednak znajdują się one na tych samych osiach. W pewnych przypadkach para wektorów jednostkowych może się nawet obracać albo zamieniać miejscami (jeżeli wariancje wyznaczane przez obydwie osie są do siebie zbliżone).

⁶ Karl Pearson, *On Lines and Planes of Closest Fit to Systems of Points in Space*, „The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science” 2, no. 11 (1901), s. 559 – 572, <https://www.tandfonline.com/doi/pdf/10.1080/14786440109462720>.

Jak więc możemy wyszukiwać główne składowe zbioru uczącego? Na szczęście istnieje standardowa technika faktoryzacji macierzy, zwana **rozkładem według wartości osobliwych** (ang. *singular value decomposition* — SVD), która rozkłada macierz zestawu danych uczących X na iloczyn macierzowy trzech macierzy: $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, gdzie V zawiera wektory jednostkowe definiujące wszystkie interesujące nas główne składowe, co zostało ukazane w równaniu 8.1.

Równanie 8.1. Macierz głównych składowych

$$\mathbf{V} = \begin{pmatrix} | & | & | \\ c_1 & c_2 & \cdots & c_n \\ | & | & & | \end{pmatrix}$$

Poniższy kod wykorzystuje funkcję `svd()` modułu NumPy w celu uzyskania wszystkich głównych składowych zbioru uczącego, a następnie wydobywa dwa wektory jednostkowe definiujące dwie pierwsze składowe:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```



Algorytm PCA przyjmuje, że zbiór danych został wyśrodkowany wobec początku układu współrzędnych. Jak się przekonamy, implementacje dostępne w module Scikit-Learn automatycznie wyśrodkowują dane. Jeśli samodzielnie tworzysz algorytm PCA (jak w powyższym przykładzie) lub korzystasz z innych bibliotek, nie zapomnij o wyśrodkowaniu zbioru danych.

Rzutowanie na d wymiarów

Po zidentyfikowaniu wszystkich głównych składowych możemy zredukować wymiarowość zbioru danych do d wymiarów poprzez rzutowanie przykładów na hiperpłaszczyznę zdefiniowaną przez d pierwszych głównych składowych. Dzięki wybraniu tej hiperpłaszczyzny rzutowany zbiór danych zachowa największą możliwą wariancję oryginalnego zestawu danych. Na przykład na rysunku 8.2 trójwymiarowy zbiór danych jest rzutowany na dwuwymiarową płaszczyznę zdefiniowaną przez dwie pierwsze główne składowe, co pozwala zachować dużą część wariancji. W rezultacie dwuwymiarowy rzut bardzo przypomina oryginalny, trójwymiarowy zbiór danych.

Aby rzutować zbiór danych uczących na hiperpłaszczyznę i uzyskać zredukowany zestaw danych X_d -rzut o wymiarowości d , wystarczy obliczyć iloczyn macierzowy macierzy danych uczących X przez macierz \mathbf{W}_d zdefiniowaną jako zawierającą d pierwszych kolumn macierzy V (równanie 8.2).

Równanie 8.2. Rzutowanie zestawu danych uczących na d wymiarów

$$\mathbf{X}_{d\text{-rzut.}} = \mathbf{X}\mathbf{W}_d$$

Z pomocą poniższego kodu możemy rzutować zbiór danych uczących na płaszczyznę zdefiniowaną przez pierwsze dwie główne składowe:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

No proszę! Potrafisz teraz redukować wymiarowość każdego zbioru danych do dowolnej liczby wymiarów przy jednoczesnym zachowaniu jak największej wariancji.

Implementacja w module Scikit-Learn

Klasa PCA modułu Scikit-Learn wykorzystuje rozkład SVD do implementacji analizy PCA w podobny sposób, jak my dokonaliśmy tego wcześniej. Poniższy fragment kodu redukuje wymiarowość zbioru danych do dwóch wymiarów (i automatycznie wyśrodkowuje dane):

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

Po dopasowaniu transformatora do danych uczących atrybut `components_` przechowuje macierz transponowaną W_d (tzn. wektor jednostkowy definiujący pierwszą główną składową jako `pca.components_.T[:,0]`).

Współczynnik wariancji wyjaśnionej

Użyteczne informacje dostarcza również **współczynnik wariancji wyjaśnionej** (ang. *explained variance ratio*) każdej głównej składowej; jest on dostępny poprzez zmienną `explained_variance_ratio_`. Określa on proporcję pomiędzy wariancjami wyznaczanymi przez każdą główną składową. Przyjrzymy się współczynnikiem wariancji wyjaśnionej dla dwóch pierwszych składowych trójwymiarowego zbioru danych z rysunku 8.2:

```
>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

Dowiadujemy się z tego wyniku, że 84,2% wariancji zbioru danych zostało zachowane wzduż pierwszej głównej składowej i 14,2% wzduż drugiej składowej. Pozostaje mniej niż 1,2% wzduż trzeciej składowej, co pozwala przypuszczać, że prawdopodobnie nie zawiera ona wielu informacji.

Wybór właściwej liczby wymiarów

Zamiast dowolnie wybierać liczbę docelowych wymiarów podprzestrzeni, łatwiej jest wyznaczyć liczbę wymiarów pozwalającą zachować wystarczająco wysoką wartość wariancji (np. 95%). Wyjątkiem jest oczywiście redukcja wymiarowości w celu wizualizacji danych — w takim przypadku chcemy ograniczyć liczbę wymiarów do dwóch lub trzech.

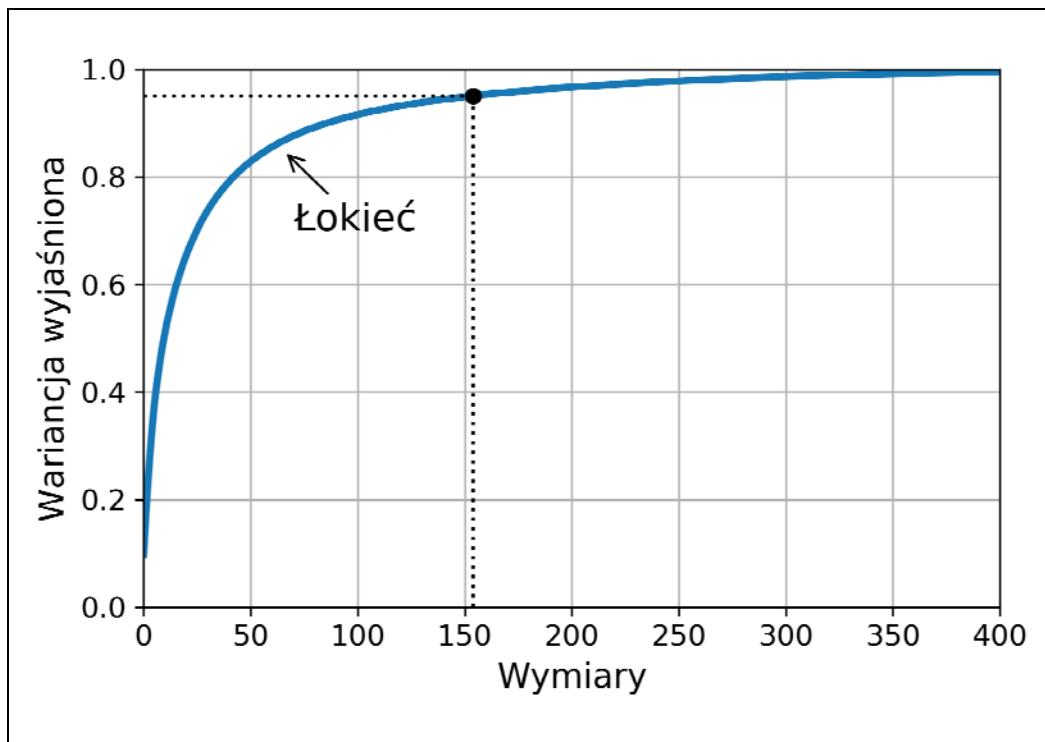
Poniższy kod przeprowadza analizę PCA bez redukowania wymiarowości, a następnie określa minimalną liczbę wymiarów potrzebnych do zachowania 95% wariancji zbioru danych testowych:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

Wystarczyłoby teraz wyznaczyć parametr `n_components=d` i ponownie uruchomić algorytm PCA. Istnieje jednak znacznie lepsze rozwiązanie: zamiast wyznaczać liczbę głównych składowych, które chcemy wykorzystać, możesz wyznaczyć wartość parametru `n_components` jako liczbę zmiennoprzecinkową w zakresie pomiędzy 0.0 a 1.0, oznaczającą odsetek wariancji, jaki chcesz zachować:

```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X_train)
```

Kolejnym sposobem jest wygenerowanie wykresu wariancji wyjaśnionej w funkcji liczby wymiarów (zwyczajny wykres `cumsum`) (rysunek 8.8). Zazwyczaj na wykresie będzie występował łokieć (zagięcie), czyli punkt, w którym wariancja wyjaśniona przestanie szybko rosnąć. W tym przypadku widzimy, że ograniczenie liczby wymiarów do około 100 nie spowoduje znacznej utraty wariancji.



Rysunek 8.8. Wariancja wyjaśniona jako funkcja liczby wymiarów

Algorytm PCA w zastosowaniach kompresji

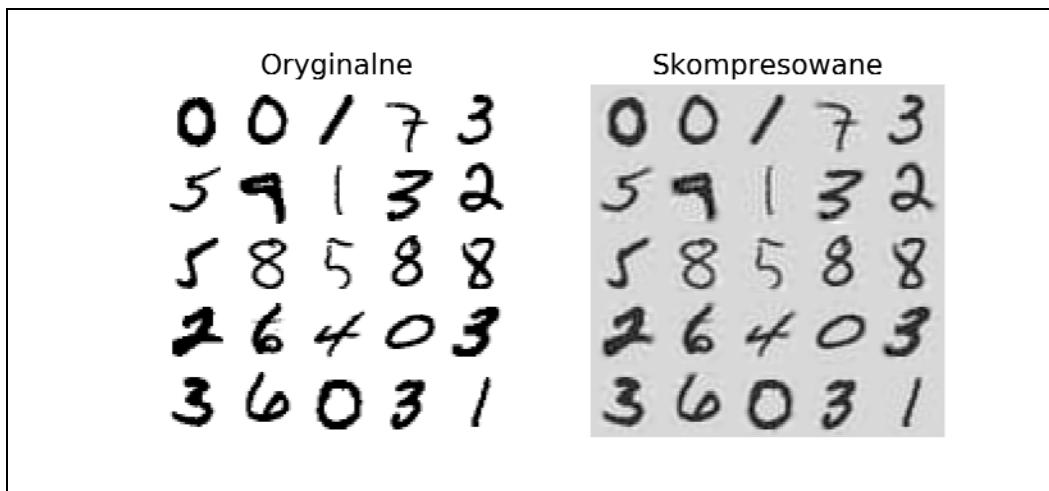
Po redukcji wymiarowości zbiór danych uczących zajmuje znacznie mniej miejsca. Spróbuj, na przykład, zastosować analizę PCA wobec zbioru danych MNIST przy zachowaniu 95% wariancji. Okaże się, że każda próbka zawiera teraz zaledwie nieco ponad 150 cech zamiast pierwotnych 784. Zatem ponad pięciokrotnie zmniejszamy rozmiar zbioru przy zachowaniu większości wariancji! Jest to rozsądny współczynnik kompresji i łatwo zrozumieć, dlaczego takie zmniejszenie rozmiaru może znacznie przyspieszyć działanie algorytmu klasyfikacji (np. maszyny SVM).

Możliwe jest również przywrócenie zbioru danych do 784 wymiarów poprzez odwrotną transformację rzutowania PCA. Nie odzyskamy pierwotnych danych, gdyż w trakcie rzutowania utraciliśmy niewielką część informacji (pominięte 5% wariancji), jednak będą one w wystarczającym stopniu przypominać pierwotny zbiór danych. Odległość średniokwadratowa dzieląca pierwotne dane od zrekonstruowanych (skompresowanych i odtworzonych) jest nazywana **błędem rekonstrukcji** (ang. *reconstruction error*).

Za pomocą poniższego kodu kompresujemy zbiór danych MNIST do 154 wymiarów, a następnie przy użyciu metody `inverse_transform()` przywracamy go do pierwotnych 784 wymiarów:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

Rysunek 8.9 przedstawia kilka cyfr z pierwotnego zestawu uczącego (po lewej) oraz te same cyfry po kompresji i rekonstrukcji (po prawej). Jak widać, obraz ma nieco gorszą jakość, ale cyfry są w większości wciąż rozpoznawalne.



Rysunek 8.9. Kompresja MNIST zachowująca 95% wariancji

Wzór na odwrotną transformację został zaprezentowany w równaniu 8.3.

Równanie 8.3. Odwrotna transformacja PCA przywracająca pierwotną liczbę wymiarów

$$\mathbf{X}_{\text{zrekonstruowany}} = \mathbf{X}_{d-\text{wym}} \mathbf{W}_d^T$$

Losowa analiza PCA

Jeżeli w hiperparametrze `svd_solver` wyznaczysz wartość "randomized", to moduł Scikit-Learn wykorzysta algorytm stochastyczny o nazwie **losowa analiza PCA** (ang. *randomized PCA*), który szybko wyszukuje przybliżenie d pierwszych głównych składowych. Jego złożoność obliczeniowa wynosi $O(m \times d^2) + O(d^3)$, co stanowi różnicę w stosunku do złożoności obliczeniowej $O(m \times n^2) + O(n^3)$

cechującej pełną technikę SVD, dlatego okazuje się drastycznie szybszy od algorytmu SVD, gdy d jest znacznie mniejsze od n :

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Domyślnie hiperparametr `svd_solver` ma wyznaczoną wartość "auto": moduł Scikit-Learn automatycznie wykorzystuje algorytm losowej analizy PCA, jeżeli m lub n są większe od 500, a d stanowi mniej niż 80% wartości m lub n ; w przeciwnym razie zostaje użyty pełny algorytm SVD. Jeżeli chcesz wymusić stosowanie pełnego algorytmu SVD, wyróżnij w hiperparametrze `svd_solver` wartość "full".

Przyrostowa analiza PCA

Jednym z problemów powyższych implementacji analizy PCA jest konieczność umieszczenia całego zbioru danych w pamięci po to, aby algorytm mógł działać. Na szczęście mamy do dyspozycji również **przyrostowe algorytmy PCA** (ang. *incremental PCA* — IPCA). Umożliwiają one dzielenie zbioru danych uczących na minigrupy i pojedynczo przekazywanie ich algorytmowi IPCA. Rozwiązanie to przydaje się w przypadku dużych zbiorów danych i do przetwarzania nowych przykładów w miarę ich pojawiania się.

Poniższy kod rozdziela zestaw danych MNIST na 100 minigrup (przy użyciu funkcji `array_split()` modułu NumPy) i przekazuje je klasie `IncrementalPCA`⁷ w celu zmniejszenia jego wymiarowości do 154 wymiarów (podobnie jak uprzednio). Zauważ, że dla każdej minigrupy wywołujemy metodę `partial_fit()`, a nie `fit()`, jak w przypadku pełnego zbioru uczącego:

```
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)
```

Ewentualnie możesz skorzystać z klasy `memmap` modułu NumPy, która umożliwia manipulowanie dużymi tablicami przechowywanymi w postaci pliku binarnego na dysku w taki sposób, jakby były wczytane w całości do pamięci; do pamięci są wczytywane wyłącznie te dane, które są w danej chwili potrzebne. Klasa `IncrementalPCA` wykorzystuje w każdym momencie tylko niewielki fragment tablicy, dlatego nie grozi nam przepełnienie pamięci. Dzięki temu możemy wywołać tradycyjną metodę `fit()`, o czym możemy się przekonać, patrząc na następujący kod:

```
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)
```

⁷ Moduł Scikit-Learn wykorzystuje algorytm opisany w: David A. Ross i in., *Incremental Learning for Robust Visual Tracking*, „International Journal of Computer Vision” 77, no. 1 – 3 (2008), s. 125 – 141, https://www.cs.toronto.edu/~dross/ivt/RossLimLinYang_ijcv.pdf.

Jądrowa analiza PCA

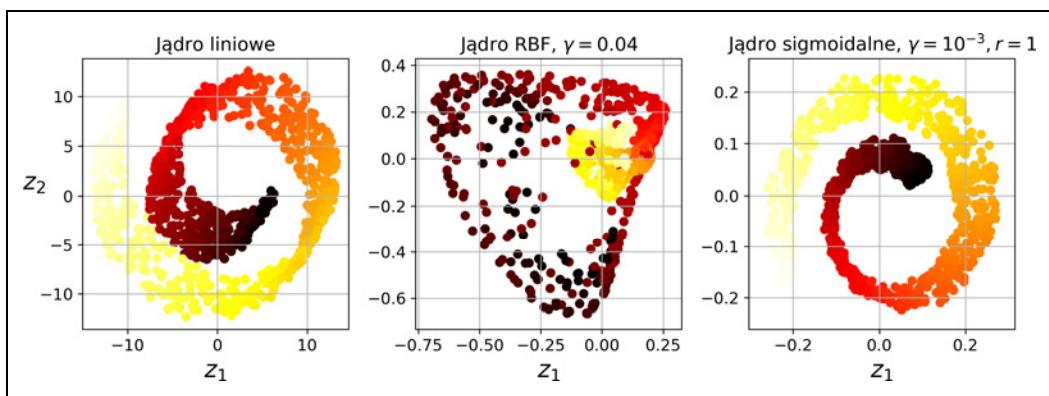
W rozdziale 5. zajmowaliśmy się sztuczką z jądrem, techniką matematyczną w sposób niejawnny rzucającą próbkę na wielowymiarową przestrzeń (zwaną **przestrzenią cech**), dzięki czemu zyskaliśmy możliwość nieliniowej klasyfikacji i regresji przy użyciu maszyn wektorów nośnych. Przypominam, że liniowa granica decyzyjna w wielowymiarowej przestrzeni cech odpowiada skomplikowanej, nieliniowej granicy decyzyjnej w **oryginalnej (pierwotnej) przestrzeni**.

Okazuje się, że możemy użyć tej samej sztuczki wraz z algorytmem PCA, przez co jesteśmy w stanie przeprowadzać złożone, nieliniowe rzutowania w celu redukcji wymiarowości. Jest to tak zwana **jądrowa analiza PCA**⁸ (ang. *kernel PCA* — kPCA). Często przydaje się ona do zachowywania skupień przykładów po przeprowadzeniu rzutowania, a czasami nawet do rozwijania zbiorów danych znajdujących się blisko skręconej rozmaitości.

Na przykład poniższy kod wykorzystuje klasę KernelPCA do przeprowadzenia jądrowej analizy PCA przy użyciu jądra RBF (w rozdziale 5. znajdziesz więcej informacji na temat jądra RBF i innych):

```
from sklearn.decomposition import KernelPCA  
  
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)  
X_reduced = rbf_pca.fit_transform(X)
```

Rysunek 8.10 przedstawia zestaw danych Swiss roll zredukowany do dwóch wymiarów za pomocą jądra liniowego (odpowiednika standardowej klasy PCA), jądra RBF i jądra sigmoidalnego.



Rysunek 8.10. Zbiór danych Swiss roll zredukowany do dwóch wymiarów za pomocą różnych algorytmów jądrowej analizy PCA

Wybór jądra i strojenie hiperparametrów

Jądrowa analiza PCA jest nienadzorowanym algorytmem uczenia, dlatego nie istnieje bezpośredni sposób pomiaru wydajności pomagający wybrać najlepsze jądro i wartości hiperparametrów. Jednak redukcja wymiarowości często stanowi etap przygotowawczy w zadaniach uczenia nadzorowanego

⁸ Bernhard Schölkopf i in., *Kernel Principal Component Analysis*, w: „Lecture Notes in Computer Science” 1327 (Berlin: Springer, 1997), s. 583 – 588, <https://link.springer.com/chapter/10.1007/BFb0020217>.

(np. klasyfikacji), dlatego możemy wykorzystać przeszukiwanie siatki w celu dobrania jądra i hiperparametrów pozwalających uzyskać najlepsze wyniki.

Poniższy kod służy do utworzenia dwuetapowego potoku. Najpierw redukujemy wymiarowość do dwóch wymiarów za pomocą jądrowej analizy PCA i wprowadzamy regresję logistyczną zajmującą się klasyfikacją próbek. Następnie korzystamy z klasy GridSearchCV, za pomocą której wyszukujemy najlepsze jądro i wartość parametru gamma, dzięki czemu na końcu potoku uzyskujemy maksymalną dostępną dokładność prognoz:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
])

param_grid = [
    {"kpca_gamma": np.linspace(0.03, 0.05, 10),
     "kpca_kernel": ["rbf", "sigmoid"]}
]

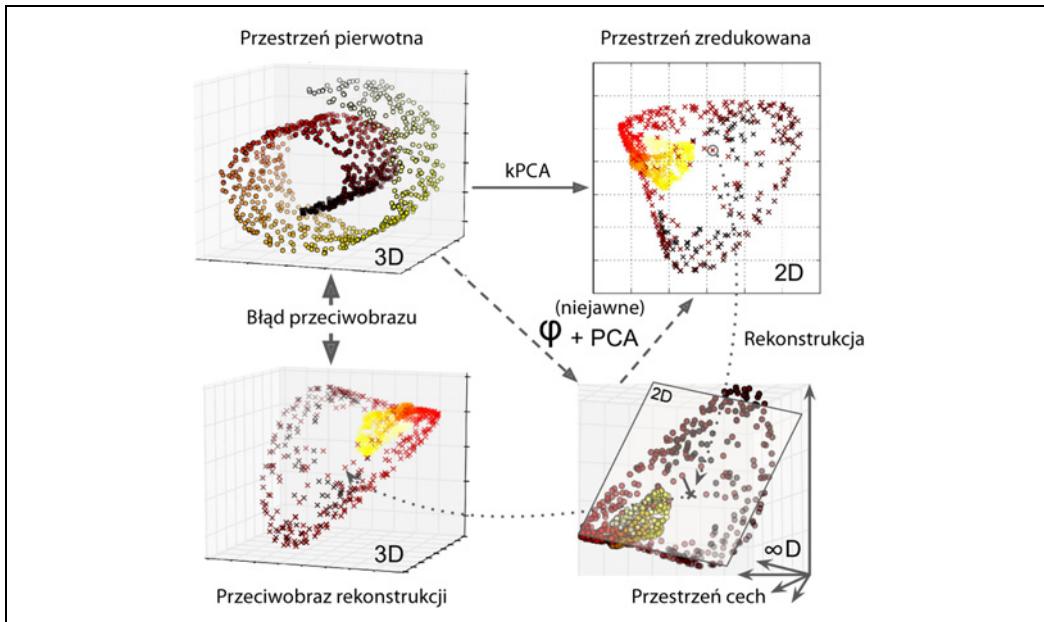
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Najlepsze jądro i hiperparametry są umieszczone w zmiennej best_params_:

```
>>> print(grid_search.best_params_)
{'kpca_gamma': 0.04333333333333335, 'kpca_kernel': 'rbf'}
```

Innym rozwiążaniem, tym razem zupełnie nienadzorowanym, jest wybór jądra i parametrów cechujących się najmniejszym błędem rekonstrukcji. Zwróć uwagę, że rekonstrukcja w przypadku liniowej analizy PCA nie jest łatwym zadaniem. Już wyjaśniam dlaczego. Rysunek 8.11 przedstawia oryginalny zbiór danych Swiss roll (na górze po lewej) oraz wynikowy dwuwymiarowy zbiór danych po przekształceniu za pomocą algorytmu kPCA wykorzystującego jądro RBF (na górze po prawej). Dzięki sztuczce z jądem przekształcenie to jest matematycznie równoważne wykorzystaniu **mapy cech** ϕ do mapowania danych uczących na przestrzeń cech o nieskończonej liczbie wymiarów (na dole po prawej), a następnie rzutowaniu przekształconego zestawu danych na przestrzeń dwuwymiarową za pomocą liniowej analizy PCA.

Zwróć uwagę, że gdybyśmy mogli przeprowadzić odwrotną analizę PCA wobec danego przykładu w zredukowanej przestrzeni, zrekonstruowany punkt znalazłby się w przestrzeni cech, a nie w przestrzeni pierwotnej (np. jak reprezentowana przez symbol X na diagramie). Przestrzeń cech ma nieskończoną liczbę wymiarów, dlatego nie jesteśmy w stanie wyliczyć odtworzonego punktu, a zatem również rzeczywistego błędu rekonstrukcji. Na szczęście możliwe jest znalezienie punktu w oryginalnej przestrzeni, który byłby rzutowany wystarczająco blisko zrekonstruowanego punktu. Punkt ten jest nazywany **przeciwobrazem** (ang. *pre-image*) rekonstrukcji. Po wyliczeniu przeciwoobrazu możemy zmierzyć kwadrat jego odległości do pierwotnej próbki. Możemy wybrać jądro i wartości hiperparametrów minimalizujące błąd tego przeciwoobrazu.



Rysunek 8.11. Jądrowa analiza PCA i błąd przeciwbrazu rekonstrukcji

Zastanawiasz się teraz pewnie, jak przeprowadzić taką rekonstrukcję. Jednym z rozwiązań jest wyuczenie nadzorowanego modelu regresji, gdzie zbiór uczący będą stanowiły rzutowane przykłady, a pierwotny zbiór pełniłby rolę wartości docelowych. Moduł Scikit-Learn przeprowadzi ten proces automatycznie, jeśli wyznaczymy parametr `fit_inverse_transform=True`, tak jak w poniższym fragmencie⁹:

```
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)
```



Domyślnie parametr `fit_inverse_transform` ma wartość `False` i klasa `KernelPCA` nie zawiera metody `inverse_transform()`. Metoda ta zostaje utworzona wyłącznie wtedy, gdy `fit_inverse_transform=True`.

Następnie możemy wyliczyć błąd przeciwbrazu rekonstrukcji:

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.78630879576612
```

Teraz jesteśmy w stanie wykorzystać przeszukiwanie siatki wraz ze sprawdzianem krzyżowym, aby określić rodzaj jądra i wartości hiperparametrów minimalizujące ten błąd.

⁹ Jeżeli wyznaczysz parametr `fit_inverse_transformer=True`, to moduł Scikit-Learn wykorzysta algorytm (oparty na jądrowej regresji grzbietowej), opisany przez Gokhana H. Bakira i in. w: *Learning to Find Pre-Images* (<https://papers.nips.cc/paper/2417-learning-to-find-pre-images.pdf>), „Proceedings of the 16th International Conference on Neural Information Processing Systems” (2004), s. 449 – 456.

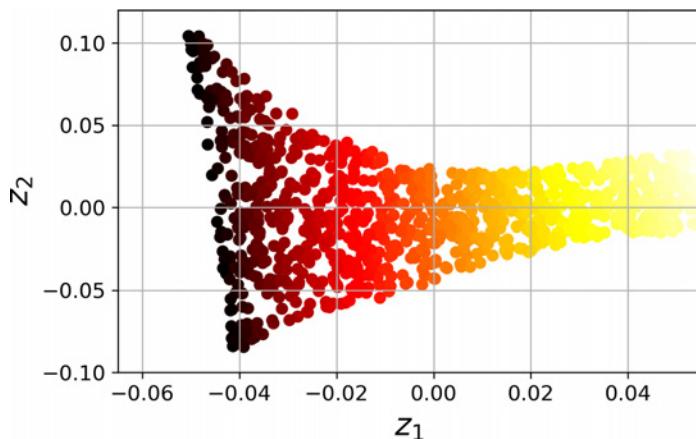
Algorytm LLE

Algorytm **lokalnie liniowego zanurzania**¹⁰ (ang. *locally linear embedding* — LLE) stanowi kolejną potężną technikę **nieliniowej redukcji wymiarowości** (ang. *nonlinear dimensionality reduction* — NLDR). Jest to metoda uczenia rozmaitościowego niewykorzystująca, w przeciwieństwie do poprzednich algorytmów, koncepcji rzutowania. Krótko mówiąc, mechanizm algorytmu LLE polega na zmierzeniu liniowej zależności próbki uczącej od jej najbliższych sąsiadów, a następnie na wyszukiwaniu małymiarowej reprezentacji zestawu uczącego, w których te lokalne relacje zostały zachowane (wkrótce przejdziemy do szczegółów). Rozwiązywanie to przydaje się zwłaszcza do rozwijania poskręcanych rozmaistości, szczególnie jeśli zaszumienie danych nie jest zbyt duże.

Z pomocą poniższego kodu używamy klasy LocallyLinearEmbedding do rozwinięcia zbioru danych Swiss roll:

```
from sklearn.manifold import LocallyLinearEmbedding  
  
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)  
X_reduced = lle.fit_transform(X)
```

Efekt działania algorytmu możemy podziwiać na rysunku 8.12. Jak widać, nasza „roladka” została całkowicie rozwinięta, a odległości pomiędzy poszczególnymi punktami zostały lokalnie całkiem wiernie zachowane. Jednak odległości te nie zostały utrzymane w większej skali: lewa część zbioru danych została rozciągnięta, a prawa — ściśnięta. Mimo wszystko algorytm LLE całkiem dobrze spisał się w modelowaniu rozmaistości.



Rysunek 8.12. Zbiór danych Swiss roll rozwinięty za pomocą algorytmu LLE

¹⁰ Sam T. Roweis i Lawrence K. Saul, *Nonlinear Dimensionality Reduction by Locally Linear Embedding*, „Science” 290, no. 5500 (2000), s. 2323 – 2326, <https://science.sciencemag.org/content/290/5500/2323>.

Mechanizm działania algorytmu przedstawia się następująco: dla każdego przykładu uczącego $\mathbf{x}^{(i)}$ określa on jej k najbliższych sąsiadów (w powyższym kodzie $k = 10$), a następnie stara się zrekonstruować $\mathbf{x}(i)$ jako funkcję liniową tych sąsiadów. Mówiąc dokładnie, znajduje wagi $w_{i,j}$ takie, że kwadrat odległości pomiędzy $\mathbf{x}^{(i)}$ a $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$ jest jak najmniejszy, przy założeniu $w_{i,j} = 0$, jeśli $\mathbf{x}^{(j)}$

nie jest jednym z k najbliższych sąsiadów próbki $\mathbf{x}^{(i)}$. Zatem pierwszym etapem algorytmu LLE jest problem ograniczonej optymalizacji zdefiniowany w równaniu 8.4, gdzie \mathbf{W} stanowi macierz wag zawierającą wszystkie wagi $w_{i,j}$. Drugie ograniczenie po prostu normalizuje wagi dla każdego przykładu uczącego $\mathbf{x}^{(i)}$.

Równanie 8.4. Etap pierwszy algorytmu LLE: liniowe modelowanie lokalnych relacji

$$\hat{\mathbf{W}} = \arg \min_{\mathbf{W}} \sum_{i=1}^m \left(\mathbf{x}^{(i)} - \sum_{j=1}^m \mathbf{w}_{i,j} \mathbf{x}^{(j)} \right)^2$$

pod warunkiem że

$$\begin{cases} w_{i,j} = 0 \text{ jeśli } \mathbf{x}^{(j)} \text{ nie jest jednym z } k \text{ najbliższych sąsiadów } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 \text{ dla } i = 1, 2, \dots, m \end{cases}$$

Po zakończeniu pierwszego kroku macierz wag $\hat{\mathbf{W}}$ (zawierająca wagi $\hat{w}_{i,j}$) zapisuje lokalne, liniowe zależności pomiędzy próbками uczącymi. Drugim etapem jest mapowanie przykładów uczących na d-wymiarową przestrzeń (gdzie $d < n$), przy jednoczesnym zachowaniu tych lokalnych relacji w jak największym stopniu. Jeżeli $\mathbf{z}^{(i)}$ jest obrazem $\mathbf{x}^{(i)}$ w d-wymiarowej przestrzeni, to chcemy, aby kwadrat odległości pomiędzy $\mathbf{z}^{(i)}$ a $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$ był jak najmniejszy. Koncepcja ta prowadzi do problemu nieograniczonej optymalizacji, opisanego za pomocą równania 8.5. Krok ten bardzo przypomina pierwszy etap, ale zamiast znajdowania optymalnych wag przy niezmiennych próbkach wykonujemy odwrotną operację: zachowujemy stałe wartości wag i wyszukujemy optymalne położenie obrazów przykładów w małymiarowej przestrzeni. Zwróć uwagę, że macierz Z zawiera wszystkie próbki $\mathbf{z}^{(i)}$.

Równanie 8.5. Etap pierwszy algorytmu LLE: redukowanie wymiarowości przy jednoczesnym zachowaniu relacji

$$\hat{\mathbf{Z}} = \arg \min_{\mathbf{Z}} \sum_{i=1}^m \left(\mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Implementacja algorytmu LLE w module Scikit-Learn ma następującą złożoność obliczeniową: $O(m \log(m) n \log(k))$ dla znalezienia k najbliższych sąsiadów, $O(mnk^3)$ dla optymalizacji wag i $O(dm^2)$ dla konstruowania małymiarowych reprezentacji. Niestety czynnik m^2 w ostatnim wyrażeniu sprawia, że algorytm ten źle się skaluje do bardzo dużych zbiorów danych.

Inne techniki redukowania wymiarowości

Istnieje wiele różnych technik redukcji wymiarowości, z których część jest dostępna w module Scikit-Learn. Poniżej wymieniam najbardziej popularnych przedstawicieli:

Rzutowania losowe (ang. *random projections*)

Jak sama nazwa wskazuje, w technice tej dane są rzutowane na małowymiarową przestrzeń za pomocą losowego rzutowania liniowego. Może brzmi to jak szaleństwo, okazuje się jednak, że takie losowe rzutowanie w rzeczywistości ma bardzo dużą szansę na właściwe zachowanie odległości, co zostało udowodnione matematycznie przez Williama B. Johnsona i Joramę Lindenstraussa w słynnym lemacie. Jakość redukcji wymiarowości zależy od liczby przykładów i wymiarowości docelowej, ale, o dziwo, nie od wymiarowości początkowej. Więcej szczegółów znajdziesz w dokumentacji pakietu `sklearn.random_projection`.

Skalowanie wielowymiarowe (ang. *multidimensional scaling* — MDS)

Redukuje wymiarowość przy jednoczesnej próbie zachowania odległości między przykładami.

Isomap

Tworzy graf poprzez łączenie próbki z jej najbliższymi sąsiadami, a następnie redukuje wymiarowość przy próbie zachowania **odległości geodezyjnej**¹¹ między przykładami.

Stochastyczne zanurzanie sąsiadów przy użyciu rozkładu t (ang. *t-Distributed stochastic neighbor embedding* — t-SNE)

Redukuje wymiarowość przy jednoczesnej próbie grupowania podobnych przykładów. Metoda ta jest głównie stosowana w celach wizualizacji, zwłaszcza ukazywania skupień próbek w wielowymiarowej przestrzeni (np. wizualizowanie obrazów MNIST w przestrzeni dwuwymiarowej).

Liniowa analiza dyskryminacyjna (ang. *linear discriminant analysis* — LDA)

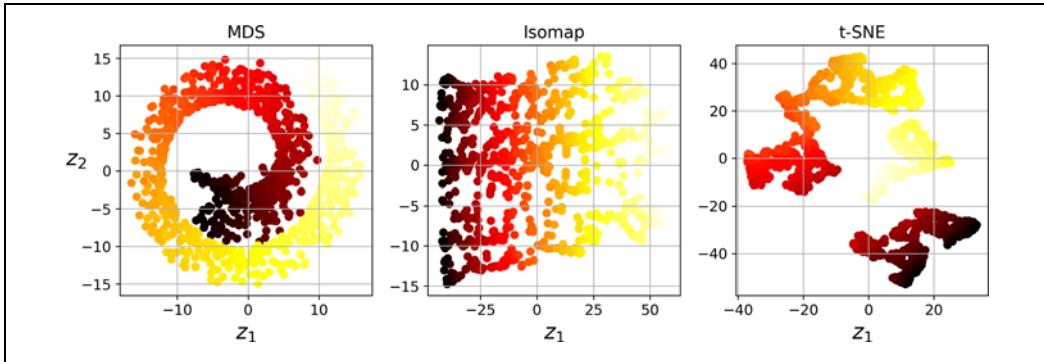
Stanowi algorytm klasyfikacyjny, ale w trakcie uczenia wykrywa on najbardziej charakterystyczne osie pomiędzy klasami, dzięki którym jesteśmy w stanie zdefiniować hiperpłaszczyznę rzutowania danych. Zaletą tego rozwiązania jest maksymalne możliwe rozdzielenie klas, zatem analiza LDA stanowi dobry sposób redukowania wymiarowości przed uruchomieniem innego algorytmu klasyfikującego, np. maszyny SVM.

Rysunek 8.13 przedstawia rezultaty niektórych spośród wymienionych technik.

Ćwiczenia

1. Jakie są główne przyczyny redukowania wymiarowości? Jakie są wady tego rozwiązania?
2. Czym jest kłata wymiarowości?
3. Czy możliwe jest odwrócenie procesu redukowania wymiarowości? Jeśli tak, to w jaki sposób? Jeśli nie, to dlaczego?

¹¹ Odlegością geodezyjną pomiędzy dwoma węzłami grafu nazywamy liczbę węzłów znajdujących się na najkrótszej drodze pomiędzy węzłami docelowymi.



Rysunek 8.13. Redukowanie zbioru danych Swiss roll do dwóch wymiarów za pomocą różnych technik

4. Czy możemy użyć analizy PCA do redukcji wymiarowości bardzo nieliniowego zbioru danych?
5. Założymy, że przeprowadzasz analizę PCA wobec zbioru danych składającego się z 1000 przykładów i wyznaczasz współczynnik wariancji wyjaśnionej na 95%. Ile wymiarów będzie zawierał zredukowany zbiór danych?
6. W jakich przypadkach używamy klasycznej analizy PCA, przyrostowej analizy PCA, losowej analizy PCA i jądrowej analizy PCA?
7. Jak możemy ocenić wydajność algorytmu redukcji wymiarowości wobec zbioru danych uczących?
8. Czy ma sens tworzenie sekwencji dwóch różnych algorytmów redukcji wymiarowości?
9. Wczytaj zbiór danych MNIST (omówiony w rozdziale 3.) i podziel go na podzbiory uczący oraz testowy (pierwsze 60 000 próbek wyznacz do uczenia, a pozostałe przeznacz do testowania). Wyucz klasyfikator losowego lasu wobec tego zestawu danych i zmierz czas, jaki mu to zajmie, a następnie oceń wydajność modelu za pomocą zbioru testowego. Teraz użyj analizy PCA do zredukowania wymiarowości zbioru danych przy ustalonym współczynniku wariancji wyjaśnionej na poziomie 95%. Wytrenuj nowy klasyfikator losowego lasu wobec zredukowanego zbioru danych i porównaj czas potrzebny na naukę modelu z poprzednim wynikiem. Czy proces nauki przebiegał znacznie szybciej? Ponownie oceń wydajność modelu za pomocą podzbioru testowego. Jakie uzyskujesz wyniki w stosunku do wcześniejszego modelu?
10. Użyj algorytmu t-SNE w celu zredukowania zestawu danych MNIST do dwóch wymiarów i stwórz wykres wynikowy przy użyciu modułu Matplotlib. Możesz skorzystać z wykresu punktowego zawierającego 10 kolorów reprezentujących klasę docelową każdego obrazu. Ewentualnie możesz zastąpić każdą kropkę na wykresie punktowym klasą odpowiedniego przykładu (cyfrą od 0 do 9), a nawet stworzyć wykres składający się z miniaturowych wersji obrazów cyfr (jeżeli umieścis na wykresie wszystkie obrazy, staną się on nieczytelny, dlatego powinienni być użyty losowy podzbiór próbek albo przykłady powinny być umieszczane tylko wtedy, gdy w pobliżu nie znajdują się już inne punkty). Powinnas/powinieneś uzyskać ładny wykres z wyraźnie rozdzielonymi skupieniami cyfr. Spróbuj skorzystać z innych algorytmów redukcji wymiarowości, takich jak PCA, LLE czy MDS, i porównaj uzyskiwane za ich pomocą wizualizacje.

Rozwiązań tych ćwiczeń znajdziesz w dodatku A.

Techniki uczenia nienadzorowanego

Chociaż większość współczesnych zastosowań uczenia maszynowego bazuje na uczeniu nadzorowanym (co sprawia, że dział ten jest najlepiej finansowany), to znaczna część dostępnych danych jest nieoznakowana: dysponujemy cechami wejściowymi X , ale nie etykietami y . Informatyk Yann LeCun ukuł słynne porównanie: „gdyby inteligencja była tortem, uczenie nienadzorowane byłoby biszkoptem wraz z masą tortową, uczenie nadzorowane stałoby się polewą, natomiast uczenie przez wzmacnianie okazałoby się wisienką na tym torcie”. Innymi słowy w uczeniu nienadzorowanym istnieje olbrzymi potencjał, w który dopiero zaczeliśmy się wgryzać.

Załóżmy, że chcesz stworzyć system wykonujący po kilka zdjęć każdego elementu sunącego po linii produkcyjnej i wykrywający usterki. Całkiem łatwo opracować system, który wykonuje automatycznie zdjęcia, generując codziennie tysiące obrazów. W ten sposób możesz uzyskać w miarę duży zestaw danych w ciągu zaledwie kilku tygodni. Ale momencik, przecież te zdjęcia nie mają etykiet! Jeżeli chcesz wytrenować standardowy klasyfikator binarny przewidujący, czy dany element jest wadliwy, musisz oznaczyć każde zdjęcie jako zawierające element „wadliwy” lub „prawidłowy”. Zazwyczaj oznacza to zatrudnienie ludzkich ekspertów, którzy muszą przejrzeć cały zestaw danych. Jest to czasochłonne, kosztowne i monotonne zadanie, dlatego przeważnie jest wykonywane na małym podzbiorze dostępnych zdjęć. W konsekwencji oznakowany zestaw danych nie będzie wcale taki duży, a wydajność klasyfikatora okaże się rozczarowująca. Ponadto w sytuacji, gdy firma wprowadzi jakąkolwiek modyfikację produkowanych elementów, cały proces należy powtórzyć od samego początku. Czy nie byłoby cudownie, gdyby algorytm mógł wykorzystywać nieoznakowane dane bez konieczności tworzenia etykiet przez ludzi? W tym miejscu wkracza uczenie nienadzorowane.

W rozdziale 8. przyjrzyliśmy się najpopularniejszemu zadaniu uczenia maszynowego: redukcji wymiarowości. Teraz poznasz kolejne zadania i algorytmy uczenia nienadzorowanego:

Analiza skupień/grupowanie (ang. *clustering*)

Celem tutaj jest pogrupowanie podobnych przykładów w **skupienia/grupy** (ang. *clusters*). Analiza skupień stanowi znakomite narzędzie analizowania danych, segmentacji klientów, systemów rekomendacji, wyszukiwarek, segmentacji obrazu, uczenia półnadzorowanego, redukcji wymiarowości itd.

Wykrywanie anomalii

Celem jest poznawanie struktury „normalnych” danych, a następnie wykorzystanie uzyskanej w ten sposób wiedzy do wykrywania anomalii, takich jak wadliwe elementy na taśmie produkcyjnej czy nowy trend w szeregu czasowym.

Szacowanie gęstości

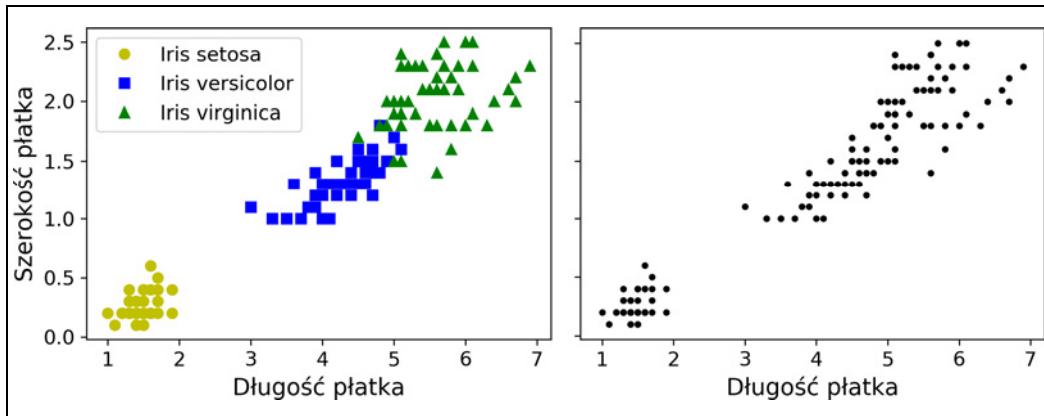
Jest to szacowanie **funkcji gęstości prawdopodobieństwa** (ang. *probability density function* — PDF) procesu losowego odpowiedzialnego za wygenerowanie zestawu danych. Szacowanie gęstości jest standardowo stosowane w zadaniu wykrywania anomalii: przykłady mieszczące się w obszarach o bardzo małej gęstości okazują się zazwyczaj anomaliami. Technika ta przydaje się również przy analizowaniu i wizualizowaniu danych.

Odrobinę tortu? Zaczniemy od analizy skupień za pomocą algorytmów centroidów i DBSCAN, następnie przejdziemy do modelu mieszanych gaussowskich, a także przekonamy się, w jaki sposób przydaje się on w zadaniach szacowania gęstości, analizy skupień i wykrywania anomalii.

Analiza skupień

Podczas wędrówki po górach natrafiasz na gatunek rośliny, jakiego wcześniej nie widziałaś/widziałeś. Rozglądasz się dookoła i dostrzegasz jeszcze kilka innych okazów. Nie są one identyczne, ale wystellarzająco podobne do siebie, więc na pierwszy rzut oka możesz stwierdzić, że są przedstawicielami tego samego gatunku (a przynajmniej rodzaju). Do określenia gatunku może być potrzebny botanik, ale zdecydowanie nie musisz być ekspertem, aby rozpoznać grupę podobnych obiektów. Jest to tak zwana **analiza skupień** — zadanie rozpoznawania podobnych przykładów i przydzielania ich do **skupień**, czyli grup podobnych do siebie przykładów.

Podobnie jak w przypadku klasyfikacji, każdy przykład zostaje przydzielony do grupy. Różnica polega jednak na tym, że analiza skupień jest zadaniem uczenia nienadzorowanego. Spójrz na rysunek 9.1: po lewej stronie widzimy opisany w rozdziale 4. zestaw danych *Iris*, gatunek każdego przykładu (tzn. jego klasa) został oznaczony innym znacznikiem. Jest to oznakowany zestaw danych, dostosowany do takich algorytmów klasyfikujących, jak klasyfikatory regresji logistycznej, maszyny SVM czy klasyfikatory lasów losowych. Na prawym wykresie widzimy ten sam zestaw danych, pozbawiony jednak etykiet, przez co nie jesteśmy w stanie użyć algorytmu klasyfikującego. Właśnie w takiej sytuacji przydają się algorytmy analizy skupień — wiele z nich z łatwością wykrywa skupienie znajdujące się w lewej dolnej części wykresu. My również jesteśmy w stanie je dostrzec gołym okiem, ale fakt, że skupienie znajdujące się w lewym górnym rogu składa się z dwóch oddzielnych podgrup, nie jest już taki oczywisty. Pamiętajmy, że zestaw danych zawiera dwie dodatkowe cechy (długość i szerokość działki kielicha), które nie zostały tu uwzględnione, natomiast algorytmy analizy skupień potrafią znakomicie wykorzystywać wszystkie dostępne cechy, zatem w rzeczywistości rozpoznają one całkiem skutecznie wszystkie trzy skupienia (np. za pomocą modelu mieszanych gaussowskiej jedynie pięć przykładów ze 150 zostaje przydzielonych do niewłaściwego skupienia).



Rysunek 9.1. Klasifikacja (po lewej) a analiza skupień (po prawej)

Analiza skupień ma wiele różnych zastosowań, w tym takie jak:

Segmentacja klientów

Możesz grupować klientów na podstawie historii ich zakupów i aktywności w serwisie. Przydaje się to do zrozumienia kategorii klientów i ich potrzeb, dzięki czemu jesteś w stanie dostosować produkty i kampanie reklamowe do każdego segmentu. Segmentacja klientów przydaje się na przykład w **systemach rekomendacji** sugerujących treści, wobec których pozostały klienci w danym skupieniu wyrażali zainteresowanie.

Analiza danych

Podczas analizowania nowego zestawu danych warto uruchomić algorytm grupowania, a następnie przeanalizować osobno poszczególne skupienia.

Technika redukowania wymiarowości

Po przeprowadzeniu analizy skupień w danym zestawie danych można zazwyczaj zmierzyć **podobieństwo** (ang. *affinity*) przykładowu do każdego skupienia (w tym kontekście podobieństwem jest każda miara dopasowania przykładowu do grupy). Następnie można zastąpić wektor cech \mathbf{x} każdego przykładowu wektorem podobieństw do poszczególnych grup. W przypadku wykrycia k skupień wektor ten będzie k -wymiarowy. Zwykle zawiera on znacznie mniej wymiarów niż pierwotny wektor cech, ale zachowuje wystarczająco wiele informacji, aby model działał prawidłowo.

Wykrywanie anomalii (wykrywanie elementów odstających)

Każdy przykład o małym podobieństwie do każdego skupienia stanowi najprawdopodobniej anomalię. Jeżeli na przykład użytkownicy serwisu zostali pogrupowani pod względem zachowania, jesteś w stanie wykrywać użytkowników o niecodziennym zachowaniu, np. niespotykane duża liczba wysyłanych zapytań na sekundę. Wykrywanie anomalii przydaje się zwłaszcza do wykrywania usterek na etapie produkcji lub do **wykrywania nadużyć finansowych** (ang. *fraud detection*).

Uczenie półnadzorowane

Jeżeli dysponujesz jedynie kilkoma etykietami, możesz przeprowadzić analizę skupień i rozprowadzić je pomiędzy wszystkie przykłady tworzące daną grupę. Technika ta pozwala znacznie zwiększyć liczbę etykiet dostępnych dla algorytmu uczenia nadzorowanego, co bardzo zwiększa jego skuteczność.

Wyszukiwarki

Niektóre wyszukiwarki pozwalają szukać obrazów podobnych do obrazu referencyjnego. W celu stworzenia takiego systemu należy najpierw przeprowadzić analizę skupień dla wszystkich obrazów w bazie danych; podobne obrazy mieściłyby się w ramach tej samej grupy. Następnie po dostarczeniu obrazu referencyjnego przez użytkownika wystarczy wykorzystać algorytm analizy skupień, aby znaleźć grupę, do której pasuje ten obraz, po czym zwrócić wszystkie obrazy tworzące to skupienie.

Segmentacja obrazu

Dzięki analizie pikseli ze względu na ich barwę, a następnie zastąpieniu koloru uśrednioną wartością barwy danego skupienia możemy znaczco zmniejszyć liczbę kolorów tworzących obraz. Segmentacja obrazu wykorzystywana jest w wielu systemach detekcji i śledzenia obiektów, ponieważ ułatwia wykrywanie konturów poszczególnych obiektów.

Nie istnieje uniwersalna definicja skupienia, wszystko bowiem zależy od kontekstu, a różne algorytmy będą wykrywały odmienne typy grup. Niektóre algorytmy wyszukują przykłady ulokowane w pobliżu szczególnego punktu, zwanego **centroidem**. Inne algorytmy znajdują obszary o większym zagęszczeniu przykładów — tego typu skupienia mogą przyjmować najróżniejsze kształty. Część algorytmów jest hierarchicznych — wyszukują one grupy skupień. Można tak wymieniać jeszcze długo.

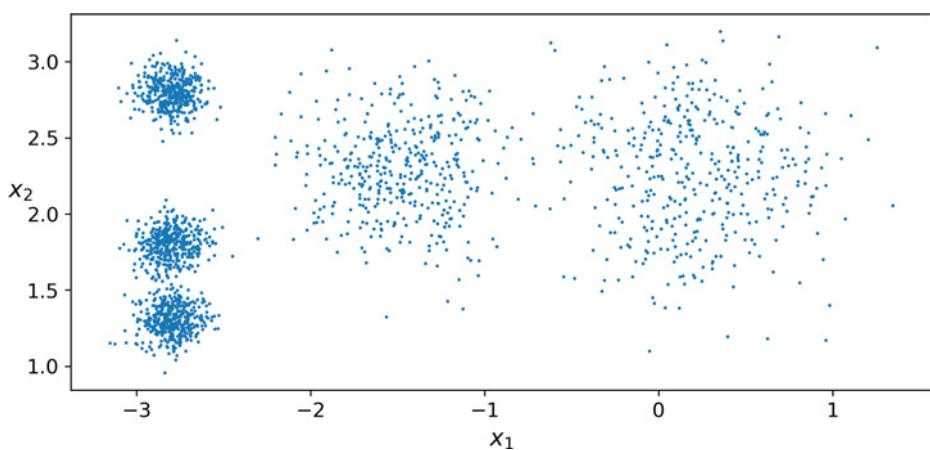
W niniejszym podrozdziale przyjrzymy się dwóm popularnym algorytmom analizy skupień: algorytmowi centroidów i DBSCAN, po czym omówimy niektóre z ich zastosowań, takie jak nieliniowa redukcja wymiarowości, uczenie półnadzorowane i wykrywanie anomalii.

Algorytm centroidów

Przyjrzyj się nieznakowanemu zestawowi danych pokazanemu na rysunku 9.2: wyraźnie widać pięć skupisk przykładów. Algorytm centroidów (zwany także algorytmem k-średnich, ang. *k-means*) jest prostym algorytmem zdolnym do bardzo szybkiego i skutecznego grupowania tego typu zestawów danych, nierzaz w zaledwie kilku przebiegach. Został on zaproponowany przez Stuarta Lloyda w firmie Bell Labs w 1957 roku jako technika modulacji impulsowo-kodowej, ale został udostępniony publicznie dopiero w 1982 roku¹. W 1965 roku Edward W. Forgy opublikował w zasadzie ten sam algorytm, dlatego czasami nazywany jest algorytmem Lloyd-Forgy`ego.

Wy trenujemy algorytm centroidów na tym zestawie danych. Spróbujmy on znaleźć środek każdego skupienia i przydzielić każdy przykład do najbliższego skupienia:

¹ Stuart P. Lloyd, *Least Squares Quantization in PCM*, „IEEE Transactions on Information Theory” 28, no. 2 (1982), s. 129 – 137, https://sites.cs.ucsb.edu/~veronika/MAE/kmeans_LLloyd_Least_Squares_Quantization_in_PCM.pdf.



Rysunek 9.2. Nieoznakowany zestaw danych składający się z pięciu grup przykładów

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

Zwróc uwagę, że konieczne jest określenie liczby skupień k , jakie mają zostać znalezione przez algorytm. W tym przykładzie jest oczywiste po spojrzeniu na wykres, że wartość k powinna być równa 5, zasadniczo jednak prawie nigdy nie jest tak łatwo. Wkrótce powrócimy do tego zagadnienia.

Każdy przykład został przydzielony do jednego z pięciu skupień. W kontekście grupowania **etykieta** przykładu stanowi indeks skupienia, do którego przykład ten zostaje przydzielony przez algorytm. Nie należy jej mylić z etykietami klas w zadaniach klasyfikacji (jak pamiętamy, analiza skupień stanowi zadanie uczenia nienadzorowanego). Wystąpienie KMeans przechowuje kopię etykiet przykładów uczących, dostępną w zmiennej `labels_`:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

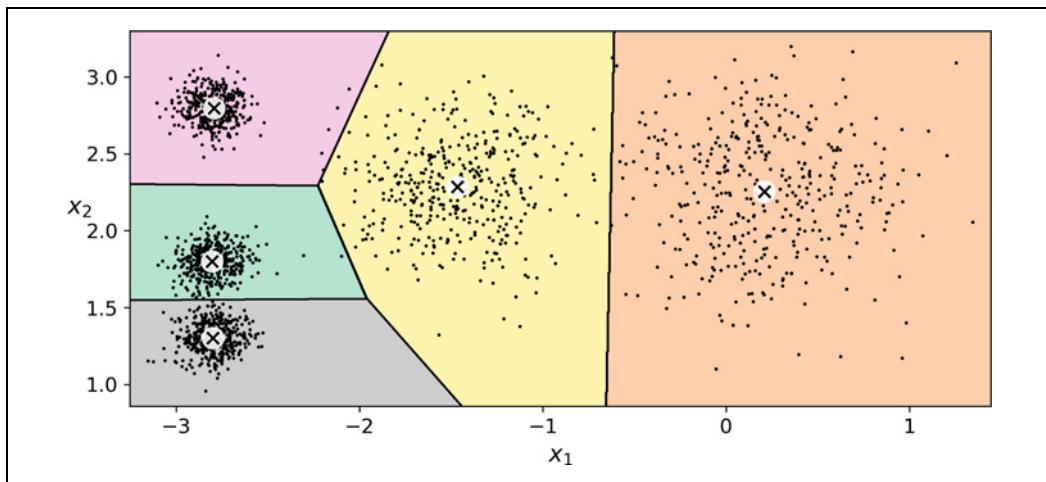
Możesz także przyjrzeć się pięciu centroidom znalezionym przez algorytm:

```
>>> kmeans.cluster_centers_
array([[-2.80389616, 1.80117999],
       [ 0.20876306, 2.25551336],
       [-2.79290307, 2.79641063],
       [-1.46679593, 2.28585348],
       [-2.80037642, 1.30082566]])
```

Jesteś też w stanie przydzielać nowe przykłady do skupienia, którego centroid znajduje się najbliżej:

```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

Jeżeli utworzysz wykres granic decyzyjnych skupień, uzyskasz diagram Woronoja (rysunek 9.3; każdy centroid jest oznaczony symbolem X).



Rysunek 9.3. Granice decyzyjne algorytmu centroidów (diagram Woronoja)

Zdecydowana większość przykładów została prawidłowo przydzielona do odpowiedniego skupienia, ale kilku z nich prawdopodobnie została nadana niewłaściwa etykieta (zwłaszcza w pobliżu granicy pomiędzy skupieniami lewym górnym i środkowym). Faktycznie, algorytm centroidów nie radzi sobie zbyt dobrze w przypadku skupisk o diametralnie różnych średnicach, ponieważ podczas przydzielania przykładu do skupienia zwraca uwagę wyłącznie na jego odległość od centroidu.

Zamiast przydzielać każdy przykład do określonego skupienia (**twarda analiza skupień**) czasami lepiej wyznaczyć przykładowi wynik przynależności do każdej grupy (**miękką analizę skupień**). Takim wynikiem może być odległość pomiędzy przykładem a centroidem; równie dobrze może być też miarą podobieństwa, taką jak na przykład gaussowska radialna funkcja bazowa (zob. rozdział 5.). W klasie KMeans występuje metoda transform(), mierząca odległość każdego przykładu od poszczególnych centroidów:

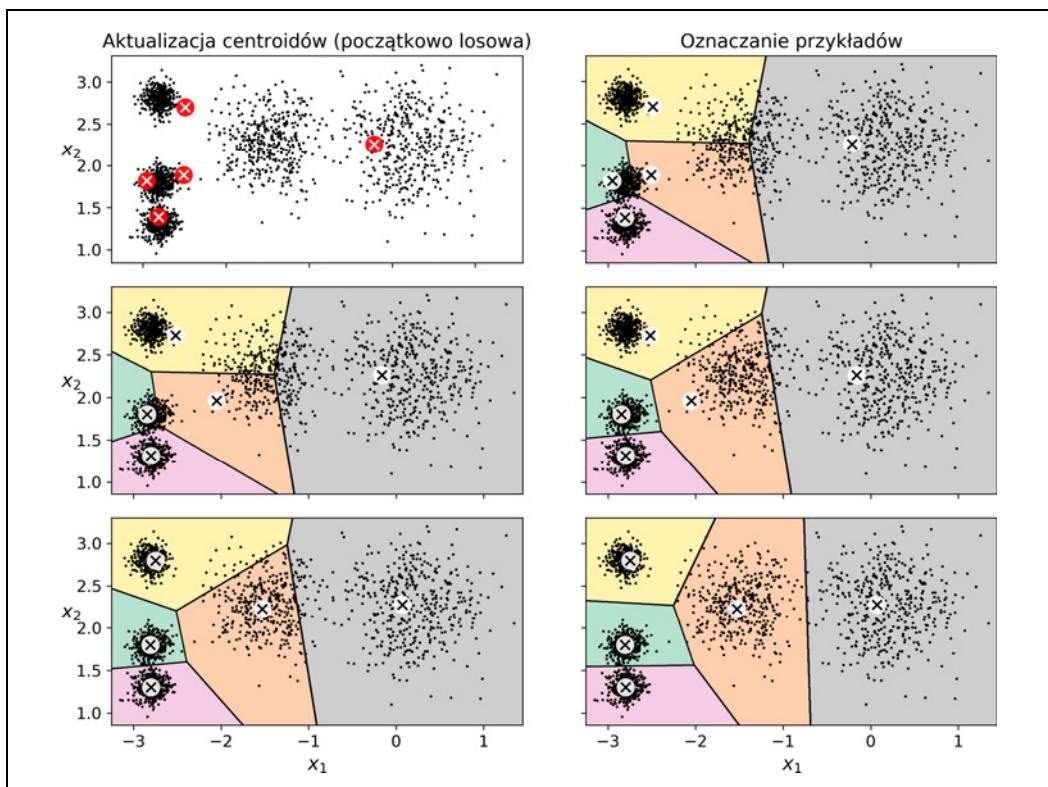
```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

W tym przykładzie pierwszy przykład w `X_new` znajduje się w odległości 2,81 od pierwszego centroidu, 0,33 od drugiego, 2,9 od trzeciego, 1,49 od czwartego i 2,89 od piątego. Jeżeli dysponujesz wielowymiarowym zestawem danych i przekształć go w ten sposób, uzyskasz k -wymiarowy zestaw danych — tego typu transformacja może stanowić bardzo wydajną technikę nieliniowej redukcji wymiarowości.

Mechanizm działania algorytmu

Jak więc ten algorytm działa? Założmy, że dysponujesz centroidami. Możesz z łatwością oznakować wszystkie przykłady w zestawie danych, przydzielając je do skupienia, którego centroid znajduje się najbliżej danego przykładu. Z drugiej strony, gdybyś miała/miał wszystkie etykiety przykładów, bez problemu zlokalizowałabyś/zlokalizowałbyś wszystkie centroidy, obliczając średnią przykładów dla każdej grupy. Nie masz jednak ani etykiet, ani centroidów, zatem co możesz zrobić? Zaczni od losowego rozmieszczenia centroidów (tzn. losowego wybrania k przykładów, które posłużą za centroidy). Następnie oznacz przykłady, zaktualizuj centroidy, znowu oznacz przykłady, zaktualizuj centroidy i powtarzaj te czynności aż do chwili, gdy centroidy przestaną się przemieszczać. Algorytm z całkowitą pewnością osiągnie zbieżność w skończonej liczbie kroków (przeważnie dość małe), nie będzie oscylował przez wieczność².

Na rysunku 9.4 widać działanie tego algorytmu: centroidy zostają zainicjalizowane losowo (lewy górny wykres), następuje oznakowanie przykładów (prawy górny), centroidy są zaktualizowane (lewy środkowy), podobnie jak etykiety przykładów (prawy środkowy) itd. Jak widać, w zaledwie trzech iteracjach algorytm uzyskał rezultat zbliżony do optymalnego.



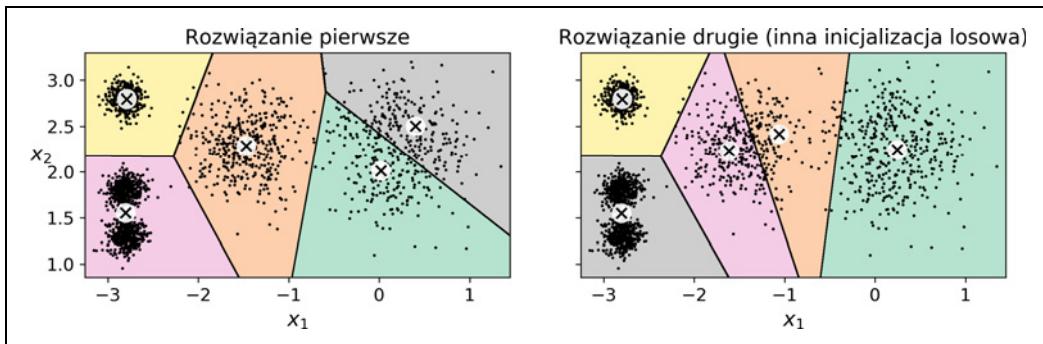
Rysunek 9.4. Algorytm centroidów

² Wynika to z faktu, że średni kwadrat odległości pomiędzy przykładami a najbliższym centroidem może z każdym krokiem jedynie maleć.



Złożoność obliczeniowa algorytmu jest zasadniczo liniowa w odniesieniu do liczby przykładów m , liczby skupień k i liczby wymiarów n . Jest to jednak prawdziwe jedynie wtedy, gdy dane mają strukturę grupową. Jeśli tak nie jest, to w najgorszym wypadku złożoność może wzrastać wykładniczo wraz z liczbą przykładów. W praktyce taka sytuacja zdarza się niezmiernie rzadko i algorytm centroidów okazuje się jednym z najszybszych algorytmów analizy skupień.

Pomimo że algorytm z pewnością uzyska zbieżność, może stać się zbieżny z nieprawidłowym rozwiązańiem (tzn. może stać się zbieżny z optimum lokalnym) — wszystko zależy od zainicjalizowania centroidów. Rysunek 9.5 prezentuje dwa niezupełnie optymalne rozwiązania, jakie może uzyskać algorytm, jeżeli będziesz mieć pecha na etapie inicjalizacji losowej.



Rysunek 9.5. Niezupełnie optymalne rozwiązania wynikające z niefortunnego inicjalizowania centroidów

Przyjrzymy się kilku sposobom uniknięcia tego ryzyka poprzez usprawnienie inicjalizacji centroidów.

Metody inicjalizowania centroidów

Jeżeli akurat wiesz w przybliżeniu, gdzie powinny znajdować się centroidy (np. wykorzystałaś/wykorzystałeś wcześniej inny algorytm analizy skupień), to możesz wyznaczyć w hiperparametrze `init` tablicę NumPy zawierającą listę centroidów i wprowadzić wartość 1 w hiperparametrze `n_init`:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Innym rozwiązaniem jest wielokrotne uruchamianie algorytmu z różnymi wartościami inicjalizacji losowej i pozostawienie najlepszego rozwiązania. Sterujemy liczbą inicjalizacji losowych za pomocą hiperparametru `n_init` — jego wartość domyślna wynosi 10, co oznacza, że cały omówiony wcześniej algorytm ma dokładnie 10 przebiegów po wywołaniu metody `fit()`, a Scikit-Learn zachowuje najlepsze rozwiązanie. Skąd jednak wiadomo, które rozwiązanie jest najlepsze? Zostaje wykorzystany wskaźnik wydajności! Wskaźnik ten jest zwany **bezwładnością** (ang. *inertia*) modelu, co w tym przypadku oznacza średni kwadrat odległości pomiędzy każdym przykładem a najbliższym centroidem. Dla modelu widocznego po lewej stronie na rysunku 9.5 wynosi ona w przybliżeniu 223,3, dla modelu po prawej 237,5, natomiast bezwładność modelu z rysunku 9.3 osiąga wartość 211,6. Klasa `KMeans` realizuje algorytm `n_init` razy i zachowuje model cechujący się najmniejszą bezwładnością. W omawianym przykładzie zostałby wybrany model z rysunku 9.3 (chyba że mielibyśmy naprawdę dużego pecha podczas kolejnych inicjalizacji losowych). Jeżeli chcesz sprawdzić bezwładność modelu, jej wartość jest przechowywana w zmiennej `inertia_`:

```
>>> kmeans.inertia_
211.59853725816856
```

Metoda `score()` zwraca wartość ujemną bezwładności. Dlaczego ujemną? Dlatego, że metoda `score()` predyktora musi zawsze przestrzegać reguły „więcej znaczy lepiej” modułu Scikit-Learn: jeżeli jakiś predyktor jest lepszy od innego, to metoda `score()` powinna zwrócić wynik o wyższej wartości.

```
>>> kmeans.score(X)
-211.59853725816856
```

Ważne udoskonalenie algorytmów centroidów, nazwane **K-Means++**, zostało zaproponowane w artykule z 2006 roku przez Davida Arthura i Sergia Vassilvitskiego³. Wprowadził oni sprytniejszy etap inicjalizacji, w którym dobierane są odległe od siebie centroidy, dzięki czemu znacznie maleje ryzyko uzyskania nieoptimalnego rozwiązania przez algorytm. Udowodniono, że warto poświęcić dodatkowy czas na obliczenia sprytniejszego etapu inicjalizacji, ponieważ znacząco redukuje on liczbę przebiegów algorytmów wymaganą do osiągnięcia optymalnego rozwiązania. Algorytm inicjalizacji K-Means++ wygląda tak:

1. Wybierz jeden centroid $c^{(1)}$ losowo z zestawu danych.
2. Dobierz kolejny centroid $c^{(i)}$, wybierając przykład $x^{(i)}$ o prawdopodobieństwie

$$D(x^{(i)})^2 / \sum_{j=1}^m D(x^{(j)})^2, \text{ gdzie } D(x^{(i)}) \text{ stanowi odległość pomiędzy przykładem } x^{(i)} \text{ a najbliższym, wybranym już centroidem. Taki rozkład prawdopodobieństwa sprawia, że przykłady znajdujące się dalej od wybranych centroidów zostaną z większym prawdopodobieństwem dobrane jako centroidy.}$$

3. Powtarzaj krok 2. aż do wybrania wszystkich k centroidów.

Klasa `KMeans` domyślnie wykorzystuje ten rodzaj inicjalizacji. Jeżeli chcesz wymusić stosowanie metody klasycznej (tzn. losowe wybieranie k przykładów definiujących początkowe centroidy), możesz wyznaczyć wartość "random" w hiperparametrze `init`. Niekrosto jest to konieczne.

Przyspieszony algorytm centroidów i algorytm centroidów z minigrupami

Kolejne ważne usprawnienie algorytmu centroidów zostało opisane w artykule z 2003 roku przez Charlesa Elkana⁴. Przyspiesza ono znacząco działanie algorytmu poprzez pomijanie wielu niepotrzebnych obliczeń odległości. Elkan osiągnął to dzięki wykorzystaniu nierówności trójkąta (tzn. faktu, że najkrótszą odległość pomiędzy dwoma punktami wyznacza zawsze linia prosta⁵), a także poprzez śledzenie ograniczenia dolnego i górnego odległości pomiędzy przykładami a centroidami. Algorytm ten jest stosowany domyślnie przez klasę `KMeans` (możesz wymusić korzystanie z algorytmu

³ David Arthur i Sergei Vassilvitskii, *k-Means++: The Advantages of Careful Seeding*, „Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms” (2007), s. 1027 – 1035, <https://scholar.google.com/scholar?q=k-means%2B%2B%3A+The+advantages+of+careful+seeding+author%3Aarthur>.

⁴ Charles Elkan, *Using the Triangle Inequality to Accelerate k-Means*, „Proceedings of the 20th International Conference on Machine Learning” (2003), s. 147 – 153, <https://www.aaai.org/Papers/ICML/2003/ICML03-022.pdf>.

⁵ Nierówność trójkąta ma postać $AC \leq AB + BC$, gdzie A, B i C są trzema punktami, natomiast AB, AC i BC to odległości pomiędzy tymi punktami.

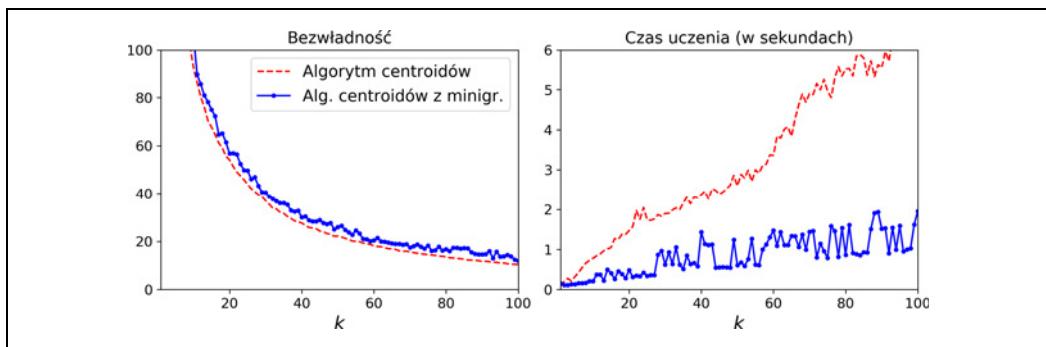
klasycznego, wyznaczając wartość "full" w hiperparametrze `algorithm`, ale raczej nie będziesz musiała/musiał nigdy tego robić).

Jeszcze inny ważny wariant algorytmu centroidów zawdzięczamy Davidowi Sculleyowi, który opublikował interesujący nas artykuł w 2010 roku⁶. W tym przypadku nie wykorzystujemy w każdej iteracji pełnego zestawu danych, lecz algorytm dostosowany do używania minigrup, co sprawia, że w każdym przebiegu centroidy przesuwają się nieznacznie. Rozwiążanie to przyspiesza działanie algorytmu zazwyczaj trzy- lub czterokrotnie i pozwala wykorzystywać olbrzymie zestawy danych, które nie mieszą się w pamięci. Moduł Scikit-Learn implementuje ten algorytm w klasie `MiniBatchKMeans`. Możesz z niej korzystać tak samo jak z klasy `KMeans`:

```
from sklearn.cluster import MiniBatchKMeans  
  
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)  
minibatch_kmeans.fit(X)
```

Jeżeli zestaw danych nie mieści się w pamięci, najprostszym rozwiązaniem okazuje się użycie klasy `memmap` w podobny sposób jak w rozdziale 8. podczas przyrostowej analizy PCA. Ewentualnie możesz przekazywać pojedynczo minigrupy metodzie `partial_fit()`, w tym przypadku jednak należy poświęcić znacznie więcej czasu, ponieważ musisz samodzielnie przeprowadzić wiele inicjalizacji i wybrać najlepszą z nich (przykład tego rozwiązania znajdziesz w sekcji notatnika Jupyter poświęconej algorytmowi centroidów z minigrupami).

Chociaż algorytm centroidów z minigrupami jest dużo szybszy od jego klasycznej wersji, cechuje się przeważnie nieco większą bezwładnością, zwłaszcza w miarę zwiększania liczby skupień. Widać to na rysunku 9.6: na prawym wykresie porównujemy bezwładności algorytmu centroidów z minigrupami i jego klasycznej wersji, które zostały wytrenowane za pomocą poprzedniego zestawu danych i z wyznaczonymi różnymi liczbami skupień (k). Różnica pomiędzy krzywymi pozostaje zasadniczo stała, ale zaczyna się powiększać wraz z parametrem k , gdyż bezwładność stopniowo maleje. Z kolei na prawym wykresie widzimy, że algorytm centroidów z minigrupami jest znacznie szybszy od jego klasycznej wersji i różnica ta rośnie wraz z wartością k .

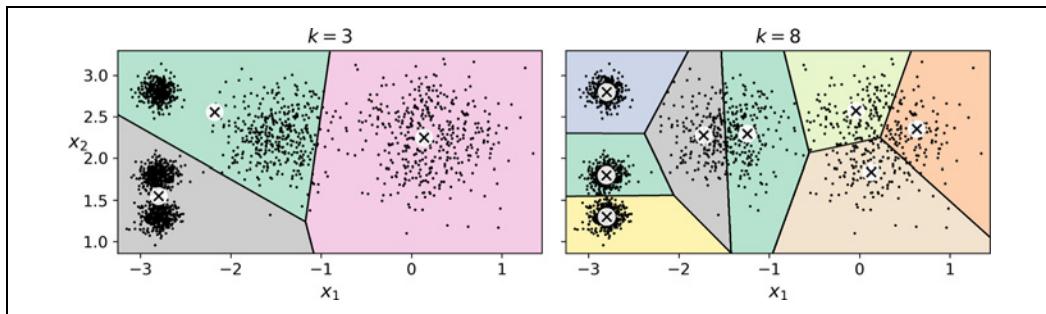


Rysunek 9.6. Algorytm centroidów z minigrupami cechuje się większą bezwładnością od standardowego algorytmu centroidów (po lewej), ale okazuje się znacznie szybszy (po prawej), zwłaszcza wraz ze wzrostem wartości parametru k

⁶ David Sculley, *Web-Scale K-Means Clustering*, „Proceedings of the 19th International Conference on World Wide Web” (2010), s. 1177 – 1178, <https://scholar.google.com/scholar?q=Web-Scale+K-Means+Clustering+author%3Aculley>.

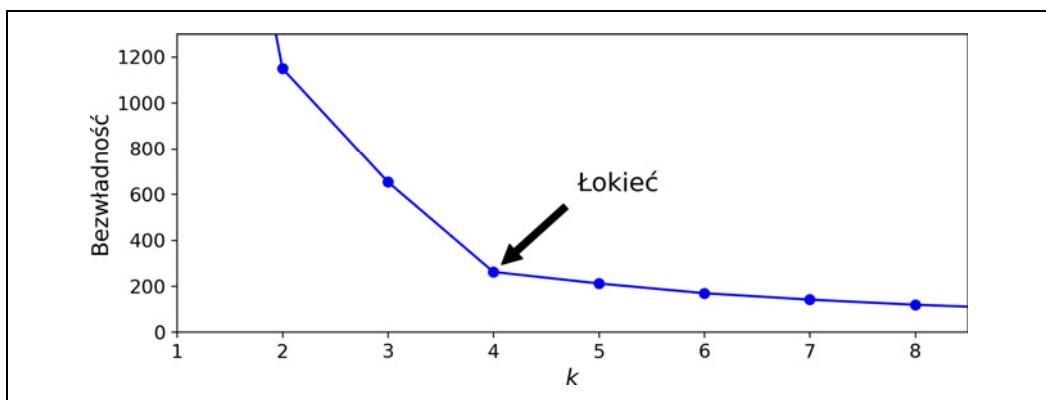
Wyszukiwanie optymalnej liczby skupień

Dotychczas wyznaczaliśmy wartość k równą 5, ponieważ wystarczy jeden rzut oka na zestaw danych, aby stwierdzić, że jest ona prawidłowa. Zasadniczo jednak wyznaczenie parametru k nie jest łatwą sprawą, a ostateczny wynik może okazać się fatalny, jeżeli dobierzymy jego nieodpowiednią wartość. Jak widać na rysunku 9.7, wyznaczenie wartości 3 i 8 generuje złe modele.



Rysunek 9.7. Niewłaściwy wybór liczby skupień: gdy wartość k jest zbyt mała, oddzielne skupienia zostają scalone (po lewej), natomiast gdy jest zbyt duża, niektóre grupy zostają podzielone na mniejsze części (po prawej)

Myślisz zapewne, że wystarczyłoby wybrać model o najmniejszej bezwładności, prawda? Niestety nie jest to takie proste. Inercja dla $k = 3$ wynosi 653,2, czyli znacznie więcej niż w przypadku $k = 5$ (211,6). Jednak w przypadku $k = 8$ osiąga ona wartość zaledwie 119,1. Bezwładność nie stanowi dobrego wskaźnika wydajności podczas decydowania o liczbie skupień, ponieważ maleje wraz ze wzrostem wartości k . W rzeczy samej, im więcej grup, tym każdy przykład ma bliżej do najbliższego centroidu, a zatem tym niższa wartość bezwładności. Narysujmy wykres bezwładności w funkcji k (rysunek 9.8).



Rysunek 9.8. Podczas kreślenia wykresu bezwładności w funkcji liczby skupień krzywa często zawiera punkt odchylenia zwany „łokiem”

Jak widać, do momentu wyznaczenia $k = 4$ bezwładność maleje bardzo szybko, następnie jednak proces ten gwałtownie zwalnia. Uzyskana krzywa przypomina z grubsza rękę, która w punkcie $k = 4$ ma łokieć. Gdybyśmy bazowali tylko na tej informacji, cztery skupienia wydawałyby się dobrym

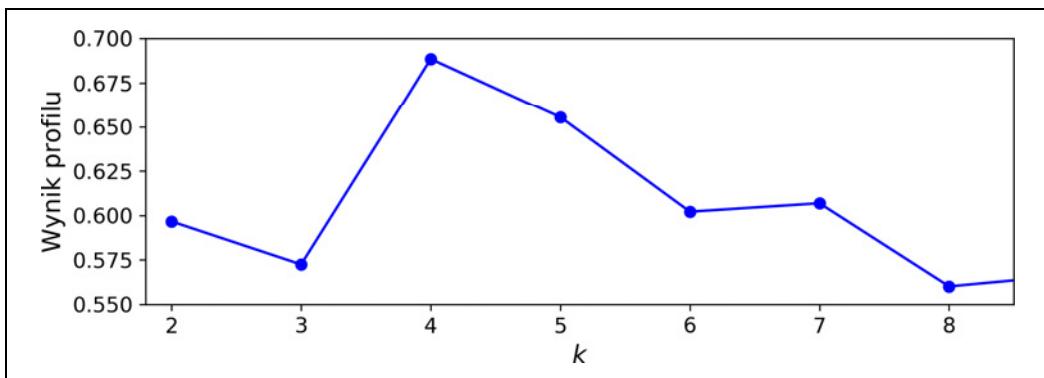
wyborem: każda mniejsza wartość miałaby dramatyczne skutki, natomiast większa nie stanowiłaby dużej pomocy, a my ostatecznie moglibyśmy rozbić całkowicie prawidłowe skupienia na pół bez żadnego dobrego powodu.

Taka technika dobierania optymalnej liczby skupień jest bardzo ogólna. Precyjniejsza (ale jednocześnie kosztowniejsza obliczeniowo) technika polega na wykorzystaniu **wyniku profilu** (ang. *silhouette score*), stanowiącego średni **współczynnik profilu** (ang. *silhouette coefficient*) ze wszystkich przykładów. Współczynnik profilu z jednego przykładu jest równy $(b-a)/\max(a, b)$, gdzie a oznacza średnią odległość do pozostałych przykładów w danym skupieniu (tzn. średnią odległość wewnątrz skupienia), natomiast b to średnia odległość do najbliższej grupy (tzn. średnia odległość do przykładów znajdujących się w najbliższym sąsiadującym skupieniu, zdefiniowanym jako minimalizujące b , przy wykluczeniu skupienia, do którego przynależy ten przykład). Współczynnik profilu może przyjmować wartość od -1 do 1. Wartość bliska 1 oznacza, że przykład znajduje się wewnątrz swojego skupienia i daleko od innych grup, z kolei wartość zbliżona do 0 mówi nam, że przykład ten mieści się w pobliżu granicy skupienia, a im bliżej wartości -1, tym większe prawdopodobieństwo, że przykład został przydzielony do niewłaściwej grupy.

Aby obliczyć wynik profilu, możesz skorzystać z funkcji `silhouette_score()`, dostarczając jej wszystkie przykłady z zestawu danych i przydzielone im etykiety:

```
>>> from sklearn.metrics import silhouette_score  
>>> silhouette_score(X, kmeans.labels_)  
0.655517642572828
```

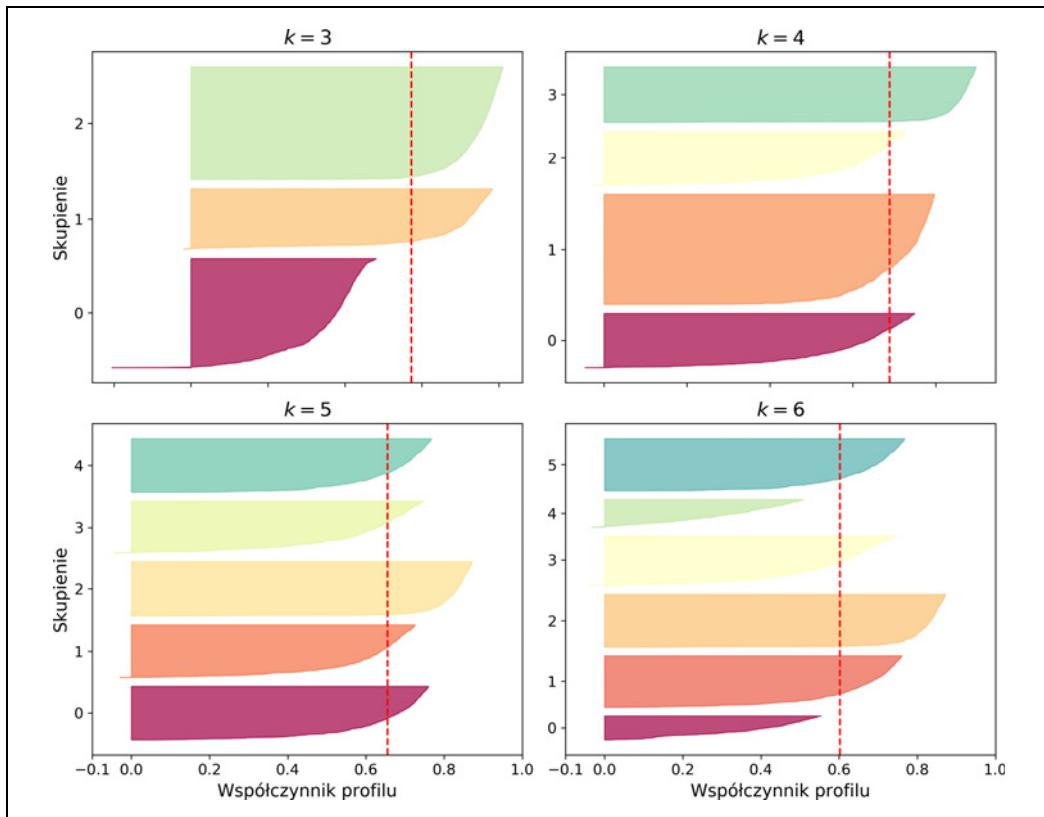
Porównajmy wyniki profilu dla różnych liczb skupień (rysunek 9.9).



Rysunek 9.9. Wybór liczby skupień k za pomocą wyniku profilu

Jak widać, taka wizualizacja dostarcza znacznie więcej informacji w porównaniu do poprzedniej: pomimo potwierdzenia, że $k = 4$ jest bardzo dobrym wyborem, widzimy, że $k = 5$ stanowi także dobrą opcję, znacznie lepszą niż $k = 6$ czy $k = 7$. Nie byliśmy w stanie tego stwierdzić podczas analizowania bezwładności.

Jeszcze więcej informacji uzyskamy, generując wykres współczynników profilu każdego przykładu, uszeregowanych zgodnie ze skupieniem, do którego zostały przydzielony, i wartością współczynnika. Jest to tzw. **wykres profilu** (ang. *silhouette diagram*) (rysunek 9.10). Każdy wykres zawiera kształt „sopelkowy” symbolizujący jedno skupienie. Wysokość kształtu reprezentuje liczbę przykładów



Rysunek 9.10. Analiza wykresów profilu dla różnych wartości k

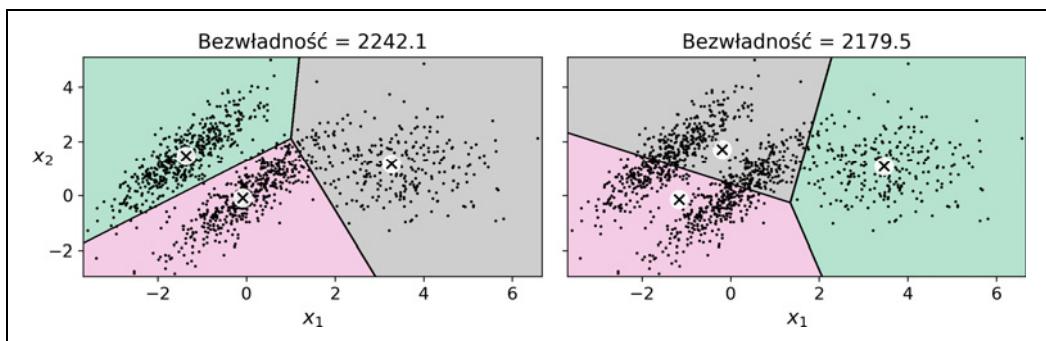
w danym skupieniu, natomiast jego szerokość ukazuje uszeregowane współczynniki profilu dla przykładów tworzących dane skupienie (im szerszy kształt, tym lepiej). Linia przerywana wskazuje średni współczynnik profilu.

Pionowa linia przerywana reprezentuje wynik profilu dla poszczególnych liczb grup. Gdy większość przykładów w danym skupieniu ma niższą wartość współczynnika profilu od tego progu (tzn. jeśli wiele przykładów nie dociera do tej linii i znajduje się po jej lewej stronie), to grupa ta jest prawdopodobnie nieprawidłowa, ponieważ oznacza to, że tworzące ją przykłady znajdują się zbyt blisko innych skupień. Widzimy to w przypadku $k = 3$ i $k = 6$. Ale w sytuacji $k = 4$ i $k = 5$ grupy wyglądają całkiem nieźle: większość przykładów przekracza linię przerywaną i zbliża się do wartości 1. Gdy $k = 4$, skupienie o indeksie 1 (trzecie od góry) jest dość duże, natomiast dla $k = 5$ wszystkie grupy mają podobny rozmiar. Zatem nawet jeśli ogólny wynik profilu dla $k = 4$ jest nieco lepszy od wyniku dla $k = 5$, dobrym rozwiązaniem jest wybór $k = 5$ z powodu uzyskania skupień o zbliżonych rozmiarach.

Granice algorytmu centroidów

Pomimo niezaprzeczalnych zalet, głównie szybkości i skalowalności, algorytm centroidów nie jest doskonały. Jak już widzieliśmy, należy uruchomić algorytm kilka razy, aby uniknąć rozwiązań nieoptimalnych, a do tego musimy wyznaczyć liczbę skupień, co stanowi problem sam w sobie.

Ponadto algorytm ten nie radzi sobie zbyt dobrze z grupami różniącymi się rozmiarem, gęstością lub kształtem odstającym od ovalnego. Na przykład na rysunku 9.11 zaprezentowałem skupienia wyznaczone przez algorytm centroidów na podstawie zestawu danych zawierającego trzy eliptyczne grupy o różnych wymiarach, gęstościach i kierunkach ułożenia.



Rysunek 9.11. Algorytm centroidów nie jest w stanie przeprowadzić prawidłowo analizy eliptycznych skupień

Jak widać, żadne z tych rozwiązań nie jest dobre. Rozwiążanie na lewym wykresie wydaje się lepsze, mimo to nadal ucina 25% środkowego skupienia i przydziela tę część do prawej grupy. Wynik z prawego wykresu jest po prostu fatalny pomimo faktu, że cechuje się mniejszą bezwładnością. Zatem w zależności od danych lepiej mogą się sprawdzać inne algorytmy analizy skupień. W przypadku takich grup eliptycznych świetnie spisują się modele mieszanin gaussowskich.



Istotną czynnością przed uruchomieniem algorytmu centroidów jest skalowanie cech wejściowych, gdyż w przeciwnym wypadku skupienia mogą okazać się bardzo rozciągnięte, co zaszkodzi jego wydajności. Skalowanie cech nie gwarantuje, że wszystkie grupy będą eleganckie i ovalne, zazwyczaj jednak pomaga.

Zastanówmy się teraz, co możemy osiągnąć za pomocą analizy skupień. Skorzystamy z algorytmu centroidów, ale nic nie stoi na przeszkodzie, aby po eksperymentować z innymi technikami grupowania.

Analiza skupień w segmentacji obrazu

Segmentacja obrazu (ang. *image segmentation*) to rozdzielanie obrazu na poszczególne segmenty. W **segmentacji semantycznej** (ang. *semantic segmentation*) wszystkie piksele stanowiące część tego samego typu obiektu zostają przydzielone do tego samego segmentu. Na przykład w systemie wizyjnym pojazdu autonomicznego wszystkie piksele tworzące obraz przechodnia mogą zostać zaklasyfikowane do segmentu „pieszy” (istniałby jeden segment zawierający wszystkich pieszych). W **segmentacji obiektów** (ang. *instance segmentation*) wszystkie piksele tworzące dany obiekt zostają przydzielone do tego samego segmentu. W tym przypadku każdy pieszy należałby do oddzielnego segmentu. Najnowocześniejsze modele segmentacji semantycznej i przykładów są uzyskiwane za pomocą skomplikowanych architektur opartych na splotowych sieciach neuronowych (zob. rozdział 14.). My zajmiemy się czymś znacznie prostszym: **segmentacją kolorów** (ang. *color segmentation*). Będziemy po prostu przydzielać piksele do tego samego segmentu na podstawie podobieństwa barwy. Rozwiązanie to jest wystarczające w niektórych zastosowaniach. Jeżeli chcesz na przykład

określić obszar zalesienia terenu za pomocą zdjęć satelitarnych, segmentacja kolorów powinna całkowicie wystarczyć do tego zadania.

Najpierw skorzystamy z funkcji `imread()` do wczytania obrazu (lewy górny obraz na rysunku 9.12):

```
>>> from matplotlib.image import imread # Lub 'from imageio import imread'  
>>> image = imread(os.path.join("obrazy", "uczenie_nienadzorowane", "ladybug.png"))  
>>> image.shape  
(533, 800, 3)
```

Obraz jest reprezentowany jako trójwymiarowa tablica. Pierwszym wymiarem jest jego wysokość, drugi wymiar to szerokość, natomiast trzeci zawiera informację o kanałach kolorów, w tym przypadku czerwonym (ang. *red*), zielonym (ang. *green*) i niebieskim (ang. *blue*), które tworzą wspólnie model RGB. Innymi słowy, każdy piksel reprezentowany jest przez trójwymiarowy wektor określający nasycenie kolorami czerwonym, zielonym i niebieskim w zakresie od 0,0 do 1,0 (lub od 0 do 255 w przypadku metody `imageio.imread()`). Niektóre obrazy mogą zawierać mniejszą liczbę kanałów, np. obrazy czarno-białe (jednokanałowe). Z kolei inne obrazy mogą mieć ich więcej, np. czasami spotykamy się z dodatkowym **kanałem alfa**, odpowiedzialnym za przezroczystość, lub ze zdjęciami satelitarnymi zawierającymi kanały dla wielu zakresów promieniowania elektromagnetycznego (np. podczerwieni). Za pomocą poniższego kodu przekształcamy tę tablicę tak, aby uzyskać długą listę kolorów RGB, które następnie zostaną poddane analizie skupień za pomocą algorytmu centroidów:

```
X = image.reshape(-1, 3)  
kmeans = KMeans(n_clusters=8).fit(X)  
segmented_img = kmeans.cluster_centers_[kmeans.labels_]  
segmented_img = segmented_img.reshape(image.shape)
```

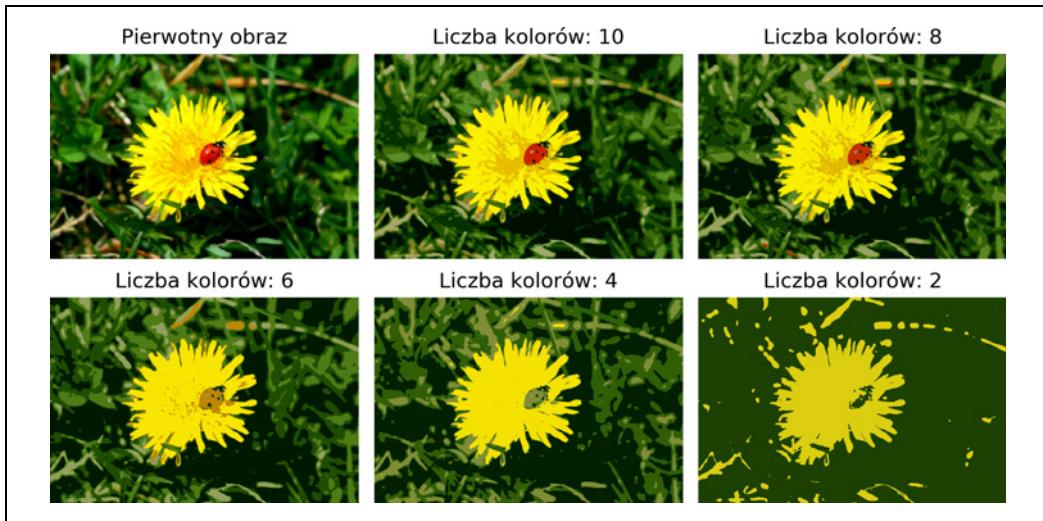
Algorytm ten jest w stanie na przykład wykryć skupienie wszystkich odcieni zielonego. Następnie dla każdej barwy (np. ciemnozielonej) obliczy uśrednioną wartość koloru w danym skupieniu. Przykładowo wszystkie odcienie zielonego mogą zostać zastąpione tą samą jasnozieloną barwą (przy założeniu, że uśrednioną wartością koloru jest jasnozielony). Na koniec ta dłuża lista kolorów zostaje przekształcona z powrotem do rozmiarów pierwotnego obrazu. Gotowe!

Dzięki powyższemu listingowi otrzymujemy obraz widoczny w prawym górnym rogu rysunku 9.12. Jak widać, możesz również poeksperymentować z różnymi liczbami skupień. Jeżeli wyznaczysz mniej niż 8 grup, to algorytm ma problem z rozpoznaniem czerwonego koloru biedronki — zostaje on scalony z barwami otoczenia. Biedronka jest malutka, znacznie mniejsza od całego zdjęcia, dlatego pomimo całej jej wyrazistości algorytm centroidów ma problem w wyznaczeniu jej osobnego skupienia.

Nie było to zbyt trudne, prawda? Przyjrzyjmy się kolejnemu zastosowaniu analizy skupień: przetwarzaniu wstępнемu.

Analiza skupień w przetwarzaniu wstępny

Analiza skupień może stanowić skuteczną metodę redukcji wymiarowości, zwłaszcza na etapie przetwarzania wstępnego przed zastosowaniem algorytmu uczenia nadzorowanego. W kolejnym przykładzie wykorzystamy zestaw danych Digits, który przypomina uproszczoną wersję zestawu danych MNIST. Składa się on z 1797 czarno-białych obrazów o rozmiarach 8×8 reprezentujących cyfry od 0 do 9. Najpierw wczytaj ten zestaw danych:



Rysunek 9.12. Segmentacja obrazu za pomocą algorytmu centroidów przy wyznaczaniu różnej liczby skupień kolorów

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

Następnie podziel go na dane uczące i dane testowe:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

Wytrenuj teraz model regresji logistycznej:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Sprawdź dokładność modelu na danych testowych:

```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

Dokładność rzędu 96,9% będzie naszą wartością bazową. Zobaczmy, czy uda nam się uzyskać lepszą skuteczność po wstępnym przetworzeniu danych za pomocą algorytmu centroidów. Utworzymy potok, w którym zestaw danych uczących zostanie pogrupowany w 50 skupień, a obrazy zostaną zastąpione ich odległościami do każdej grupy, po czym użyjemy algorytmu regresji logistycznej:

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```



Mamy do czynienia z dziesięcioma cyframi, dlatego może Cię kusić, by wyznaczyć dziesięć skupień. Jednak każda z tych cyfr może zostać zapisana na różne sposoby, dlatego lepiej utworzyć większą liczbę grup, jak choćby 50.

Oceńmy teraz ten potok klasyfikacyjny:

```
>>> pipeline.score(X_test, y_test)  
0.9777777777777777
```

Co Ty na to? Zmniejszyliśmy stopę błędu o niemal 30% (z 3,1% do około 2,2%)!

Wyznaczyliśmy jednak liczbę skupień k całkiem dowolnie, a z pewnością istnieje lepszy sposób. Skoro algorytm centroidów stanowi jedynie etap przetwarzania wstępnego w potoku klasyfikacyjnym, określenie prawidłowej wartości k jest łatwiejsze niż wcześniej. Nie musimy przeprowadzać analizy profilu ani minimalizować bezwładności — najlepszą wartością k jest ta, dzięki której uzyskujemy najlepszą wydajność klasyfikacji podczas sprawdzianu krzyżowego. Optymalną liczbę skupień możemy znaleźć za pomocą klasy GridSearchCV:

```
from sklearn.model_selection import GridSearchCV  
  
param_grid = dict(kmeans_n_clusters=range(2, 100))  
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)  
grid_clf.fit(X_train, y_train)
```

Sprawdźmy najlepszą wartość k i wydajność uzyskanego potoku:

```
>>> grid_clf.best_params_  
{'kmeans_n_clusters': 99}  
>>> grid_clf.score(X_test, y_test)  
0.9822222222222222
```

Po wyznaczeniu $k = 99$ skupień jesteśmy świadkami znacznego skoku wydajności, gdyż uzyskujemy 98,22% dokładności dla zestawu testowego. Świetnie! Możesz sprawdzić również większe wartości k , gdyż 99 to maksimum zakresu wyznaczonego przez nas w tym przykładzie.

Analiza skupień w uczeniu półnadzorowanym

Kolejnym przykładem zastosowania analizy skupień jest uczenie półnadzorowane, w którym dysponujemy wieloma przykładami nieoznakowanymi i niewielką liczbą oznakowanymi. Wytrenujmy model regresji logistycznej na próbce składającej się z 50 oznakowanych przykładów stanowiących część zestawu danych Digits:

```
n_labeled = 50  
log_reg = LogisticRegression()  
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Jaka jest wydajność tego modelu dla zestawu testowego?

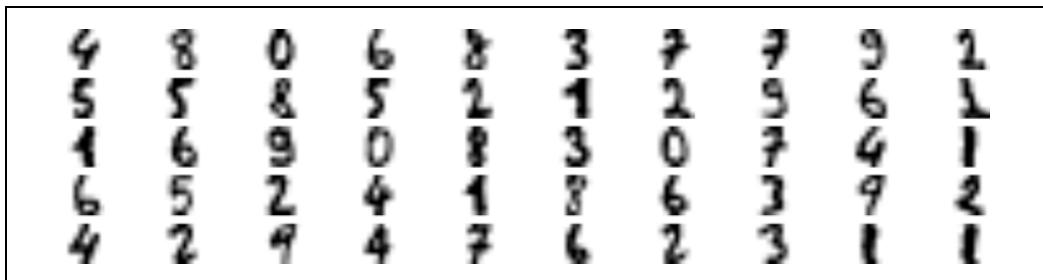
```
>>> log_reg.score(X_test, y_test)  
0.833333333333334
```

Dokładność wynosi zaledwie 83,3%. Nie powinno nas dziwić, że jest ona znacznie mniejsza niż wcześniej, gdy trenowaliśmy model na pełnym zestawie uczącym. Zobaczmy, w jaki sposób możemy poprawić ten wynik. Najpierw pogrupujmy zestaw danych uczących w 50 skupień. Następnie dla

każdego skupienia wyszukajmy obraz znajdujący się najbliżej centroidu. Obrazy te nazwiemy **obrazami reprezentatywnymi**:

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Rysunek 9.13 przedstawia 50 takich obrazów reprezentatywnych.



Rysunek 9.13. Pięćdziesiąt obrazów reprezentatywnych cyfr (po jednym na każde skupienie)

Przyjrzyjmy się każdemu obrazowi i własnoręcznie go oznaczmy:

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Uzyskaliśmy w ten sposób zestaw danych zawierający jedynie 50 oznakowanych przykładów, nie są one jednak losowe, lecz każdy z nich stanowi obraz reprezentatywny danego skupienia. Sprawdźmy, czy udało nam się poprawić wydajność:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9222222222222233
```

Wow! Przeskoczyliśmy z 83,3% dokładności na 92,2%, mimo że trenujemy model wyłącznie na 50 przykładach. Oznaczanie przykładów jest często kosztowne i bolesne, zwłaszcza gdy muszą zajmować się tym eksperci, dlatego dobrym rozwiązaniem okazuje się oznaczanie przykładów reprezentatywnych zamiast losowych.

Może jednak jesteśmy w stanie zrobić coś więcej? A gdybyśmy rozprzestrzenili etykiety na wszystkie pozostałe przykłady w danym skupieniu? Jest to proces zwany **propagacją etykiet** (ang. *label propagation*):

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

Ponownie wytrenujmy teraz model i sprawdźmy jego skuteczność:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9333333333333333
```

Udało nam się nieźle poprawić dokładność, ale nie jesteśmy świadkami niczego spektakularnego. Problem polega na tym, że rozprowadziliśmy etykię każdego przykładu reprezentatywnego na wszystkie przykłady w tym samym skupieniu, w tym również na przykłady znajdujące się na obrzeżach grupy, czyli te, które są bardziej narażone na nieprawidłowe oznakowanie. Sprawdźmy, co się stanie, jeżeli ograniczymy propagację etykiet wyłącznie na 20% przykładów znajdujących się najbliżej centroidów:

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Wyuczmy jeszcze raz nasz model na częściowo oznakowanym zestawie danych:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.94
```

Uczenie aktywne

W celu dalszego usprawnienia modelu i zestawu danych uczących następnym etapem powinno być kilka przebiegów **uczenia aktywnego** (ang. *active learning*), czyli etapu, w którym żywy ekspert oddziałuje z algorytmem uczenia i dostarcza etykiety dla określonych przykładów wyznaczanych przez ten algorytm. Istnieje wiele różnych strategii uczenia aktywnego, a jedna z najpowszechniejszych to **próbkowanie niepewności** (ang. *uncertainty sampling*). Mechanizm jej działania wygląda następująco:

- Model zostaje wyuczony na dotychczas zdobytych przykładach oznakowanych i za jego pomocą uzyskujemy prognozy dla wszystkich przykładów nieoznakowanych.
- Przykłady, wobec których model wykazuje największą niepewność (tzn. tam, gdzie oszacowane prawdopodobieństwo jest najmniejsze), zostają przekazane ekspertowi do oznakowania.
- Powtarzasz ten proces aż do momentu, w którym skoki wydajności przestaną być warte oznakowywania przykładów.

Wśród innych strategii znajdziemy takie jak oznaczanie przykładów, które mogą najmocniej wpływać na zmiany modelu lub powodujące największy spadek błędu walidacyjnego modelu, czy też przykłady, wobec których różne modele (np. maszyny wektorów nośnych czy klasyfikatory lasu losowego) dają odmienne wyniki.

Ładnie! Wystarczyło zaledwie 50 oznakowanych przykładów (średnio po pięć na jedną klasę!), aby uzyskać dokładność rzędu 94%, co jest wynikiem całkiem zbliżonym do uzyskanego przez algorytm regresji logistycznej na całkowicie oznakowanym zestawie danych Digits (otrzymaliśmy wtedy 96,9% dokładności). Taka dobra skuteczność wynika z faktu, że rozprowadzone etykiety są

w rzeczywistości naprawdę niezłe — ich dokładność sięga 99%, co możemy sprawdzić za pomocą następującego listingu:

```
>>> np.mean(y_train_partially_propagated == y_train[partially_propagated])
0.9896907216494846
```

Zanim przejdziemy do modelu mieszanin gaussowskich, przyjrzyjmy się popularnemu algorytmowi DBSCAN, w którym zastosowano zupełnie odmienne podejście, oparte na lokalnym oszacowaniu gęstości. Rozwiążanie to pozwala rozpoznawać skupienia o dowolnych kształtach.

Algorytm DBSCAN

Algorytm ten definiuje skupienia jako ciągłe rejony o dużej gęstości. Jego mechanizm działania jest następujący:

- Algorytm oblicza dla każdego przykładu liczbę przykładów mieszących się w niewielkiej odległości ϵ (epsilon) od niego. Obszar ten nosi nazwę **ϵ -sąsiedztwa** przykładu.
- Jeżeli w ϵ -sąsiedztwie przykładu znajduje się co najmniej `min_samples` przykładów (włącznie z nim samym), to jest on uznawany za **przykład rdzeniowy** (ang. *core instance*). Innymi słowy, przykłady rdzeniowe to te, które mieszą się w obszarach o dużym zagęszczeniu.
- Wszystkie przykłady w sąsiedztwie rdzenia przynależą do tego samego skupienia. W sąsiedztwie tym mogą występować inne przykłady rdzeniowe. Zatem dłużna sekwencja sąsiadujących przykładów rdzeniowych tworzy pojedyncze skupienie.
- Każdy przykład niebędący rdzeniowym, w którego sąsiedztwie nie występują przykłady rdzeniowe, jest uznawany za anomalię.

Algorytm ten sprawuje się bardzo dobrze wtedy, gdy wszystkie skupienia są wystarczająco gęste i gdy są rozdzielone obszarami o małym zagęszczeniu. Klasa DBSCAN z modułu Scikit-Learn jest bardzo łatwa w użyciu. Przetestujmy ją na zestawie danych sierpowatych z rozdziału 5.:

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbSCAN = DBSCAN(eps=0.05, min_samples=5)
dbSCAN.fit(X)
```

Etykiety wszystkich przykładów są teraz dostępne w zmiennej `labels_`:

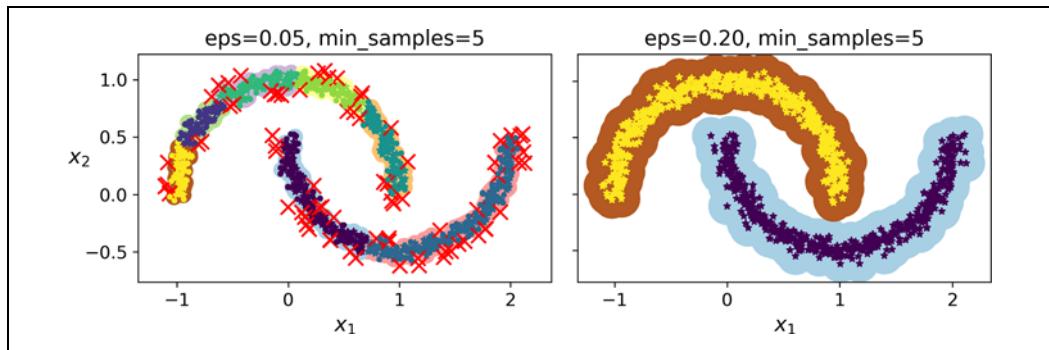
```
>>> dbSCAN.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Zwróć uwagę, że niektóre przykłady mają indeks skupienia o wartości -1 , co oznacza, że algorytm traktuje je jako anomalie. Indeksy przykładów rdzeniowych znajdziemy w zmiennej `core_sample_indices_`, natomiast same przykłady rdzeniowe są dostępne w zmiennej `components_`:

```
>>> len(dbSCAN.core_sample_indices_)
808
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbSCAN.components_
```

```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

Wyniki przeprowadzonej w ten sposób analizy skupień są widoczne na lewym wykresie z rysunku 9.14. Jak widać, algorytm wykrył wiele anomalii i siedem różnych skupień. Co za rozczałowanie! Na szczęście jeżeli poszerzymy sąsiedztwo każdego przykładu, zwiększając wartość eps o 0,2, to zostanie przeprowadzona analiza skupień widoczna na prawym wykresie — ten rezultat wygląda doskonale. Zostańmy przy tym modelu.



Rysunek 9.14. Analiza skupień za pomocą algorytmu DBSCAN przy użyciu dwóch różnych wartości promienia sąsiedztwa

Dosyć zaskakujący jest fakt, że klasa DBSCAN nie zawiera metody `predict()`, ale znajdziemy w niej metodę `fit_predict()`. Inaczej mówiąc, algorytm nie jest w stanie przewidzieć, do którego skupienia przynależy nowy przykład. Ta decyzja projektowa wiąże się z tym, że różne algorytmy klasyfikacji mogą sprawdzać się lepiej w odmiennych zadaniach, dlatego twórcy zapewnili swobodę w ich doborze. Co więcej, implementacja okazuje się całkiem łatwa. Wyuczmy na przykład klasyfikator `KNeighborsClassifier`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbSCAN.components_, dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

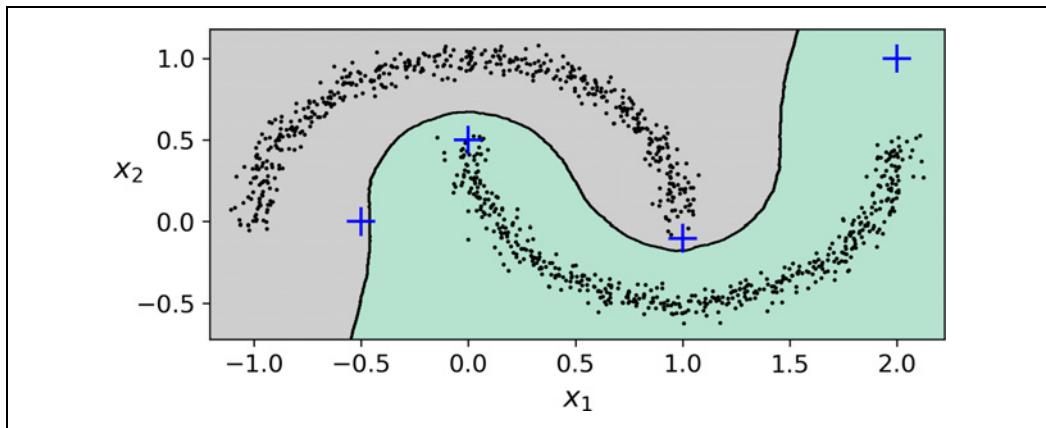
Teraz na podstawie kilku nowych przykładów możemy prognozować skupienia, do których będą one najprawdopodobniej przynależeć, a nawet oszacować prawdopodobieństwo przynależności do poszczególnych grup:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1., 0. ],
       [0.12, 0.88],
       [1., 0. ]])
```

Zwróć uwagę, że wyuczylismy model wyłącznie na przykładach rdzeniowych, jednak równie dobrze moglibyśmy wyznaczyć do tego zadania wszystkie przykłady lub wszystkie z pominięciem anomalii — decyzja ta zależy od zadania końcowego.

Granica decyzyjna została zaprezentowana na rysunku 9.15 (krzyżyki reprezentują cztery przykłady zdefiniowane w X_{new}). Skoro w zestawie danych uczących nie występują żadne anomalie, to klasyfikator zawsze wybiera jakieś skupienie, nawet jeśli okazuje się odległe. Dosyć prostym rozwiązaniem jest wprowadzenie maksymalnej odległości, dzięki której dwa przykłady znajdują się daleko od obydwu skupień i zostają zaklasyfikowane jako anomalie. W tym celu możesz wykorzystać metodę `kneighbors()` z klasy `KNeighborsClassifier`. Mając dany określony zestaw przykładów, metoda ta zwraca odległości i indeksy k najbliższych sąsiadów w zbiorze danych uczących (dwie macierze, każda zawierająca po k kolumnach):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```



Rysunek 9.15. Granica decyzyjna pomiędzy dwoma skupieniami

Mówiąc krótko, DBSCAN jest prostym, ale potężnym algorytmem, zdolnym do rozpoznawania dowolnej liczby skupień o dowolnym kształcie. Jest odporny na elementy odstające i zawiera dwa hiperparametry (eps i min_samples). Jeżeli jednak poszczególne skupienia różnią się znacząco gęstością, algorytm ten może nie być w stanie prawidłowo określić wszystkich skupień. Jego złożoność obliczeniowa wynosi w przybliżeniu $O(m \log m)$, co sprawia, że jest niemal liniowa pod względem liczby przykładów, ale implementacja zawarta w module Scikit-Learn może wymagać do $O(m^2)$ pamięci przy dużej wartości hiperparametru eps .



Możesz również wypróbować **hierarchiczny algorytm DBSCAN** (HDBSCAN), zaimplementowany w projekcie `scikit-learn-contrib` (<https://github.com/scikit-learn-contrib/hdbscan/>).

Inne algorytmy analizy skupień

Moduł Scikit-Learn zawiera implementacje kilku innych algorytmów analizy skupień, na które warto zwrócić uwagę. Nie mogę ich tu opisać szczegółowo, ale zamieszczać ich ogólny przegląd:

Agglomeracyjna analiza skupień

Tworzona jest hierarchia skupień — od najprostszych do coraz bardziej złożonych. Wyobraź sobie mnóstwo bąbelków unoszących się w wodzie i stopniowo łączących się ze sobą aż do powstania większej bańki powietrza. Na podobnej zasadzie w każdej iteracji algorytm aglomeracyjny łączy najbliższą parę skupień (począwszy od pojedynczych przykładów). Jeżeli rozrysujesz drzewo, którego każdym odgałęzieniem jest para scalonych skupień, otrzymasz drzewo binarne grup, w których poszczególne przykłady są symbolizowane jako liście. Algorytm ten wykrywa skupienia o różnych kształtach, generuje elastyczne i zawierające mnóstwo informacji drzewo skupień, co oznacza, że nie musisz wyznaczać określonej skali skupienia, a do tego możesz wyznaczyć dowolny wskaźnik odległości pomiędzy poszczególnymi parami. Skaluje się on elegancko do dużej liczby przykładów, jeżeli wyznaczysz macierz sąsiedztwa, czyli macierz rzadką o rozmiarze $m \times m$ wskazującą pary sąsiadujących ze sobą przykładów (np. jest ona zwracana przez metodę `sklearn.neighbors.kneighbors_graph()`). Bez macierzy sąsiedztwa algorytm nie skaluje się dobrze do dużych zestawów danych.

BIRCH

Algorytm BIRCH (ang. *Balanced Iterative Reducing and Clustering using Hierarchies* — zrównoważone, iteracyjne redukowanie i analiza skupień za pomocą hierarchii) został zaprojektowany z myślą o bardzo dużych zestawach danych i przy podobnych rezultatach może być szybszy od wsadowego algorytmu centroidów pod warunkiem, że liczba cech nie będzie zbyt duża (< 20). W fazie uczenia budowana jest drzewiasta struktura hierarchiczna zawierająca wystarczająco wiele informacji, aby można było szybko przydzielić każdy nowy przykład do skupienia bez konieczności przechowywania wszystkich przykładów w strukturze drzewa — technika ta pozwala korzystać z ograniczonych zasobów pamięci przy jednoczesnym przetwarzaniu olbrzymich zestawów danych.

Mean-Shift

Pierwszym etapem tego algorytmu jest umieszczenie okręgu wokół każdego przykładu (będącego środkiem tego okręgu), następnie zostaje obliczona średnia wszystkich przykładów znajdujących się w obrębie okręgu, po czym środek okręgu zostaje przesunięty w punkt wyznaczony przez tę średnią. Faza przesuwania do średniej jest powtarzana aż do momentu ustabilizowania okręgów (tzn. dopóki każdy z okręgów nie zostanie wyśrodkowany zgodnie ze średnią obejmowaną przez niego przykładów). Algorytm Mean-Shift przesuwa okręgi w kierunku obszarów o większej gęstości aż do chwili znalezienia przez każdy z nich maksimum lokalnego zagęszczenia. Do tego wszystkie przykłady, których okręgi zostały umieszczone w tym samym punkcie (lub w jego pobliżu), zostają przydzielone do tego samego skupienia. Znajdziemy tu pewne cechy wspólne z algorytmem DBSCAN, np. zdolność wyszukiwania dowolnej liczby skupień o różnych kształtach, niewielką liczbę hiperparametrów (a w zasadzie tylko jeden: promień okręgów, zwany **przepustowością**) oraz zależność od oszacowania gęstości lokalnej. W przeciwieństwie jednak do algorytmu DBSCAN, Mean-Shift ma tendencję do rozbijania skupień na

mniejsze składowe w przypadku ich wewnętrznego zróżnicowania gęstości. Niestety jego złożoność obliczeniowa wynosi $O(m^2)$, więc nie nadaje się do przetwarzania dużych zestawów danych.

Propagacja podobieństwa

Algorytm ten wykorzystuje system głosowania, w którym przykłady wybierają reprezentatywne próbki, a po osiągnięciu zbieżności każdy przykład reprezentatywny wraz z przykładami głosującymi na niego tworzą skupienie. Propagacja podobieństwa jest w stanie wykrywać dowolną liczbę skupień o różnych kształtach. Niestety złożoność obliczeniowa wynosi tu $O(m^2)$, więc algorytm ten również nie nadaje się do dużych zestawów danych.

Widmowa analiza skupień

Algorytm ten przyjmuje macierz podobieństwa przykładów i tworzy na jej podstawie małowy-miarowe odwzorowanie (tzn. redukuje wymiarowość), a następnie stosuje kolejną metodę analizy skupień (implementacja zawarta w module Scikit-Learn wykorzystuje algorytm centroidów). Widmowa analiza skupień jest w stanie wykrywać złożone struktury grup, a także służy do obcinania grafów (np. w celu identyfikowania skupień znajomych w sieciach społecznościowych). Nie skaluje się zbyt dobrze do dużej liczby przykładów i nie sprawdza się zbyt dobrze w sytuacji, gdy skupienia znacznie różnią się rozmiarami.

Przejdźmy do modeli mieszanin gaussowskich, których możemy używać do szacowania gęstości, analizy skupień i wykrywania anomalii.

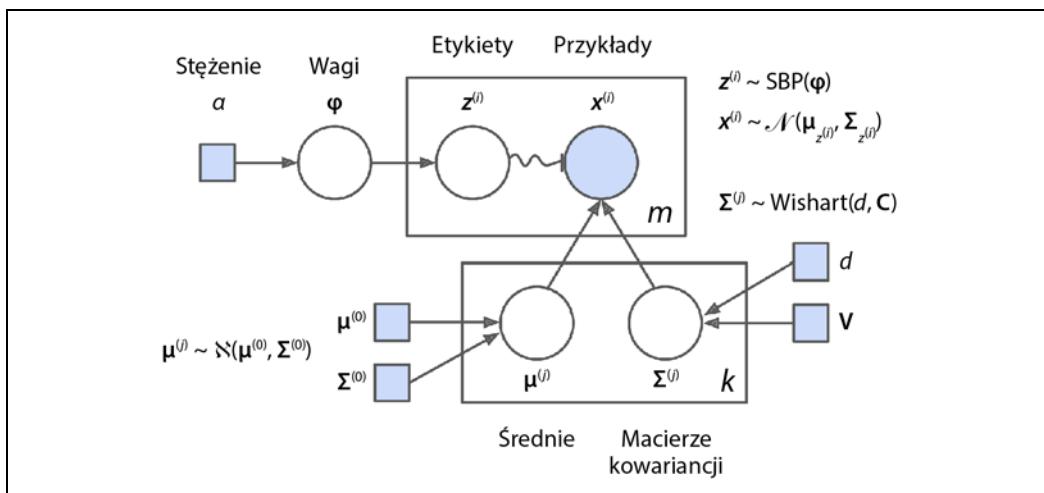
Mieszaniny gaussowskie

Model mieszaniny gaussowskiej (ang. *Gaussian Mixture Model* — GMM) jest modelem probabilistycznym, w którym zakładamy, że przykłady zostały wygenerowane z mieszaniny kilku rozkładów gaussowskich o nieznanych parametrach. Wszystkie przykłady wygenerowane z pojedynczego rozkładu gaussowskiego tworzą skupienie o kształcie najczęściej zbliżonym do eliptycznego. Każde skupienie może być elipsą o odmiennym kształcie, rozmiarze, gęstości i kierunku, co widać na rysunku 9.11. Obserwując przykład, wiesz, że został on wygenerowany z rozkładu gaussowskiego, ale nie wiesz z którego i nie znasz parametrów tego rozkładu.

- Istnieje kilka odmian modeli GMM. W najprostszym wariantie, zaimplementowanym w klasie `GaussianMixture`, musisz znać liczbę rozkładów gaussowskich k . Zakładamy tu, że zestaw danych X jest generowany zgodnie z następującym procesem probabilistycznym:
 - W przypadku każdego przykładu skupienie jest dobierane losowo z k różnych skupień. Prawdopodobieństwo dobrania j -tego skupienia jest określone przez $\phi^{(j)}$, wagę skupienia⁷. Indeks skupienia wybranego dla i -tego przykładu zapisujemy jako $z^{(i)}$.
 - Jeżeli $z^{(i)} = j$, oznacza to, że i -ty przykład został przydzielony do j -tego skupienia, położenie $\mathbf{x}^{(i)}$ przykładu jest losowo próbkiowane z rozkładu Gaussa o średniej $\boldsymbol{\mu}^{(j)}$ i macierzy kowariancji $\boldsymbol{\Sigma}^{(j)}$. Zapisujemy to następująco: $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$.

⁷ Fi (ϕ lub φ) jest 21. literą alfabetu greckiego.

Ten proces generatywny możemy zilustrować w postaci modelu graficznego. Rysunek 9.16 przedstawia strukturę zależności warunkowych pomiędzy zmiennymi losowymi.



Rysunek 9.16. Reprezentacja graficzna modelu mieszaniny gaussowskiej, włącznie z jego parametrami (kwadraty), zmiennymi losowymi (kółka) i ich wzajemnymi zależnościami warunkowymi (strzałki ciągłe)

Rysunek 9.16 należy interpretować następująco⁸:

- Kółka reprezentują zmienne losowe.
- Kwadraty symbolizują wartości ustalone (np. parametry modelu).
- Duże prostokąty to tzw. **ptyły** (ang. *plates*). Wskazują one, że ich zawartość jest kilkukrotnie powtarzana.
- Liczba w prawym dolnym rogu każdej płyty mówi, ile razy jej zawartość jest powtarzana. Istnieje zatem m zmiennych $z^{(i)}$ (od $z^{(1)}$ do $z^{(m)}$) i m zmiennych losowych $x^{(i)}$. Do tego występuje k średnich $\mu^{(i)}$ i k macierzy kowariancji $\Sigma^{(i)}$. Widzimy także, że dostępny jest tylko jeden wektor wag ϕ (zawierający wszystkie wagи od $\phi^{(1)}$ do $\phi^{(k)}$).
- Każda zmienna $z^{(i)}$ jest pozyskiwana z rozkładu kategorialnego z wagami ϕ . Każda zmienna $x^{(i)}$ jest pozyskiwana z rozkładu normalnego o średniej i macierzy kowariancji zdefiniowanej przez jej skupienie $z^{(i)}$.
- Strzałka ciągła symbolizuje zależności warunkowe. Na przykład rozkład prawdopodobieństwa dla każdej zmiennej losowej $z^{(i)}$ zależy od wektora wag ϕ . Zwróci uwagę, że jeżeli strzałka przekroczy granicę płyty, znaczy to, że zależność ta będzie stosowana wobec wszystkich powtórzeń definiowanych przez tę płytę. Przykładowo wektor wag ϕ warunkuje rozkłady prawdopodobieństwa wszystkich zmiennych losowych od $x^{(1)}$ do $x^{(m)}$.

⁸ Znaczna część zastosowanej tu notacji jest standardowa, ale wziętem kilka dodatkowych elementów z artykułu Wikipedii poświęconemu **notacji płytowej** (ang. *plate notation*; https://en.wikipedia.org/wiki/Plate_notation).

- Zyzkakowata strzałka wychodząca z $z^{(i)}$ do $x^{(i)}$ symbolizuje przełącznik: zależnie od wartości $z^{(i)}$ przykład $x^{(i)}$ będzie próbowany z innego rozkładu gaussowskiego. Na przykład jeżeli $z^{(i)} = j$, to $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{(j)}, \boldsymbol{\Sigma}^{(j)})$.
- Zacienione węzły wyznaczają znaną wartość. Zatem w naszym przypadku jedynie zmienne losowe $x^{(i)}$ mają znane wartości — są to tak zwane **zmienne obserwowane** (ang. *observed variables*). Nieznane zmienne losowe $z^{(i)}$ to **zmienne ukryte** (ang. *latent variables*).

Co więc możemy zrobić z takim modelem? Mając dany zestaw danych X , zazwyczaj chcemy najpierw oszacować wagi ϕ i wszystkie parametry rozkładów (od $\boldsymbol{\mu}^{(1)}$ do $\boldsymbol{\mu}^{(k)}$ i od $\boldsymbol{\Sigma}^{(1)}$ do $\boldsymbol{\Sigma}^{(k)}$). Znacznie ułatwia nam to klasa GaussianMixture:

```
from sklearn.mixture import GaussianMixture
gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Spójrzmy na parametry oszacowane przez algorytm:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717, 1.05933727],
       [-1.40763984, 1.42710194],
       [ 0.05135313, 0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
       [-0.03270354, 0.95496237]],
      [[ 0.63478101, 0.72969804],
       [ 0.72969804, 1.1609872 ]],
      [[ 0.68809572, 0.79608475],
       [ 0.79608475, 1.21234145]]])
```

Algorytm działa jak należy! Istotnie, wagи użyte do wygenerowania danych miały wartości 0,2, 0,4 i 0,4; również macierze średnich i kowariancji są zbliżone do wykrytych przez algorytm. Ale jak? Klasa ta bazuje na **algorytmie oczekiwania – maksymalizacji** (ang. *Expectation-Maximization* — EM), który w wielu aspektach przypomina algorytm centroidów: inicjalizuje losowo parametry skupień, następnie powtarza dwa etapy aż do uzyskania zbieżności. W pierwszym z nich przykłady są przydzielane do skupienia (**faza oczekiwania**), a w drugim skupienia zostają zaktualizowane (**faza maksymalizacji**). Brzmi znajomo, prawda? W kontekście analizy skupień możemy traktować algorytm EM jako uogólnienie algorytmu centroidów pozwalające wyszukiwać nie tylko środki skupień (od $\boldsymbol{\mu}^{(1)}$ do $\boldsymbol{\mu}^{(k)}$), lecz również ich kształty, rozmiary i kierunki (od $\boldsymbol{\Sigma}^{(1)}$ do $\boldsymbol{\Sigma}^{(k)}$), a także ich względne wagи (od $\phi^{(1)}$ do $\phi^{(k)}$). Jednak w przeciwieństwie do algorytmu centroidów, tutaj zastosowanie znajduje miękka analiza skupień, a nie twarda. W trakcie fazy oczekiwania dla każdego przykładu szacowane jest prawdopodobieństwo przynależności do każdej grupy (na podstawie bieżących parametrów skupienia). Następnie na etapie maksymalizacji każde skupienie jest aktualizowane za pomocą *wszystkich* przykładów z zestawu danych, gdzie każdy przykład jest ważony zgodnie z szacowanym prawdopodobieństwem jego przynależności do tej grupy. Prawdopodobieństwa te są nazywane **odpowiedzialnościami** (ang. *responsibility*) skupień względem przykładów.

W fazie maksymalizacji na aktualizację każdego skupienia będą najbardziej wpływać przykłady, za które jest ono najbardziej odpowiedzialne.



Niestety podobnie jak algorytm centroidów, algorytm EM ma również tendencję do uzyskiwania zbieżności z nieoptimalnymi rozwiązaniami, dlatego należy uruchamiać go kilkakrotnie i pozostawiać wyłącznie najlepsze rozwiązanie. Z tego powodu wyznaczamy w hiperparametrze `n_init` wartość 10. Pamiętaj: domyślnie wartość ta wynosi 1.

Możesz sprawdzić, czy algorytm uzyskał zbieżność, a jeśli tak, to również ile potrzebował iteracji do tego:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Po oszacowaniu położenia, rozmiaru, kształtu, kierunku i względnej wagi każdego skupienia model jest w stanie z łatwością przydzielić każdy przykład do najbardziej prawdopodobnego skupienia (twarda analiza skupień) lub oszacować jego przynależność do określonej grupy (miękką analizą skupień). Wystarczy wybrać metodę `predict()` dla grupowania twardego lub `predict_proba` dla miękkiej analizy skupień:

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

Model mieszaniny gaussowskiej jest **modelem generatywnym**, co oznacza, że możemy próbować z niego nowe przykłady (są one ułożone zgodnie z indeksem skupienia):

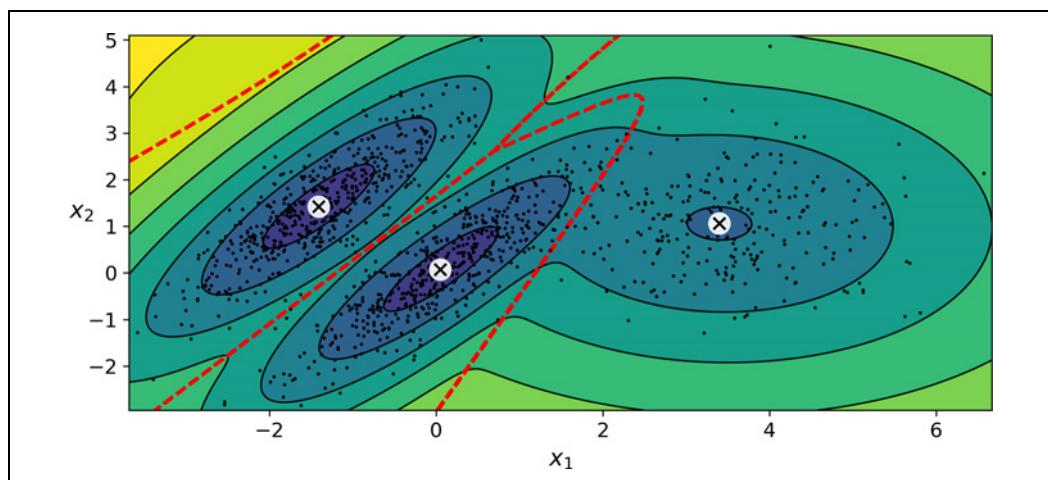
```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366 ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])
>>> y_new
array([0, 1, 2, 2, 2, 2])
```

Możliwe jest również oszacowanie gęstości modelu w dowolnym obszarze. Służy do tego metoda `score_samples()`, która oszacowuje dla każdego przekazanego przykładu logarytm **funkcji gęstości prawdopodobieństwa** (ang. *Probability Density Function* — PDF) w danym obszarze. Im wyższy wynik, tym większa gęstość:

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

Jeżeli obliczysz funkcję wykładniczą tych wyników, uzyskasz wartość funkcji PDF w obszarze tych przykładów. Nie są to prawdopodobieństwa, lecz **gęstości** prawdopodobieństwa — mogą one przyjmować dowolną wartość dodatnią, nie ograniczają się wyłącznie do wartości 0 i 1. Aby oszacować prawdopodobieństwo przynależności przykładu do danego rejonu, należy przecałkować tę funkcję po danym obszarze (gdybyśmy przecałkowali ją po całym dostępnym rejonie przykładów, otrzymalibyśmy wartość 1).

Rysunek 9.17 pokazuje wartości średnie skupień, granice decyzyjne (linie przerywane) i kontury gęstości w tym modelu.



Rysunek 9.17. Wartości średnie skupień, granice decyzyjne i kontury gęstości w wytrenowanym modelu mieszanin gaussowskich

Ładnie! Wyraźnie widać, że algorytm znalazł znakomite rozwiązanie. Oczywiście znacznie ułatwiliśmy zadanie poprzez wygenerowanie danych za pomocą szeregu dwuwymiarowych rozkładów gaussowskich (niestety rzeczywiste dane nie zawsze mają rozkład gaussowski i mało wymiarów). Wyznaczyliśmy również właściwą liczbę skupień. W przypadku wielu skupień, wielu wymiarów lub małej liczby przykładów algorytm EM może mieć problem z uzyskaniem optymalnego rozwiązania. Musielibyśmy zmniejszyć trudność zadania poprzez ograniczenie liczby parametrów, które algorytm musi poznać. Jednym ze sposobów osiągnięcia tego jest ograniczenie zakresu kształtów i kierunków, jakie może przyjąć dane skupienie. Wystarczy narzuścić ograniczenia na macierze kowariancji. W tym celu wyznacz hiperparametrowi covariance_type jedną z następujących wartości:

"spherical"

Wszystkie skupienia muszą być sferyczne, ale mogą różnić się średnicą (tzn. wariancjami).

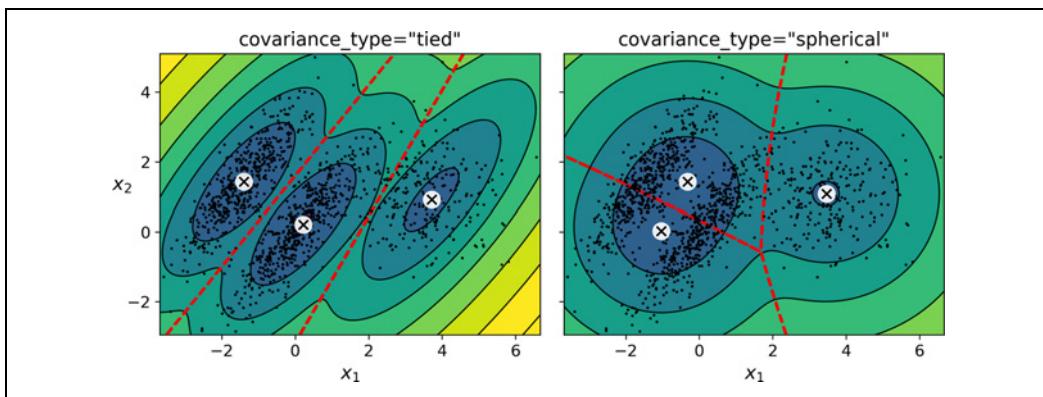
"diag"

Skupienia mogą przyjmować dowolny kształt eliptyczny o dowolnym rozmiarze, ale ich osie muszą być ułożone równolegle do osi współrzędnych (tzn. macierze kowariancji muszą być diagonalne).

"tied"

Wszystkie skupienia muszą mieć taki sam eliptyczny kształt, rozmiar i kierunek (tzn. wszystkie skupienia mają taką samą macierz kowariancji).

Domyślną wartością hiperparametru covariance_type jest "full", co oznacza, że każde skupienie może przyjąć dowolny kształt, rozmiar i kierunek (każda grupa ma własną, nieograniczoną macierz kowariancji). Na rysunku 9.18 widzimy rozwiązania wyszukane przez algorytm EM dla wartości "tied" i "spherical".



Rysunek 9.18. Mieszanki gaussowskie dla skupień powiązanych (po lewej) i sferycznych (po prawej)



Złożoność obliczeniowa uczenia modelu GaussianMixture zależy od liczby przykładów m , liczby wymiarów n , liczby skupień k i ograniczeń macierzy kowariancji. Jeżeli hiperparametr covariance_type ma wartość "spherical" lub "diag", to złożoność obliczeniowa wynosi $O(kmn)$ przy założeniu, że dane tworzą skupienia. W przypadku wartości "tied" lub "full" przyjmuje ona postać $O(kmn^2 + kn^3)$, zatem nie będzie się skalować zbyt dobrze do dużej liczby cech.

Modele mieszanych gaussowskich można również stosować do wykrywania anomalii. Pokażę teraz, jak to robić.

Wykrywanie anomalii za pomocą mieszanych gaussowskich

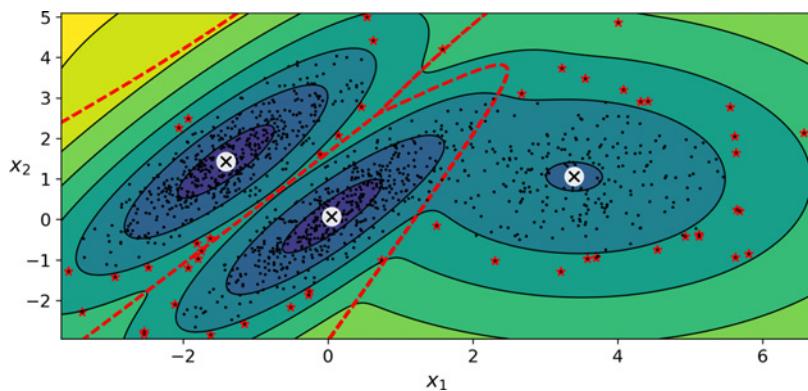
Wykrywanie anomalii (zwane także **wykrywaniem elementów odstających**) to wyszukiwanie przykładów, które znacznie odbiegają od normy. Przykłady te są nazywane **anomaliami** lub **elementami odstającymi** (ang. *outlier*), natomiast zwykłe przykłady to **elementy typowe** (ang. *inlier*). Wykrywanie anomalii przydaje się w licznych zastosowaniach, takich jak wykrywanie oszustw finansowych, rozpoznawanie wadliwych elementów na etapie produkcji czy usuwanie elementów od-

stających z zestawu danych przed przystąpieniem do uczenia innego modelu (co może znacznie poprawić jego skuteczność).

Wykorzystanie modelu mieszanin gaussowskich do wykrywania anomalii jest dość prostym zadaniem: każdy przykład znajdujący się w obszarze o małej gęstości jest uznawany za element odstający. Trzeba wyznaczyć wartość progową gęstości. Na przykład w firmie wytwarzającej części, która próbuje wykrywać wadliwe egzemplarze, odsetek wadliwości jest zazwyczaj znany. Powiedzmy, że wynosi on 4%. Wyznaczamy w takim przypadku wartość progową gęstości, w wyniku której zostanie wyznaczone 4% przykładów znajdujących się w obszarach poniżej gęstości progowej. Jeżeli uzyskasz zbyt wiele przykładów fałszywie pozytywnych (tzn. zupełnie sprawnych produktów, które zostały niewłaściwie oznaczone jako wadliwe), to możesz obniżyć wartość progową. Z kolei w przypadku przykładów fałszywie negatywnych (produktów wadliwych, które zostały oznaczone jako prawidłowe), możesz zwiększyć wartość progową. Mamy tu do czynienia z klasycznym kompromisem pomiędzy precyją a pełnością (zob. rozdział 3.). Oto sposób wykrywania elementów odstających, gdzie jako wartość progowa zostanie użyty czwarty percentyl najmniejszej gęstości (tzn. mniej więcej 4% przykładów zostanie oznaczonych jako anomalie):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

Na rysunku 9.19 anomalie te zostały oznaczone gwiazdkami.



Rysunek 9.19. Wykrywanie anomalii za pomocą modelu mieszanin gaussowskich

Blisko powiązanym zadaniem jest **wykrywanie nowości** (ang. *novelty detection*), które różni się od algorytmu wykrywania anomalii tym, że zakładamy tu uczenie modelu na „czystym” zestawie danych (pozbawionym elementów odstających). W rzeczywistości, algorytm wykrywania anomalii często służy do oczyszczania zestawu danych.

Podobnie jak w przypadku algorytmu centroidów, klasa GaussianMixture wymaga określenia liczby skupień. Jak więc ją wyznaczamy?



Modele mieszanin gaussowskich starają się dopasowywać do wszystkich danych, w tym do elementów odstających, dlatego jeżeli występuje ich zbyt wiele w zestawie danych, mogą wypaczyć „normalną” perspektywę i niektóre anomalie mogą być uznawane za elementy typowe. W takiej sytuacji możesz spróbować raz dopasować model, wykorzystać go do wykrycia i usunięcia najbardziej skrajnych elementów odstających, a potem ponownie wytrenować model na oczyszczonym zestawie danych. Innym rozwiązaniem jest użycie odpornych metod szacowania kowariancji (zob. klasę `EllipticEnvelope`).

Wyznaczanie liczby skupień

W przypadku algorytmu centroidów możesz skorzystać z bezwładności lub wyniku profilu do wybrania właściwej liczby skupień. Wskaźniki te są jednak bezużyteczne podczas używania modelu mieszanin gaussowskich, ponieważ ich wyniki okazują się niezbyt rzetelne, gdy skupienia nie są sferyczne lub różnią się rozmiarami. Możesz natomiast wyszukać model minimalizujący **kryterium informacji teoretycznej**, takie jak **bayesowskie kryterium informacyjne** (ang. *Bayesian Information Criterion* — BIC) lub **kryterium informacyjne Akaikego** (ang. *Akaike Information Criterion* — AIC) (równanie 9.1).

Równanie 9.1. Bayesowskie kryterium informacyjne (BIC) i kryterium informacyjne Akaikego (AIC)

$$BIC = \log(m)p - 2\log(\hat{L})$$

$$AIC = 2p - 2\log(\hat{L})$$

W tych równaniach:

- m , jak zwykle, jest liczbą przykładów.
- p jest liczbą parametrów, których wyuczył się model.
- \hat{L} jest maksymalizowaną wartością **funkcji wiarygodności** modelu.

Zarówno kryteria BIC, jak i AIC karzą modele, które muszą poznać więcej parametrów (np. skupień), a nagradzają te, które dobrze dopasowują się do danych. Obydwie kategorie często wybierają ten sam model. W przypadku różnic model wybrany przez kryterium BIC zazwyczaj jest prostszy (mniej parametrów), ale nie dopasowuje się tak dobrze do danych jak model wyznaczony przez kryterium AIC (jest to prawdziwe szczególnie w przypadku dużych zestawów danych).

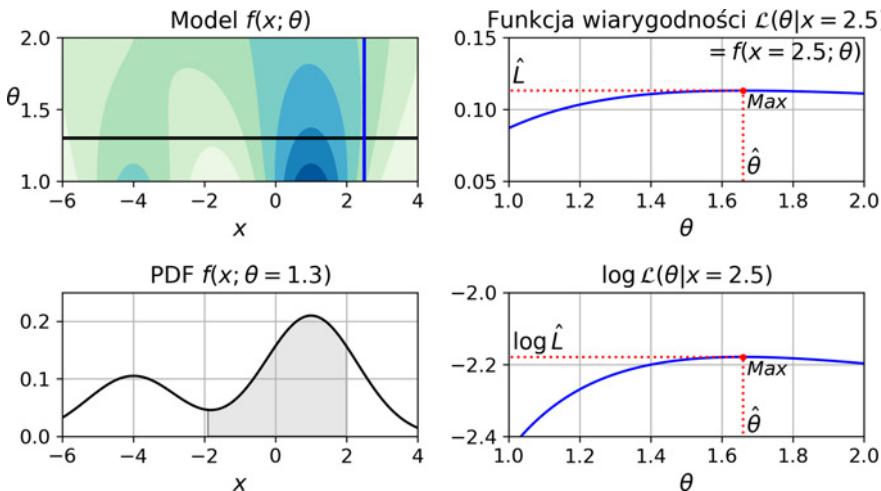
Aby obliczyć kryteria BIC i AIC, wywołaj metody, odpowiednio, `bic()` i `aic()`:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

Funkcja wiarygodności

Gdy mamy dany model statystyczny z jakimiś parametrami θ , słowo „prawdopodobieństwo” (ang. *probability*) opisuje sensowność przyszłego wyniku x (przy znajomości wartości parametrów θ), natomiast słowo „wiarygodność” (ang. *likelihood*) określa sensowność określonego zestawu wartości parametrów θ już po poznaniu wyniku x .

Wyobraź sobie jednowymiarowy model dwóch rozkładów gaussowskich wyśrodkowanych w punktach -4 i 1 . Dla uproszczenia model ten zawiera tylko jeden parametr θ regulujący odchylenie standardowe obydwu rozkładów. Lewy górny wykres na rysunku 9.20 prezentuje cały model $f(x; \theta)$ jako funkcję zarówno x , jak i θ . Aby oszacować rozkład prawdopodobieństwa przyszłego wyniku x , musimy wyznaczyć parametr θ modelu. Na przykład jeżeli wyznaczysz $\theta = 1,3$ (linia pozioma), otrzymasz funkcję gęstości prawdopodobieństwa $f(x; \theta = 1,3)$, widoczną na lewym dolnym wykresie. Powiedzmy, że chcesz oszacować prawdopodobieństwo występowania x w przedziale od -2 do 2 . Musisz obliczyć całkę z funkcji PDF dla tego przedziału (tzn. powierzchnię zacienionego obszaru). Co jednak w sytuacji, gdy nie znamy wartości θ , ale zaobserwowałyśmy pojedynczy przykład $x = 2,5$ (pionowa linia na lewym górnym wykresie)? W takim przypadku otrzymujesz funkcję wiarygodności $\mathcal{L}(\theta | x = 2,5) = f(x = 2,5; \theta)$, pokazaną na prawym górnym wykresie.



Rysunek 9.20. Funkcja parametryczna modelu (lewy górny wykres), a także niektóre funkcje pochodne: PDF (lewy dolny), funkcja wiarygodności (prawy górny) i logarytmiczna funkcja wiarygodności (prawy dolny)

Mówiąc krótko, PDF jest funkcją x (przy stałym parametrze θ), natomiast funkcja wiarygodności jest funkcją θ (przy stałej wartości x). Bardzo ważne jest zrozumienie, że funkcja wiarygodności **nie** jest rozkładem prawdopodobieństwa: jeżeli przecałkujesz rozkład prawdopodobieństwa po wszystkich możliwych wartościach x , zawsze otrzymasz 1; jeśli jednak przecałkujesz funkcję prawdopodobieństwa po wszystkich możliwych wartościach θ , to w rezultacie możesz uzyskać dowolną wartość dodatnią.

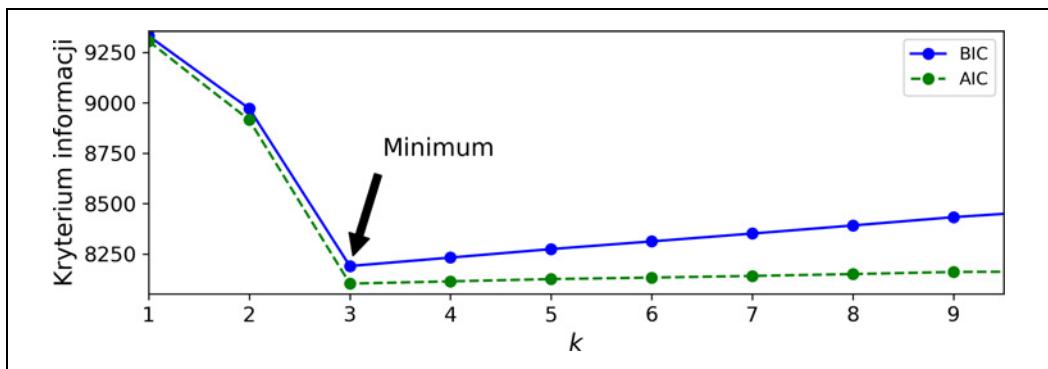
Przy danym zestawie danych X popularnym zadaniem jest oszacowanie najbardziej wiarygodnych wartości parametrów modelu. W tym celu musisz określić wartości maksymalizujące funkcję wiarygodności przy danym zestawie danych X . W omawianym przykładzie, jeżeli zaobserwowałaś/zaobserwowałeś pojedyncze wystąpienie $x = 2,5$, **oszacowanie maksymalnej wiarygodności** (ang. *Maximum Likelihood*

Estimate — MLE) parametru θ wynosi $\hat{\theta} = 1,5$. Jeżeli istnieje rozkład prawdopodobieństwa wstępnego g po θ , to można wziąć go pod uwagę poprzez maksymalizację $\mathcal{A}(\theta|x)g(\theta)$ zamiast maksymalizowania $\mathcal{A}(\theta|x)$. Jest to tzw. **oszacowanie maksymalne a posteriori** (ang. *Maximum A-Posteriori* — MAP). Ogranicza ono wartości parametru, dlatego możemy traktować je jako regularyzowaną wersję oszacowania MLE.

Zwróć uwagę, że maksymalizacja funkcji wiarygodności jest równoważna maksymalizacji jej logarytmu (prawy dolny wykres na rysunku 9.20). Istotnie, logarytm jest funkcją silnie rosnącą, zatem jeżeli θ maksymalizuje logarytm wiarygodności, to maksymalizuje również samą funkcję wiarygodności. Okazuje się, że zasadniczo łatwiej jest maksymalizować logarytm wiarygodności. Jeżeli na przykład zaobserwowałaszt/zaobserwowałeś kilka niezależnych wystąpień $x^{(1)}$ do $x^{(m)}$, będziesz musiał/musiał znaleźć wartość θ maksymalizującą iloczyn poszczególnych funkcji wiarygodności. To samo jednak (i znacznie prościej) osiągniemy, maksymalizując sumę (nie iloczyn) funkcji logarytmu wiarygodności dzięki cudownym własnościom logarytmowania, które sprawiają, że iloczyn zostaje przekształcony w sumowanie: $\log(ab) = \log(a)+\log(b)$.

Po uzyskaniu oszacowania $\hat{\theta}$, czyli wartości θ maksymalizującej funkcję wiarygodności, możesz przystąpić do obliczenia $\hat{L} = \mathcal{L}(\hat{\theta}, \mathbf{X})$, która jest wartością służącą do obliczenia kryteriów BIC i AIC; można ją traktować jako miarę dopasowania modelu do danych.

Na rysunku 9.21 zaprezentowałem kryterium BIC dla różnych liczb skupień k . Jak widać, obydwa kryteria osiągają minimum dla $k = 3$, zatem jest to prawdopodobnie najlepszy wybór. Zwróć uwagę, że przeanalizowaliśmy także najlepszą wartość hiperparametru covariance_type. Na przykład jeśli wyznaczmy wartość "spherical" zamiast "full", to model będzie musiał poznać znacznie mniej parametrów, ale nie będzie tak dobrze dopasowany do danych.



Rysunek 9.21. Kryteria BIC i AIC dla różnych liczb skupień k

Modele bayesowskie mieszanin gaussowskich

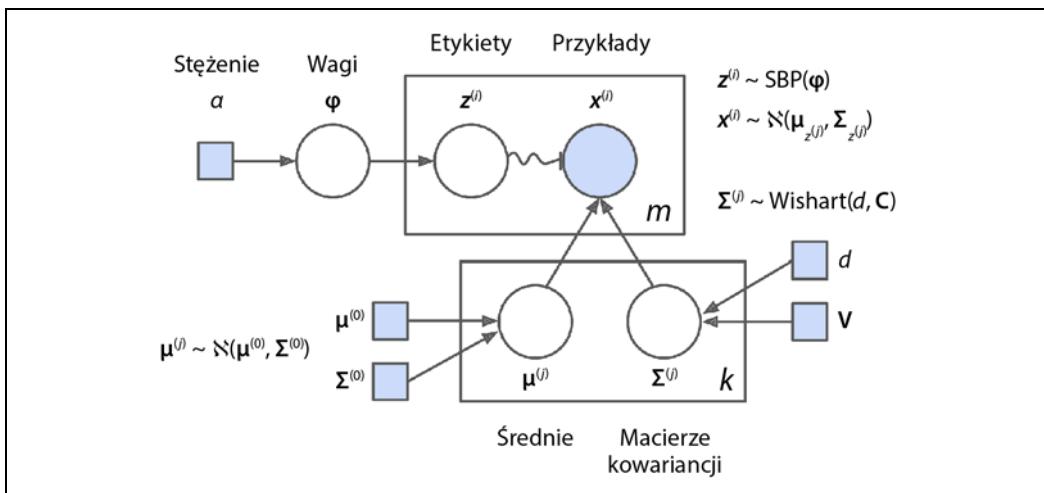
Zamiast własnoręcznie wyszukiwać optymalną liczbę skupień możesz użyć klasy BayesianGaussianMixture, która jest w stanie nadawać wartości równe lub zbliżone do 0 niepotrzebnym skupieniom. Wyznacz liczbę skupień `n_clusters` taką, jaką uznajesz za większą od wartości optymalnej (zakładamy w ten sposób minimalną wiedzę na temat problemu), a algorytm automatycznie wy-

eliminuje niepotrzebne skupienia. Wyznaczmy na przykład liczbę skupień równą 10 i zobaczymy, co się stanie:

```
>>> from sklearn.mixture import BayesianGaussianMixture  
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)  
>>> bgm.fit(X)  
>>> np.round(bgm.weights_, 2)  
array([0.4 , 0.21, 0.4 , 0. , 0. , 0. , 0. , 0. , 0. , 0.])
```

Doskonale: algorytm automatycznie odkrył, że potrzebne są wyłącznie trzy skupienia, które są niemal identyczne z rezultatem widocznym na rysunku 9.17.

W modelu tym parametry skupień (w tym macierze wag, średnich i kowariancji) nie są traktowane jako ustalone parametry modelu, lecz jako ukryte zmienne losowe, podobnie do przydziałów skupień (rysunek 9.22). Zatem z przechowuje teraz zarówno parametry, jak i przydziały skupień.

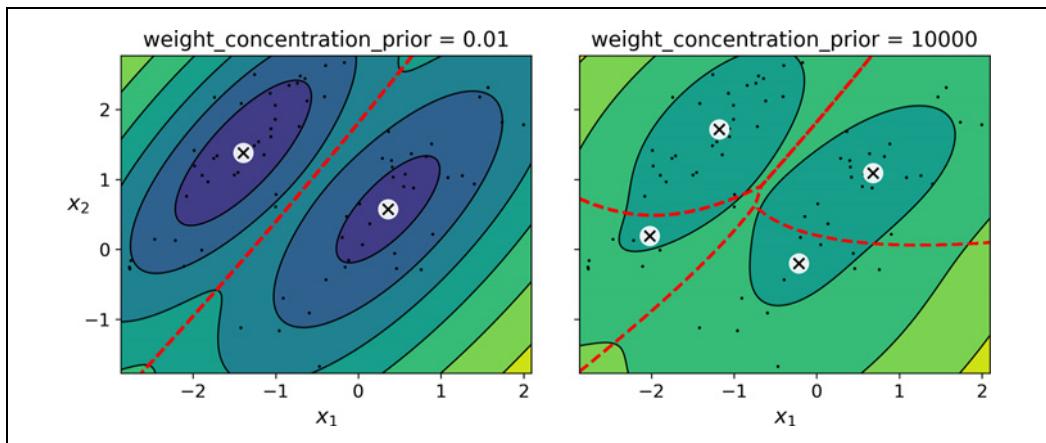


Rysunek 9.22. Model bayesowski mieszaniny gaussowskiej

Do modelowania zmiennych losowych, których wartości mieścią się w stałym zakresie, często wykorzystujemy rozkład beta. W tym przypadku zakres ten ma postać od 0 do 1. Tak zwaną konstrukcję odcinania przedziałów „stick breaking” Sethuramana najlepiej wyjaśnić na przykładzie: założymy, że $\Phi = [0.3, 0.6, 0.5, \dots]$, a wówczas 30% przykładów zostanie przydzielonych do skupienia 0., 60% pozostałych przykładów trafi do skupienia 1., 50% reszty przykładów trafi do skupienia 2. itd. Proces ten stanowi dobry model dla zestawów danych, w których nowe przykłady mają większe prawdopodobieństwo trafienia do dużego skupienia (np. ludzie mają tendencję do przeprowadzania się do dużych miast). Jeżeli stężenie α jest duże, to wartości Φ będą zbliżone do 0, a proces SBP wygeneruje dużo skupień. Z kolei jeżeli stężenie będzie małe, to wartości Φ będą dążyć do 1, co spowoduje wygenerowanie mniejszej liczby skupień. Do tego macierze kowariancji są próbkiowane za pomocą rozkładu Wisharta — parametry d i V regulują kształt skupień.

Uprzednią wiedzę na temat zmiennych ukrytych z można zakodować w rozkładzie prawdopodobieństwa $p(z)$ zwany **prawdopodobieństwem apriorycznym (zaczątkowym)** (ang. *prior*). Na przykład możemy być przekonani a priori, że najprawdopodobniej będzie tylko kilka skupień (małe

stężenie) lub że będzie ich dużo (duże stężenie). Takie przekonanie a priori dotyczące liczby skupień można dostosować za pomocą hiperparametru `weight_concentration_prior`. Wartości 0,01 i 10 000 dają zupełnie inne skupienia (rysunek 9.23). Jednak im większą ilością danych dysponujemy, tym mniej istotne staje się prawdopodobieństwo aprioryczne. W rzeczywistości aby tworzyć diagramy o tak dużych różnicach, należy korzystać z bardzo silnych prawdopodobieństw apriorycznych i niewielkiej ilości danych.



Rysunek 9.23. Wyznaczenie różnych prawdopodobieństw apriorycznych stężenia na tych samych danych skutkuje otrzymaniem różnej liczby skupień

Twierdzenie Bayesa (równanie 9.2) określa sposób aktualizowania rozkładu prawdopodobieństwa dla zmiennych losowych po zaobserwowaniu jakichś danych X . Oblicza ono **rozkład aposterioryczny** (ang. *posterior*) $p(z|X)$, czyli prawdopodobieństwo warunkowe z dla danych X .

Równanie 9.2. Twierdzenie Bayesa

$$p(z|X) = \text{rozk. a posteriori} = \frac{\text{wiarygodność} \times \text{rozk. a priori}}{\text{dowód}} = \frac{p(X|z)p(z)}{p(X)}$$

Niestety w modelu mieszanin gaussowskich (i wielu innych problemach) mianownik $p(X)$ jest trudny do obliczenia, ponieważ wymaga przecałkowania po wszystkich możliwych wartościach z (równanie 9.3), co oznaczałoby wzięcie pod uwagę wszystkich możliwych kombinacji parametrów skupień i przydziałów skupień.

Równanie 9.3. Dowód $p(X)$ jest często trudny do obliczenia

$$p(X) = \int p(X|z)p(z)dz$$

Ta trudność stanowi jeden z głównych problemów statystyki bayesowskiej, ale istnieje kilka sposobów jego rozwiązania. Jednym z nich jest **wnioskowanie wariacyjne** (ang. *variational inference*), w którym wybierana jest rodzina rozkładów $q(z; \lambda)$ zawierająca własne parametry wariacyjne λ (lambda), a następnie przeprowadzana jest optymalizacja tych parametrów tak, aby $q(z)$ stanowiła dobre przybliżenie $p(z|X)$. Osiągamy to, wyszukując taką wartość λ , która minimalizuje rozbieżność KL

od $q(z)$ do $p(z|X)$, co zapisujemy jako $D_{KL}(q||p)$. Równanie rozbieżności KL zostało zaprezentowane w równaniu 9.4 — możemy je zapisać jako różnicę logarytmu dowodu ($\log p(X)$) i **ograniczenia dolnego dowodu** (ang. *Evidence Lower Bound* — ELBO). Logarytm dowodu nie zależy od q , zatem jest to człon stały, przez co w celu zminimalizowania rozbieżności KL wystarczy zmaksymalizować ELBO.

Równanie 9.4. Rozbieżność KL od $q(z)$ do $p(z|X)$

$$\begin{aligned}
 D_{KL}(q||p) &= \mathbb{E}_q \left[\log \frac{q(z)}{p(z|X)} \right] = \\
 &= \mathbb{E}_q \left[\log q(z) - \log p(z|X) \right] = \\
 &= \mathbb{E}_q \left[\log q(z) - \log \frac{p(z,X)}{p(X)} \right] = \\
 &= \mathbb{E}_q \left[\log q(z) - \log p(z,X) + \log p(X) \right] = \\
 &= \mathbb{E}_q \left[\log q(z) \right] - \mathbb{E}_q \left[\log p(z,X) \right] + \mathbb{E}_q \left[\log p(X) \right] = \\
 &= \mathbb{E}_q \left[\log p(X) \right] - \left(\mathbb{E}_q \left[\log p(z,X) \right] - \mathbb{E}_q \left[\log q(z) \right] \right) = \\
 &= \log p(X) - ELBO
 \end{aligned}$$

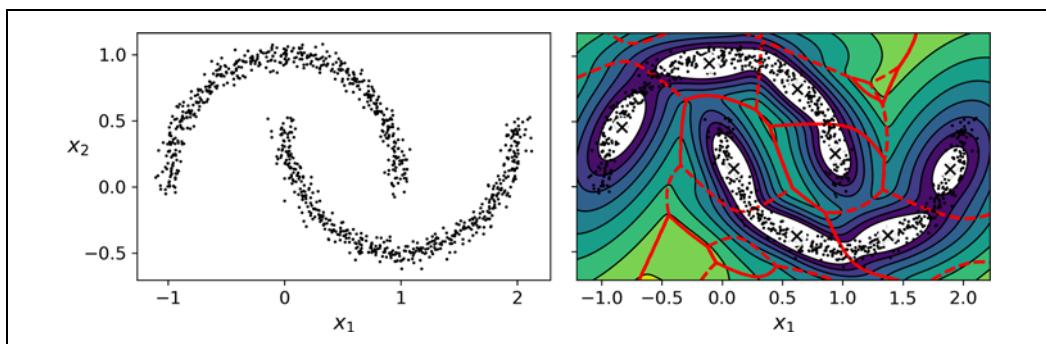
gdzie $ELBO = \mathbb{E}_q \left[\log p(z,X) \right] - \mathbb{E}_q \left[\log q(z) \right]$

W praktyce istnieją różne techniki maksymalizowania ELBO. W **średniopolowym wnioskowaniu wariacyjnym** (ang. *mean field variational inference*) należy bardzo ostrożnie wybrać rodzinę rozkładów $q(z; \lambda)$ i prawdopodobieństwo aprioryczne $p(z)$, aby mieć pewność, że równanie ELBO zostanie uproszczone do formy obliczalnej. Niestety nie istnieje ogólny mechanizm tego procesu. Wybór właściwej rodziny rozkładów i prawdopodobieństwa apriorycznego zależy od zadania i wymaga wiedzy matematycznej. Na przykład rozkłady i równania ograniczenia dolnego zastosowane w klasie `BayesianGaussianMixture` zostały opisane w dokumentacji (<https://scikit-learn.org/0.19/modules/dp-derivation.html>). Dzięki tym równaniom możliwe jest wyprowadzenie wzorów aktualizacji parametrów i zmiennych przydziałów — zostają one następnie użyte w sposób zbliżony do algorytmu oczekiwania – maksymalizacji. Istotnie, złożoność obliczeniowa klasy `BayesianGaussianMixture` jest podobna do złożoności klasy `GaussianMixture` (ale zasadniczo okazuje się znacznie wolniejsza). Prostszym sposobem maksymalizacji ELBO jest **czarnoskrzynkowe stochastyczne wnioskowanie wariacyjne** (ang. *Black Box Stochastic Variational Inference* — BBSVI): w każdej iteracji zostaje pobranych kilka przykładów z q , które zostają następnie użyte do oszacowania gradientów ELBO w odniesieniu do parametrów wariacyjnych λ wykorzystanych w fazie wznoszenia gradientów. W ten sposób wnioskowanie bayesowskie może być stosowane z dowolnym modelem (pod warunkiem, że jest różniczkowalny), nawet z głębokimi sieciami neuronowymi — połączenie wnioskowania bayesowskiego z głębokimi sieciami neuronowymi jest nazywane bayesowskim uczeniem głębokim.



Jeżeli interesują Cię szczegóły statystyki bayesowskiej, zajrzyj do książki *Bayesian Data Analysis* (<http://www.stat.columbia.edu/~gelman/book/>) Andrew Gelmana i in. (Chapman & Hall).

Modele mieszanin gaussowskich sprawują się doskonale w przypadku skupień o eliptycznych kształtach, jeżeli jednak spróbujesz dopasować zestaw danych o innym kształcie, mogą czekać Cię niemiłe niespodzianki. Zobaczmy na przykład, co się stanie, jeżeli użyjemy bayesowskiego modelu mieszanin gaussowskich analizy skupień w zestawie danych sierpowatych (rysunek 9.24).



Rysunek 9.24. Dopasowanie modelu mieszanin gaussowskich do skupień nieeliptycznych

Ups! Algorytm uparł się, żeby znaleźć elipsy, dlatego zamiast dwóch różnych skupień wyznaczył ich aż osiem. Oszacowanie gęstości jest nienajgorsze, zatem model ten mógłby zostać użyty do wykrywania anomalii, nie był jednak w stanie rozpoznać dwóch sierpow. Spójrzmy teraz na kilka algorytmów analizy skupień radzących sobie z grupami mającymi dowolny kształt.

Inne algorytmy służące do wykrywania anomalii i nowości

Moduł Scikit-Learn zawiera również inne algorytmy wyspecjalizowane w wykrywaniu anomalii lub nowości:

Analiza PCA (a także inne techniki redukowania wymiarowości wykorzystujące metodę `inverse_transform()`)

Jeżeli porównasz błąd rekonstrukcji normalnego przykładu z błędem rekonstrukcji anomalii, to jego wartość w tym drugim przypadku bywa zazwyczaj znacznie większa. Jest to prosta i często całkiem skuteczna metoda wykrywania anomalii (w jednym z ćwiczeń umieszczonych na końcu rozdziału poznasz praktyczne zastosowanie tego rozwiązania).

Fast-MCD (ang. Minimum Covariance Determinant — minimalizacja wyznacznika macierzy kowariancji)

Algorytm ten, zaimplementowany w klasie `EllipticEnvelope`, przydaje się do wykrywania elementów odstających, szczególnie podczas oczyszczania zestawu danych. Zakładamy, że normalne przykłady (elementy typowe) są generowane z jednego rozkładu gaussowskiego (nie z ich mieszaniny). Zakładamy także, że zestaw danych zawiera elementy odstające, które nie zostały wygenerowane z tego rozkładu. Gdy algorytm oszacowuje parametry rozkładu gaussowskiego (tzn.

kształt zarysu eliptycznego otaczającego elementy typowe), stara się on ostrożnie ignorować przykłady będące najprawdopodobniej elementami odstającymi. Technika ta zapewnia lepsze oszacowanie eliptycznej obwiedni, dzięki czemu algorytm lepiej sobie radzi z wykrywaniem elementów odstających.

Las izolacyjny

Jest to skuteczny algorytm wykrywania anomalii, zwłaszcza w wielowymiarowych zestawach danych. Tworzony jest w nim las losowy, w którym każde drzewo decyzyjne rozrasta się stochastycznie: w każdym węźle wybierana jest losowa cecha, następnie losowa wartość progowa (w zakresie pomiędzy wartością minimalną a maksymalną) w celu rozdzielenia zestawu danych na pół. Zestaw danych jest w ten sposób stopniowo rozdrabniany aż do momentu odizolowania każdego przykładu od pozostałych. Anomalie zazwyczaj znajdują się w dużej odległości od pozostałych przykładów, zatem przeciętnie (w kontekście wszystkich drzew decyzyjnych) zostają one odizolowane kilka przebiegów wcześniej od elementów typowych.

Lokalny współczynnik elementów odstających (ang. Local Outlier Factor — LOF)

Również ten algorytm dobrze sobie radzi z wykrywaniem elementów odstających. Porównuje zagęszczenie przykładów znajdujących się wokół danego elementu z zagęszczeniem wokół jego sąsiadów. Anomalia często jest bardziej odizolowana od jej k sąsiadów.

Jednoklasowa maszyna wektorów nośnych

Algorytm ten lepiej nadaje się do wykrywania nowości. Jak zapewne pamiętasz, kernelizowany klasyfikator SVM rozdziela dwie klasy najpierw poprzez rzutowanie (niejawnie) wszystkich przykładów na przestrzeń wielowymiarową, a następnie ich rozdzielenie w tej przestrzeni za pomocą liniowej maszyny wektorów nośnych (zob. rozdział 5.). Dysponujemy tylko jedną klasą przykładów, dlatego algorytm jednoklasowej maszyny SVM stara się oddzielić przykłady w przestrzeni wielowymiarowej od tej klasy. W pierwotnej przestrzeni oznacza to wyznaczenie małego obszaru obejmującego wszystkie przykłady. Jeżeli nowy element nie znajduje się w tym obszarze, to stanowi anomalię. Należy tu dostroić kilka hiperparametrów: tradycyjne sterujące kernelizowaną maszyną SVM, a do tego hiperparametr marginesu odpowiadający prawdopodobieństwu nieprawidłowego zaklasyfikowania przykładu jako nowości, podczas gdy okazuje się elementem typowym. Algorytm ten sprawdza się wspaniale, zwłaszcza w przypadku wielowymiarowych zestawów danych, ale podobnie jak w przypadku wszystkich maszyn wektorów nośnych nie skaluje się zbyt dobrze z dużymi zestawami danych.

Ćwiczenia

1. Jak zdefiniujesz analizę skupień? Możesz wymienić kilka algorytmów grupowania?
2. Wymień główne zastosowania algorytmów analizy skupień.
3. Opisz dwie techniki doboru właściwej liczby skupień podczas stosowania algorytmu centroidów.
4. Czym jest propagacja etykiet? Dlaczego i w jaki sposób należy ją implementować?
5. Wymień dwa algorytmy analizy skupień skalujące się do dużych zestawów danych, a także dwa algorytmy poszukujące obszarów o dużej gęstości.

- 6.** Podaj sytuację, w której może się przydać uczenie aktywne. Jak można je zaimplementować?
- 7.** Jaka jest różnica między wykrywaniem anomalii a wykrywaniem nowości?
- 8.** Czym jest mieszanina gaussowska? W jakich zadaniach można ją wykorzystać?
- 9.** Wymień dwie techniki wyszukiwania właściwej liczby skupień podczas korzystania z modelu mieszanin gaussowskich.
- 10.** Klasyczny zestaw danych Olivetti zawiera 400 czarno-białych zdjęć twarzy o wymiarach 64×64 . Każdy obraz został spłaszczony do jednowymiarowego wektora o rozmiarze 4096. Sfotografowano dziesięć różnych osób (każdą dziesięć razy) i tradycyjnym zadaniem jest wyuczenie modelu tak, aby rozpoznawał osobę widoczną na prezentowanym zdjęciu. Wczytaj omawiany zestaw danych za pomocą funkcji `sklearn.datasets.fetch_olivetti_faces()`, następnie podziel go na zbiory uczący, walidacyjny i testowy (zwróć uwagę, że zestaw ten jest już przeskalowany w zakresie od 0 do 1). Zestaw danych jest dość mały, dlatego prawdopodobnie zechcesz zastosować losowanie warstwowe, aby zapewnić taką samą liczbę zdjęć każdej osoby we wszystkich podzbiorach danych. Następnie przeprowadź analizę skupień za pomocą algorytmu centroidów tak, aby uzyskać odpowiednią liczbę skupień (za pomocą jednej z technik omówionych w tym rozdziale). Stwórz wizualizację skupień. Czy każde skupienie zawiera podobne twarze?
- 11.** Korzystając z zestawu danych Olivetti, wytrenuj klasyfikator w przewidywaniu osób widocznych na zdjęciach i zweryfikuj go na zbiorze walidacyjnym. Następnie użyj algorytmu centroidu jako narzędzia redukowania wymiarowości i wyucz model na tak zmniejszonym zestawie danych. Wyszukaj liczbę skupień umożliwiającą klasyfikatorowi uzyskanie maksymalnej wydajności. Jaką skuteczność jesteś w stanie osiągnąć? Co się stanie, jeśli dodasz cechy ze zredukowanego zestawu do pierwotnych cech (również w celu poszukiwania optymalnej liczby skupień)?
- 12.** Wyucz model mieszanin gaussowskich na zestawie danych Olivetti. Aby przyspieszyć działanie algorytmu, należy prawdopodobnie zredukować wymiarowość zestawu danych (np. za pomocą analizy PCA, zachowując 99% wariancji). Użyj modelu do wygenerowania nowych twarzy (przy użyciu metody `sample()`) i zwizualizuj je (w przypadku użycia analizy głównych składowych konieczne będzie zastosowanie jej metody `inverse_transform()`). Spróbuj zmodyfikować niektóre obrazy (np. obróć je, utwórz lustrzane odbicie, przyciemnij) i sprawdź, czy model jest w stanie wykrywać anomalie (porównując wynik metody `score_samples()` dla zwykłych zdjęć i anomalii).
- 13.** Niektórych technik redukowania wymiarowości można używać także do wykrywania anomalii. Na przykład przeprowadź redukcję wymiarowości zestawu danych Olivetti za pomocą analizy PCA przy zachowaniu 99% wariancji. Następnie oblicz błąd rekonstrukcji dla każdego obrazu. Teraz skorzystaj ze zmodyfikowanych w poprzednim ćwiczeniu zdjęć i sprawdź ich błąd rekonstrukcji — zwróć uwagę, że jego wartość w ich przypadku jest większa. Jeżeli zwizualizujesz zrekonstruowany obraz, zrozumiesz, dlaczego tak jest: próbuje zrekonstruować normalną twarz.

Rozwiązań przykładów znajdziesz w dodatku A.

Sieci neuronowe i uczenie głębokie

Wprowadzenie do sztucznych sieci neuronowych i ich implementacji z użyciem interfejsu Keras

Ptaki zainspirowały człowieka do opanowania przestworzy, dzięki owocostanom łopianu wymyśliliśmy rzep, a natura zawiera niezliczoną liczbę pomysłów na wynalazki. Jest więc logiczne, że przyjrzenie się budowie mózgu będzie stanowić podwaliny pod stworzenie inteligentnych maszyn. To właśnie ta koncepcja pozwoliła nam stworzyć **sztuczne sieci neuronowe** (SSN, ang. *Artificial Neural Networks* — ANN) — model uczenia maszynowego zainspirowany występującymi w naszych mózgach sieciami neuronów biologicznych. Jednak mimo że samoloty są inspirowane ptakami, to nie muszą machać skrzydłami. Na drodze analogii sztuczne sieci neuronowe stopniowo zaczęły oddalać się od swoich biologicznych kuzynów. Niektórzy badacze twierdzą wręcz, że powinniśmy porzucić analogię biologiczną (np. mówiąc „jednostki” zamiast „neurony”), aby nie ograniczać naszej twórczości do tworów możliwych do istnienia w naturze¹.

Sztuczne sieci neuronowe stanowią podstawę uczenia głębokiego. Są one wszechstronne, potężne i skalowalne, dzięki czemu nadają się idealnie do olbrzymich i skomplikowanych zadań uczenia maszynowego, np. klasyfikowania miliardów obrazów (Zdjęcia Google), usług rozpoznawania mowy (Apple Siri), polecania filmików setkom milionów użytkowników każdego dnia (np. YouTube) lub uczenia komputera gry w Go (DeepMind AlphaGo).

W pierwszej części tego rozdziału przyjrzymy się sztucznym sieciom neuronowym, począwszy od spojrzenia na pierwsze, historyczne sieci SSN, a następnie zajmiemy się **perceptronami wielowarstwowymi** (ang. *Multi-Layer Perceptron* — MLP), których dziś używa się powszechnie zadaniach (pozostałe architektury omówię w kolejnych rozdziałach). W części drugiej nauczysz się implementować sieci neuronowe za pomocą popularnego interfejsu Keras. Jest to znakomicie zaprojektowany i prosty, wyspecjalizowany interfejs API, który służy do tworzenia, trenowania, oceniania i uruchamiania sieci neuronowych. Nie daj się jednak zwieść jego prostocie, okazuje się bowiem wystarczająco wyrazisty i elastyczny, aby umożliwiać tworzenie różnorodnych typów sieci neuronowych. W istocie,

¹ Możemy czerpać korzyści z obydwu światów, będąc otwartymi na biologiczne inspiracje, jednocześnie nie bojąc się tworzyć nirealistycznych biologicznie modeli, dopóki będą one dobrze działały.

Keras prawdopodobnie wystarczy w większości tradycyjnych zastosowań. Jeśli natomiast potrzebujesz dodatkowej elastyczności, zawsze możesz stworzyć niestandardowe moduły Keras za pomocą ogólnego API (zob. rozdział 12.).

Najpierw jednak cofnijmy się w czasie i zobaczymy, w jaki sposób przyszły na świat sieci neuronowe.

Od biologicznych do sztucznych neuronów

Co ciekawe, sztuczne sieci neuronowe wymyślono dość dawno temu — po raz pierwszy zostały zaproponowane już w 1943 roku przez neurofizjologa Warrena McCullocha i matematyka Waltera Pittsa. W swoim epokowym artykule *A Logical Calculus of Ideas Immanent in Nervous Activity* (<https://homl.info/43>)² obaj badacze pokazali uproszczony matematyczny model działania zespołów neuronów w zwierzęcych mózgach podczas przeprowadzania skomplikowanych obliczeń przy użyciu **rachunku zdań** (ang. *propositional logic*). Autorzy opisali architekturę pierwszej sztucznej sieci neuronowej. Jak się niebawem przekonamy, od tamtego czasu wymyślono już wiele innych architektur SSN.

Dzięki początkowym sukcesom na polu SSN powszechnie wierzono, że już wkrótce możliwe staną się rozmowy z prawdziwą sztuczną inteligencją. Gdy w latach 60. XX wieku stało się jasne, że to marzenie nie zostanie spełnione (przynajmniej przez jakiś czas), wstrzymano finansowanie badań nad sztucznymi sieciami neuronowymi i dla tej dziedziny nastąpiła dłużna „zima”. Na początku lat 80. ponownie pojawiło się zainteresowanie SSN, gdyż wymyślono nowe architektury sieci oraz opracowano lepsze techniki uczenia wchodzące w skład dziedziny zwanej **koneksjonizmem** (badanie sieci neuronowych). Ale rozwój był powolny i w latach 90. utworzono inne potężne modele uczenia maszynowego, takie jak maszyny wektorów nośnych (zob. rozdział 5.). Uważano, że techniki te pozwalają uzyskiwać lepsze rezultaty i mają lepsze podłożę teoretyczne niż SSN, dlatego znów świat stracił zainteresowanie sieciami neuronowymi.

Obecnie jesteśmy świadkami kolejnej fali zainteresowania SSN. Czy ta fala zaniknie równie szybko jak poprzednie? Istnieje kilka dobrych powodów, by wierzyć, że tym razem będzie inaczej i że takie ponownie wzniecone zainteresowanie sztucznymi sieciami neuronowymi będzie miało coraz większy wpływ na nasze życie:

- Obecnie jest dostępna olbrzymia ilość danych uczących sieci neuronowe, a architektura SSN często znacznie przewyższa wydajnością inne techniki uczenia maszynowego w kwestii dużych i skomplikowanych problemów.
- Znaczny wzrost mocy obliczeniowej od początku lat 90. ubiegłego wieku umożliwia trenowanie dużych sieci neuronowych w rozsądnym przedziale czasu. Wynika to częściowo z prawa Moore'a (w ciągu ostatnich 50 lat liczba elementów mieszących się w układach scalonych ulegała podwojeniu średnio co dwa lata), lecz również z rozwoju branży gier, z powodu której jest produkowana olbrzymia liczba kart graficznych (w milionach sztuk). Do tego usługi rozproszone sprawiają, że każdy użytkownik uzyskuje dostęp do tej mocy.

² Warren S. McCulloch i Walter Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, „The Bulletin of Mathematical Biology” 5, no. 4 (1943), s. 115 – 113.

- Algorytmy uczące zostały udoskonalone. Szczerze mówiąc, niewiele się one różnią od wersji z lat 90., ale te pozornie drobne modyfikacje mają bardzo pozytywny wpływ.
- Pewne teoretyczne ograniczenia sieci SSN okazały się w praktyce nieszkodliwe. Na przykład wiele osób uważało, że algorytmy uczące sieci SSN będą bezużyteczne, gdyż prawdopodobnie nie będą wychodzić poza optima lokalne. W rzeczywistości takie sytuacje zdarzają się bardzo rzadko (a nawet jeśli, to algorytmy te najczęściej zatrzymują się w pobliżu minimum globalnego).
- Wygląda na to, że dziedzina SSN wkroczyła w elitarny krąg dofinansowań i rozwoju. Zdumiewające produkty wykorzystujące tę technologię często są umieszczane w nagłówkach gazet i paskach informacyjnych, co przyciąga coraz większą uwagę (i inwestorów), dzięki czemu sztuczne sieci neuronowe są jeszcze bardziej rozwijane i wypuszcza się coraz ciekawsze produkty.

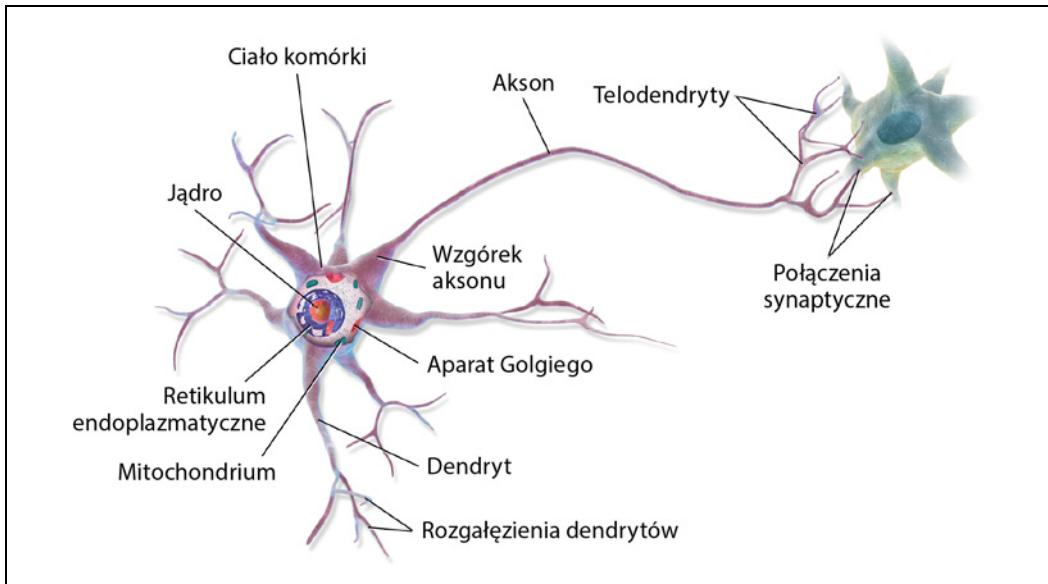
Neurony biologiczne

Zanim zajmiemy się sztucznymi neuronami, poznajmy najpierw budowę biologicznej komórki nerwowej (ukazanej na rysunku 10.1). Jest to specyficznie wyglądająca komórka mieszcząca się w zwierzęcym mózgu. Jej głównymi elementami są **ciało komórki** zawierające jądro i większość organelli komórkowych, wiele rozgałęziających się wypustek zwanych **dendrytami** oraz jedna bardzo dłuża wypustka — **akson**. Akson może być zaledwie kilkakrotnie dłuższy od ciała komórki, ale zdarzają się neurony, w których aksony są wiele tysięcy razy dłuższe. Na drugim końcu aksonu występują liczne rozgałęzienia, tzw. **telodendrony**, które są zakończone mikroskopijnymi tworami zwany mi **połączonymi synaptycznymi** (w skrócie **synapsami**), służącymi do łączenia się z dendrytami (lub z ciałem komórki) drugiej komórki nerwowej³. Neurony biologiczne generują krótkie impulsy elektryczne zwane **potencjałami czynnościowymi** (lub po prostu **sygnałami**), które są przenoszone wzdłuż aksonów i powodują uwalnianie w synapsach sygnałów chemicznych zwanych **neuroprzekaźnikami**. Kiedy komórka nerwowa otrzymuje dostateczną liczbę neuroprzekaźników od innych neuronów w ciągu kilku milisekund, to sama zaczyna wysyłać własne sygnały elektryczne (w rzeczywistości jest to uzależnione od neuroprzekaźników, gdyż niektóre z nich hamują aktywność komórki nerwowej).

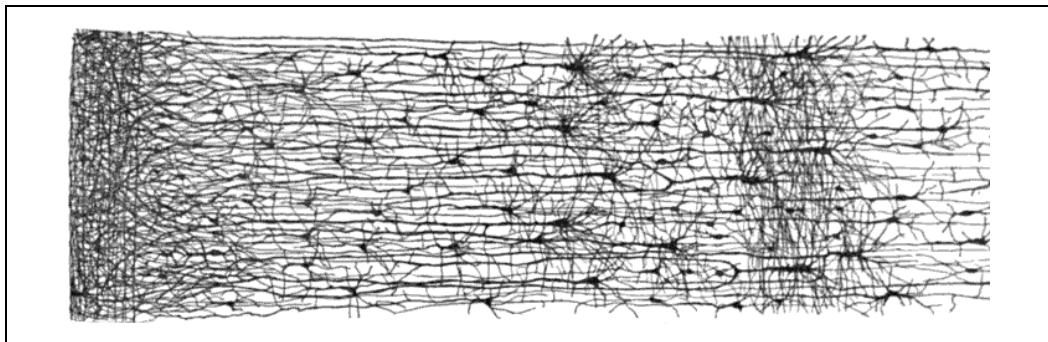
Zatem mechanizm działania poszczególnych neuronów jest dość prosty, tworzą one jednak rozległą sieć składającą się z miliardów komórek nerwowych, gdzie zazwyczaj jeden neuron łączy się z tysiącami innych. Dzięki tak olbrzymiej sieci zawierającej proste komórki nerwowe mogą być wykonywane skomplikowane obliczenia — przypomina to trochę złożone mrowisko, które zostaje utworzone przez armię względnie prostych mrówek. Architektura biologicznych sieci neuronowych (BSN)⁴ cały czas stanowi obiekt aktywnych badań, ale udało się nam już ustalić budowę niektórych obszarów mózgu i okazuje się, że neurony często są ulożone warstwowo, zwłaszcza w korze mózgowej (tzn. zewnętrznej warstwie mózgu), co możemy zobaczyć na rysunku 10.2.

³ Nie łączą się one ze sobą bezpośrednio, ale znajdują się wystarczająco blisko siebie, aby móc z dużą szybkością wymieniać się sygnałami chemicznymi.

⁴ W kontekście uczenia maszynowego wyrażenie „sieci neuronowe” przeważnie dotyczy sieci SSN, a nie BSN.



Rysunek 10.1. Neuron biologiczny⁵



Rysunek 10.2. Wielowarstwość biologicznej sieci neuronowej (kora mózgowa człowieka)⁶

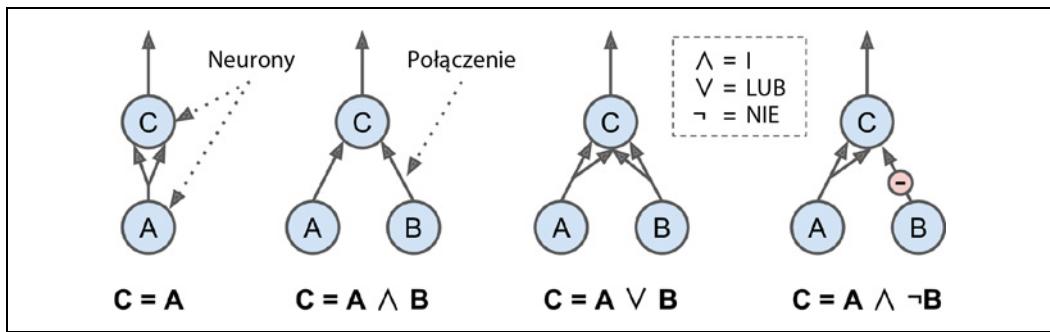
Operacje logiczne przy użyciu neuronów

McCulloch i Pitts zaproponowali bardzo prosty model biologicznego neuronu, który obecnie znamy jako **sztuczny neuron** (ang. *artificial neuron*): ma on co najmniej jedno binarne (stan 0 lub 1) wejście i tylko jedno wyjście binarne. W sztucznym neuronie wyjście jest uaktywniane wtedy, gdy jest aktywna określona liczba wejść. Twórcy modelu udowodnili, że za pomocą nawet tak uproszczonego modelu możliwe jest stworzenie sieci neuronowej rozwiązującej dowolne zadanie logiczne. Aby zrozumieć ich

⁵ Autor: Bruce Blaus (Uznanie autorstwa 3.0, <https://creativecommons.org/licenses/by/3.0/deed.pl>). Rysunek pochodzi ze strony <https://en.wikipedia.org/wiki/Neuron>.

⁶ Szkic uwarstwienia kory mózgowej autorstwa Santiago Ramón y Cajala (własność publiczna). Rysunek pochodzi ze strony https://en.wikipedia.org/wiki/Cerebral_cortex.

mechanizm działania, utwórzmy kilka sieci SSN wykonujących różne operacje logiczne (rysunek 10.3), przy założeniu, że neuron zostaje uaktywniony, gdy przynajmniej dwa wejścia będą aktywne.



Rysunek 10.3. Sztuczne sieci neuronowe przeprowadzające proste operacje logiczne

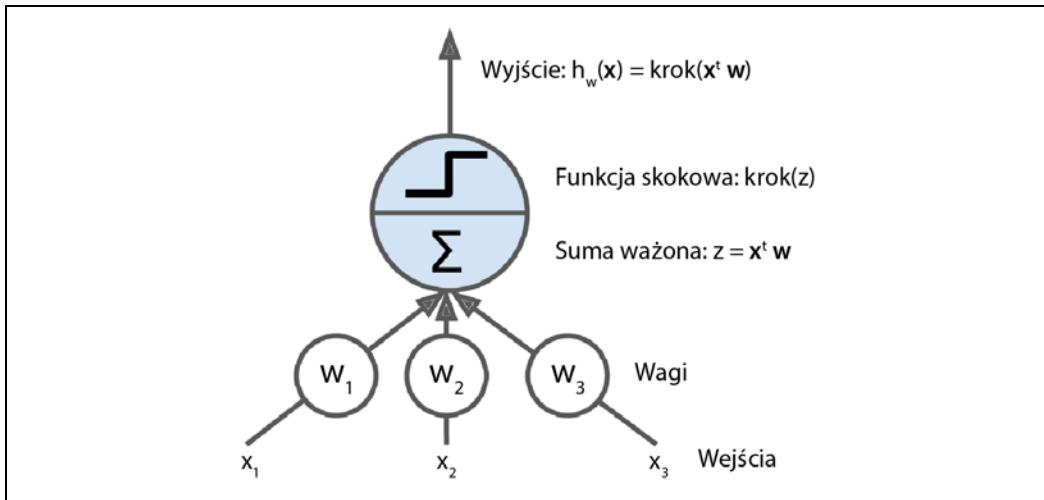
Zobaczmy, jakie operacje takie sieci mogą wykonywać:

- Pierwsza sieć po lewej to funkcja tożsamościowa: jeśli neuron A zostanie uaktywniony, aktywacji ulegnie także neuron C (gdyż odbiera dwa sygnały wejściowe z neuronu A); natomiast jeśli neuron A będzie wyłączony, to nie będzie również aktywny neuron C).
- Druga sieć wykonuje operację logiczną I: neuron C zostaje aktywowany jedynie wtedy, gdy obydwa neurony — A i B — będą włączone (pojedynczy sygnał wejściowy nie wystarczy do aktywacji neuronu C).
- Trzecia sieć przeprowadza operację logiczną LUB: neuron C jest aktywny, jeśli któryś z pozostałych neuronów (albo obydwa) zostanie aktywowany.
- W ostatnim przypadku przy założeniu, że sygnał może hamować aktywność neuronu (tak jak to się dzieje w neuronach biologicznych), omawiana sieć przeprowadza nieco bardziej skomplikowaną operację logiczną. Neuron C zostaje aktywowany wyłącznie wtedy, gdy otrzyma sygnał z neuronu A, natomiast neuron B musi być wyłączony. Jeśli neuron A będzie cały czas włączony, to otrzymamy bramkę negacji: neuron C jest aktywny przy wyłączonym neuronie B i odwrotnie.

Możemy sobie wyobrazić sposoby łączenia tych sieci w celu przeprowadzania złożonych operacji logicznych (jeden z przykładów znajdziesz w ćwiczeniach na końcu rozdziału).

Perceptron

Perceptron stanowi jedną z najprostszych architektur SSN, zaproponowaną w 1957 roku przez Franka Rosenblatta. Jego podstawą jest nieco zmodyfikowany sztuczny neuron (rysunek 10.4), zwany **progową jednostką logiczną** (ang. *Threshold Logic Unit* — TLU) lub **liniową jednostką progową** (ang. *Linear Threshold Unit* — LTU). Wartościami wejść/wyjść są liczby (a nie stany binarne), a każde połączenie ma przyporządkowaną wagę. Jednostka TLU wylicza ważoną sumę sygnałów wejściowych ($z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{x}^T\mathbf{w}$), a następnie zostaje użyta **funkcja skokowa** wobec tej sumy, dająca ostateczny wynik: $h_w(\mathbf{x}) = \text{skok}(z)$, gdzie $z = \mathbf{x}^T\mathbf{w}$.



Rysunek 10.4. Liniowa jednostka logiczna: sztuczny neuron obliczający sumę ważoną sygnałów wejściowych, a następnie stosujący funkcję skokową

Najczęściej używaną funkcją skokową w perceptronach jest **funkcja skokowa Heaviside'a** (równanie 10.1). Czasami zamiast niej stosuje się funkcję signum.

Równanie 10.1. Najpowszechniejsze funkcje skokowe wykorzystywane w perceptronach (przy założeniu, że $\text{próg}=0$)

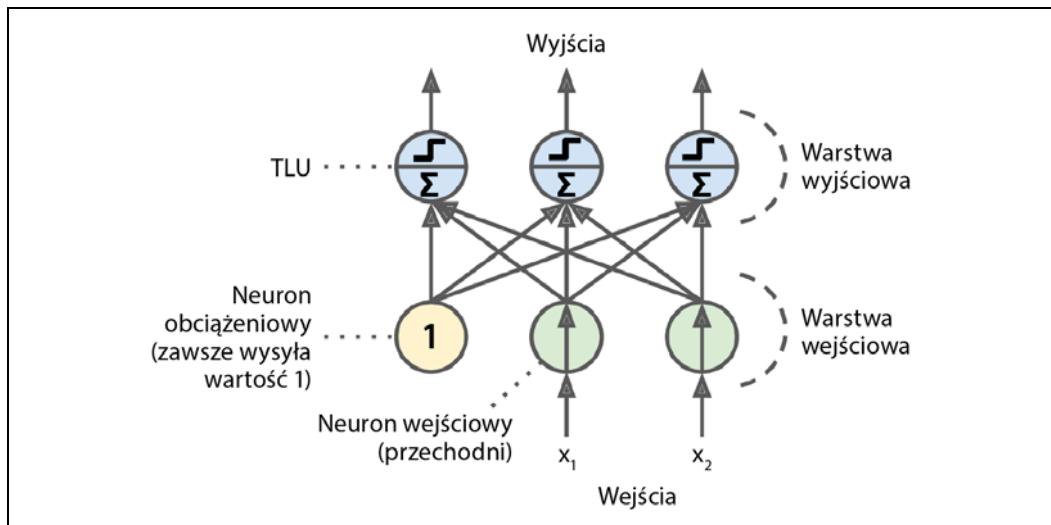
$$\text{Heaviside}(z) = \begin{cases} 0 & \text{jeśli } z < 0 \\ 1 & \text{jeśli } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{jeśli } z < 0 \\ 0 & \text{jeśli } z = 0 \\ +1 & \text{jeśli } z > 0 \end{cases}$$

Pojedyncza jednostka TLU może być używana w prostych zadaniach klasyfikacji. Olicza liniową kombinację danych wejściowych i jeżeli wynik przekracza określony próg, to klasyfikuje przykład do klasy pozytywnej, jeśli zaś rezultat nie przekroczy progu — do klasy negatywnej (podobnie jak w klasyfikatorze regresji logistycznej lub w liniowym klasyfikatorze SVM). Przykładowo jesteśmy w stanie przy użyciu jednej TLU klasyfikować kwiaty kosaćca na podstawie długości i szerokości płatka (nie zapominając o dodaniu dodatkowej cechy obciążenia $x_0 = 1$, tak jak robiliśmy to w poprzednich rozdziałach). Trening TLU oznacza w tym przypadku znalezienie właściwych wartości wag w_0 , w_1 i w_2 (wkrótce zajmiemy się algorytmem uczącym).

Perceptron składa się po prostu z jednej warstwy jednostek TLU⁷, gdzie każdy neuron jest połączony ze wszystkimi wejściami. Tego typu warstwa nazywana jest **warstwą w pełni połączoną** lub **warstwą gęstą**. Sygnały wejściowe perceptronu są następnie dostarczane do specyficznych neuronów przechodniczych, zwanych **neuronami wejściowymi** (ang. *input neuron*): przekazują dalej wszelkie dostarczane do nich dane wejściowe. Wszystkie neurony wejściowe tworzą **warstwę wejściową** (ang. *input layer*). Ponadto najczęściej jest również wstawiane dodatkowe obciążenie ($x_0 = 1$), przeważnie reprezentowane za pomocą tak zwanego **neuronu obciążeniowego** (ang. *bias neuron*) —

⁷ Nazwą „perceptron” jest czasami określana niewielka sieć zawierająca jedną LTU.

jego zadaniem jest wysyłanie na wyjście wartości 1. Na rysunku 10.5 widzimy perceptron z dwoma wejściami i trzema wyjściami. Może on jednocześnie klasyfikować próbki do trzech różnych klas binarnych, czyli stanowi on klasyfikator wielowyjściowy.



Rysunek 10.5. Diagram perceptronu składającego się z dwóch neuronów wejściowych, jednego obciążeniowego i trzech wyjściowych

Dzięki magii algebry liniowej wzór widoczny w równaniu 10.2 jest w stanie skutecznie obliczać sygnały warstwy wyjściowej sztucznych neuronów dla kilku przykładów naraz.

Równanie 10.2. Obliczanie sygnałów wyjściowych w warstwie gęstej

$$h_{W,b}(X) = \phi(XW + b)$$

W tym równaniu:

- Jak zwykle X symbolizuje macierz cech wejściowych. Każdy rząd jest poświęcony przykładowi, a kolumna cesze.
- Macierz wag W zawiera wszystkie wagi połączeń oprócz wychodzących z neuronem obciążeniowym. Każdy rząd jest poświęcony neuronowi wejściowemu, a kolumna sztucznemu neuronowi w danej warstwie.
- Vektor obciążzeń b zawiera wszystkie wagi połączeń pomiędzy neuronem obciążeniowym a pozostałymi neuronami. Występuje tu po jednym członie obciążenia na każdy sztuczny neuron.
- Funkcja ϕ to tak zwana funkcja aktywacji: w przypadku TLU jest ona funkcją skokową (ale już niebawem zajmiemy się omówieniem innych rodzajów funkcji aktywacji).

Jak więc wygląda trenowanie perceptronu? Algorytm uczący zaproponowany przez Franka Rosenblatta był w znacznej mierze inspirowany **regułą Hebb'a**. Donald Hebb w swojej książce *The Organization of Behavior* (Wiley) z 1949 roku zasugerował, że gdy biologiczny neuron często pobudza inną komórkę nerwową, to połączenie pomiędzy tymi dwoma neuronami staje się silniejsze. Koncepcja ta

została później błyskotliwie podsumowana przez Siegrida Löwela: „Cells that fire together, wire together”, czyli w wolnym tłumaczeniu: „Komórki pobudzające się wzajemnie wiążą się ze sobą”, a zatem waga połączenia pomiędzy dwoma neuronami zazwyczaj staje się coraz większa w przypadku, gdy są one aktywowane jednocześnie. Reguła ta została później nazwana **regułą Hebb'a** (lub **uczeniem hebbowskim**). Perceptrony są uczone za pomocą odmiany tej reguły, w której jest brany pod uwagę błąd popełniany przez sieć podczas prognozowania wyniku; wzmacniane są połączenia pomagające zmniejszyć wartość błędu. Dokładniej mówiąc, perceptron przetwarza w danym momencie jeden przykład uczący i wyciąga dla niego prognozy. Na każdy neuron wyjściowy odpowiedzialny za nieprawidłowy wynik następuje zwiększenie wag połączeń ze wszystkimi wejściami przyczyniającymi się do właściwej prognozy. Reguła ta została ukazana w równaniu 10.3.

Równanie 10.3. Reguła uczenia perceptronu (aktualizowanie wag)

$$w_{i,j}^{(\text{następny krok})} = w_{i,j} + \eta (y_j - \hat{y}_j) x_i$$

W tym równaniu:

- $w_{i,j}$ — waga połączenia pomiędzy i -tym neuronem wejściowym i j -tym neuronem wyjściowym,
- x_i — i -ta wartość wejściowa bieżącego przykładu uczącego,
- \hat{y}_j — wynik j -tego neuronu wyjściowego dla bieżącego przykładu uczącego,
- y_j — docelowy wynik j -tego neuronu wyjściowego dla bieżącego przykładu uczącego,
- η — współczynnik uczenia.

Granica decyzyjna każdego neuronu wyjściowego jest liniowa, dlatego perceptron nie jest w stanie uczyć się skomplikowanych wzorców (jak klasyfikatory regresji logistycznej). Rosenblatt udowodnił jednak, że jeśli próbki uczące będą liniowo rozdzielne, to algorytm osiągnie zbieżność⁸. Jest to tak zwane **twierdzenie o zbieżności perceptronu**.

Moduł Scikit-Learn zawiera klasę Perceptron, implementującą pojedynczą sieć TLU. Możemy korzystać z niej zgodnie z oczekiwaniami — np. wobec zbioru danych Iris (omówionego w rozdziale 4.):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)] # Długość płatka, szerokość płatka
y = (iris.target == 0).astype(np.int) # Iris setosa?

per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

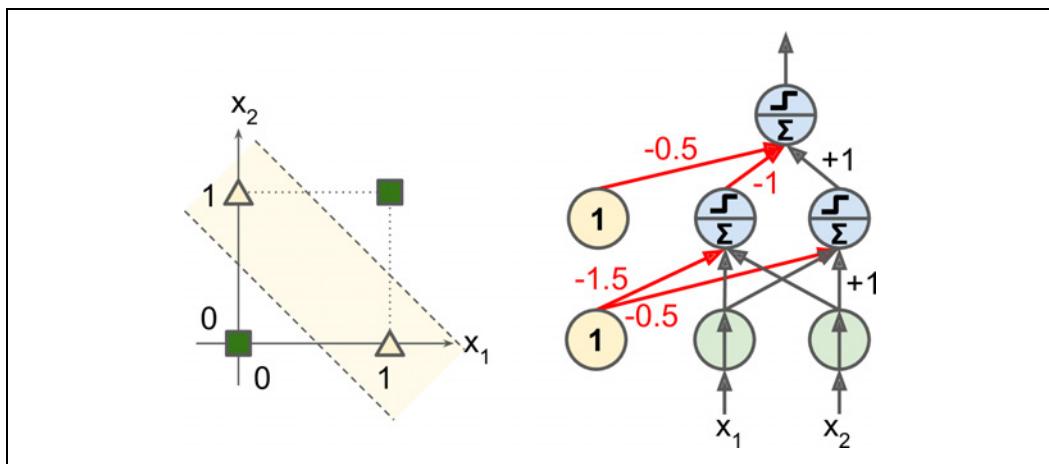
Być może nie uszło Twojej uwadze duże podobieństwo algorytmu uczącego perceptron do stochastycznego spadku wzduż gradientu. Faktycznie, klasa Perceptron jest równoważna stosowaniu

⁸ Zwrócić uwagę, że to rozwiązanie nie jest takie niespotykane: gdy punkty danych są liniowo rozdzielne, istnieje nieskończona liczba hiperpłaszczyzn służących do oddzielania danych.

klasy SGDClassifier z następującymi parametrami: loss="perceptron", learning_rate="constant", eta0=1 (współczynnik uczenia) oraz penalty=None (brak regularyzacji).

Zwróć uwagę, że w przeciwieństwie do klasyfikatorów regresji logistycznej perceptrony nie wyliczają prawdopodobieństwa przynależności do klasy: określają one prognozy na podstawie wyznaczonego progu. Jest to jeden z powodów, dla których warto wybierać klasyfikatory regresji logistycznej przed perceptronami.

Marvin Minsky i Seymour Papert w swojej monografii *Perceptrons* z 1969 roku wykazali wiele poważnych wad modelu perceptronu, zwłaszcza związanych z niemożnością rozwiązywania pewnych trywialnych problemów (np. zadania klasycznego alternatywy rozłącznej — XOR) (rysunek 10.6). Można to samo powiedzieć również o dowolnym innym modelu klasifikacji liniowej (np. algorytmach regresji logistycznej), badacze jednak oczekiwali znacznie więcej od perceptronów, dlatego nie ma co się dziwić skali ich rozczarowania: w konsekwencji wielu naukowców zupełnie porzuciło badania nad sieciami neuronowymi na rzecz ogólniejszych zagadnień, np. logiki, rozwiązywania lub wyszukiwania rozwiązań problemów.



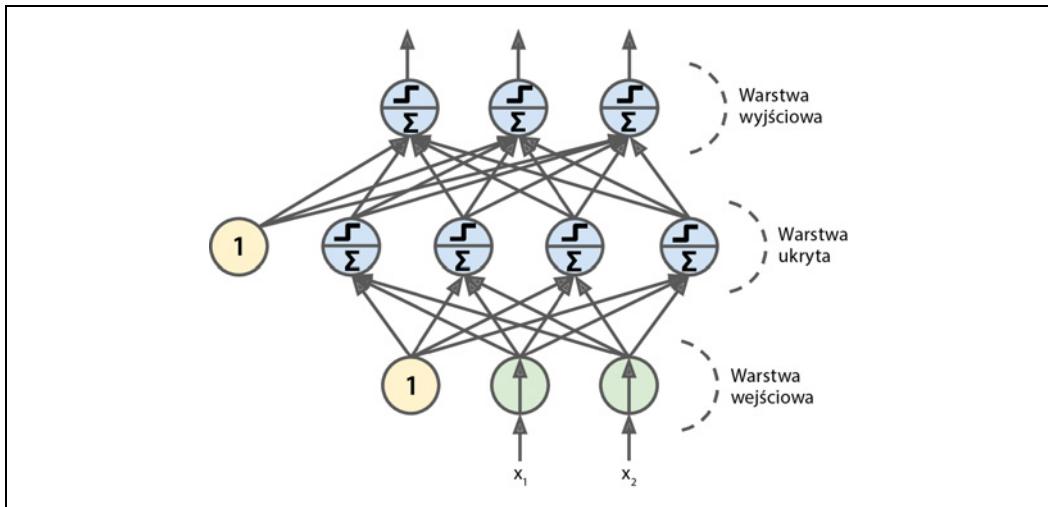
Rysunek 10.6. Problem klasyfikacji XOR i model perceptronu MLP zdolny do jego rozwiązania

Okazuje się, że możemy wyeliminować część ograniczeń, tworząc wiele warstw perceptronów. Tego typu architekturę SSN nazywamy **perceptronem wielowarstwowym** (ang. *Multi-Layer Perceptron* — MLP). Perceptron MLP może wykonywać operację XOR, co jesteśmy w stanie samodzielnie sprawdzić, wyliczając wynik perceptronu zaprezentowanego po prawej stronie rysunku 10.6, dla każdej kombinacji danych wejściowych: dla danych (0, 0) i (1, 1) wynik zawsze wynosi 0, a dla stanów (0, 1) i (1, 0) — 1. Oprócz czterech zaprezentowanych na rysunku połączeń wagi wszystkich pozostały są równe 1. Spróbuj samodzielnie sprawdzić, czy ta sieć rzeczywiście jest w stanie wykonać operację XOR.

Perceptron wielowarstwowy i propagacja wsteczna

Perceptron MLP składa się z jednej **warstwy wejściowej** (przechodniej), co najmniej jednej warstwy jednostek TLU (tzw. **warstwy ukryte**) i ostatniej warstwy jednostek TLU (**warstwa wyjściowa**)

(rysunek 10.7). Warstwy znajdujące się bliżej warstwy wejściowej są zazwyczaj nazywane **warstwami dolnymi** (ang. *lower layer*), natomiast te, które występują bliżej warstwy wyjściowej, to **warstwy górne** (ang. *upper layer*). Oprócz warstwy wyjściowej każda warstwa zawiera neuron obciążający i jest w pełni połączona z następną warstwą.



Rysunek 10.7. Architektura perceptronu wielowarstwowego zawierającego dwa neurony wejściowe, jedną warstwę ukrytą składającą się z czterech neuronów i trzy neurony wyjściowe (zostały pokazane również neurony obciążeniowe, ale przeważnie nie są one uwidaczniane)



Sygnal przepływa tylko w jednym kierunku (od wejść do wyjść), dlatego jest to przykład **jednokierunkowej sieci neuronowej** (ang. *Feedforward Neural Network* — FNN).

Gdy SSN zawiera wiele warstw ukrytych⁹, nosi nazwę **głębokiej sieci neuronowej** (ang. *Deep Neural Network* — DNN). Analizą DNN, a ogólniej modeli zawierających głębokie stopy obliczeniowe, zajmuje się dział uczenia głębokiego. Mimo to wiele osób mówi o uczeniu głębokim nawet w kontekście płytowych sieci neuronowych.

Przez wiele lat naukowcy próbowali bezskutecznie znaleźć sposób uczenia perceptronów wielowarstwowych. Dopiero w 1986 roku David Rumelhart, Geoffrey Hinton i Ronald Williams opublikowali przełomowy artykuł (<https://homl.info/44>)¹⁰, w którym wprowadzili koncepcję algorytmu **propagacji wstecznej** (ang. *backpropagation*)¹¹, który jest stosowany do teraz. Mówiąc krótko, jest to

⁹ Jeszcze w latach 90. XX wieku za sieci głębokie uznawano sieci składające się z ponad dwóch warstw ukrytych. Obecnie powszechnie spotykane są sieci zawierające dziesiątki, a nawet setki warstw, zatem pojęcie „głęboka” jest dość względne.

¹⁰ David Rumelhart i in., *Learning Internal Representations by Error Propagation* (raport techniczny Defense Technical Information Center, September 1985).

¹¹ W rzeczywistości algorytm ten był już kilka razy odkrywany niezależnie przez różnych naukowców w różnych dziedzinach wiedzy, a pierwszy był Paul Werbos w 1984 roku.

algorytm gradientu prostego (zob. rozdział 4.) przy użyciu skutecznej techniki automatycznego obliczania gradientu: algorytm propagacji wstecznej w zaledwie dwóch przebiegach (po jednym do przodu i do tyłu) jest w stanie obliczyć gradient błędu sieci w odniesieniu do każdego parametru modelu. Innymi słowy, może określić stopień zmodyfikowania wszystkich wag połączeń i członów obciążenia w sposób zmniejszający wartość błędu. Po uzyskaniu tych gradientów realizowany jest klasyczny algorytm gradientu prostego, a cały proces jest powtarzany do momentu uzyskania zbieżności z rozwiązaniem.



Automatyczne obliczanie gradientów nazywane jest **różniczkowaniem automatycznym** (ang. *automatic differentiation* lub *autodiff*). Istnieje wiele metod różniczkowania automatycznego, z których każda ma swoje wady i zalety. W algorytmie propagacji wstecznej wykorzystywana jest metoda **odwrotnego różniczkowania automatycznego** (ang. *reverse-mode autodiff*). Jest to szybkie i precyzyjne rozwiązanie, a także sprawdza się w przypadku występowania wielu zmiennych w różniczkowalnej funkcji (np. wag połączeń) przy niewielkiej liczbie wyjść (np. jednej funkcji straty). Więcej informacji na temat różniczkowania automatycznego znajdziesz w dodatku D.

Przyjrzyjmy się temu algorytmowi nieco dokładniej:

- Za każdym razem przetwarza on po jednej minigrupie (np. zawierającej 32 przykłady), a cały zestaw danych zostaje użyty wielokrotnie. Każdy przebieg nazywany jest epoką (ang. *epoch*).
- Każda minigrupa jest przekazywana do warstwy wejściowej sieci, a stamtąd do pierwszej warstwy ukrytej. Algorytm następnie oblicza wynik wszystkich neuronów w tej warstwie (dla każdego przykładu z minigrupy). Rezultat zostaje przekazany do następnej warstwy, gdzie jest obliczany wynik przekazywany do kolejnej warstwy i proces ten jest powtarzany aż do osiągnięcia warstwy wyjściowej. Jest to przebieg w przód (ang. *forward pass*) — różni się on od uzyskiwania prognoz jedynie tym, że wszystkie wyniki pośrednie zostają zachowane, gdyż będą potrzebne w przebiegu wstecznym.
- Następnie zostaje zmierzony błąd na wyjściu sieci (tzn. wykorzystujemy funkcję straty do porównania różnicy pomiędzy oczekiwany a uzyskanym wynikiem, zostaje zwrócona jakaś miara błędu).
- Teraz zostaje obliczony wkład każdego połączenia wyjściowego w wartość błędu. Dokonujemy tego analitycznie za pomocą tzw. **reguły łańcuchowej** (ang. *chain rule*; być może jest to najważniejsza reguła w analizie matematycznej), dzięki czemu faza ta jest szybka i precyzyjna.
- Algorytm następnie mierzy wpływ każdego połączenia na tę wartość błędu w warstwie znajdującej się poniżej, również za pomocą reguły łańcuchowej, i kieruje się wstecz aż do osiągnięcia warstwy wejściowej. Jak już wiemy, taki odwrotny przebieg skutecznie mierzy gradient błędu we wszystkich wagach połączeń poprzez propagację tego gradientu w kierunku początku sieci (stąd nazwa algorytmu).
- Na koniec algorytm wykorzystuje mechanizm gradientu prostego do usprawnienia wszystkich wag połączeń w sieci za pomocą dopiero co obliczonych gradientów błędu.

Algorytm ten jest tak istotny, że należy go jeszcze raz podsumować: dla każdej próbki uczącej algorytm propagacji wstecznej najpierw wylicza prognozę (przebieg w przód) i mierzy błąd, następnie cofa

się w kierunku początku sieci, sprawdzając wkład każdego połączenia w zmierzony błąd (odwrotny przebieg), po czym modyfikuje wagi w celu zredukowania błędu (etap gradientu prostego).



Należy koniecznie zainicjalizować losowo wagi połączeń wszystkich warstw ukrytych, gdyż w przeciwnym wypadku proces uczenia zakończy się niepowodzeniem. Na przykład jeżeli wszystkie wagi i obciążenia zostaną zainicjalizowane z wartością 0, to wszystkie neurony w danej warstwie będą identyczne, a zatem faza propagacji wstępnej pozmienna je w taki sam sposób, więc pozostały ostatecznie takie same. Innymi słowy, nawet pomimo występowania setek neuronów w danej warstwie model będzie działał tak, jakby w każdej warstwie znajdował się tylko jeden neuron: nie będzie zbyt inteligentny. Jeżeli natomiast zainicjalizujesz losowo wagi, to **złamiesz symetrię**, dzięki czemu algorytm propagacji wstępnej będzie w stanie wytrenować zespół zróżnicowanych neuronów.

Żeby ten algorytm działał prawidłowo, jego twórcy musieli wprowadzić kluczową zmianę w architekturze MLP: zastąpili funkcję skokową funkcją logistyczną (sigmoidalną) — $\sigma(z) = 1/(1+\exp(-z))$. Było to niezbędne, ponieważ funkcja skokowa zawiera jedynie płaskie segmenty, dlatego nie pozwala korzystać z gradientu (algorytm gradientu prostego nie może poruszać się po płaskim przebiegu funkcji), natomiast funkcja logistyczna ma w każdym punkcie zdefiniowaną pochodną niezerową, dzięki czemu algorytm gradientu prostego może na każdym etapie uzyskiwać lepsze wyniki. W rzeczywistości algorytm propagacji wstępnej dobrze współdziała z wieloma innymi **funkcjami aktywacji**, nie tylko z funkcją logistyczną. Poniżej przedstawiam dwie inne popularne funkcje aktywacji:

Funkcja tangensa hiperbolicznego: $\tanh(z) = 2\sigma(2z)-1$

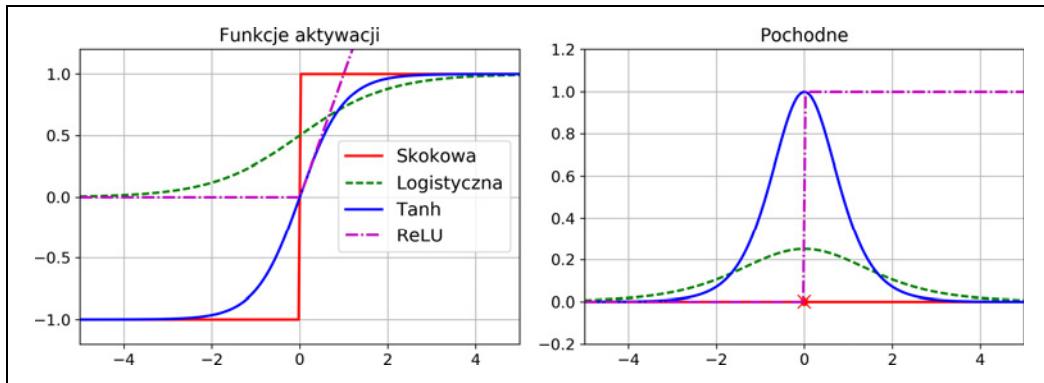
Ta funkcja aktywacji jest S-kształtna, ciągła i różniczkowalna, podobnie jak funkcja logistyczna, ale zakres wartości wyjściowych wynosi w niej od -1 do 1 (w funkcji logistycznej zakres ogranicza się od 0 do 1). Dzięki temu zakresowi wynik każdej warstwy jest mniej więcej wyśrodkowany wobec zera na początku uczenia, co często pomaga w szybszym uzyskaniu zbieżności.

Funkcja ReLU (ang. Recified Linear Unit — prostowana jednostka liniowa): $\text{ReLU}(z) = \max(0, z)$

Funkcja ReLU jest ciągła, ale nieróżniczkowalna w punkcie $z = 0$ (nachylenie zmienia się gwałtownie, przez co metoda gradientu prostego może mieć tu problem), a jej pochodna dla $z < 0$ wynosi 0. W praktyce jednak spisuje się znakomicie, a dodatkowym atutem jest szybkość przetwarzania, przez co stała się funkcją domyślną¹². Najważniejsze jednak, że nie ma ona maksymalnej wartości wyjściowej, co pomaga zredukować pewne problemy z metodą gradientu prostego (w rozdziale 11. powróćmy jeszcze do tego zagadnienia).

Popularne funkcje aktywacji wraz z ich pochodnymi zostały zaprezentowane na rysunku 10.8. Ale chwila! Dlaczego w ogóle są nam potrzebne funkcje aktywacji? Jeżeli połączysz sekwencyjnie kilka przekształceń liniowych, otrzymasz transformację liniową. Na przykład jeżeli $f(x) = 2x+3$, a $g(x) = 5x-1$, to z połączenia tych dwóch funkcji otrzymasz kolejną funkcję liniową: $f(g(x)) = 2(5x-1)+3 = 10x+1$.

¹² Uważa się, że neurony biologiczne wykorzystują funkcję aktywacji zbliżoną do sigmoidalnej (S-kształtną), dlatego naukowcy przez bardzo długi czas poprzestawali na niej. Okazuje się jednak, że funkcja aktywacji ReLU spisuje się generalnie lepiej w przypadku sztucznych sieci neuronowych. Jest to jedna z sytuacji, w których analogia z biologicznym układem nerwowym jest nietrafiona.



Rysunek 10.8. Funkcje aktywacji wraz z ich pochodnymi

Zatem jeśli pomiędzy warstwami nie występuje pewna nieliniowość, to nawet głęboki stos warstw staje się równoważny pojedynczej warstwie, przez co nie bylibyśmy w stanie rozwiązywać wielu problemów. Z kolei wystarczająco rozbudowana DNN zawierająca nieliniowe funkcje aktywacji może w teorii aproksymować dowolną funkcję ciągłą.

Świętne! Wiesz już, skąd wywodzą się sieci neuronowe, jak wygląda ich struktura oraz jak obliczamy ich wyniki. Wiesz także co nieco na temat algorytmu propagacji wstecznej. Co właściwie jednak możemy z nimi robić?

Regresyjne perceptrony wielowarstwowe

Perceptrony wielowarstwowe można stosować w zadaniach regresji. Jeżeli chcesz prognozować pojedynczą wartość (np. cenę domu na podstawie wielu cech), to wystarczy pojedynczy neuron wyjściowy — jego wynikiem będzie prognozowana wartość. W przypadku regresji wielu zmiennych (tzn. takiej, w której przewidywanych jest wiele wartości jednocześnie) wymagany jest jeden neuron wyjściowy na każdy wymiar wynikowy. Przykładowo gdybyśmy chcieli zlokalizować środek obiektu na zdjęciu, musielibyśmy prognozować współrzędne dwuwymiarowe, czyli potrzebne by były dwa neurony wyjściowe. Aby umieścić ten obiekt w prostokącie ograniczającym, wymagane byłyby jeszcze dwie dodatkowe wartości: szerokość i długość obiektu. Zatem musielibyśmy wyznaczyć cztery neurony wyjściowe.

Zasadniczo w trakcie tworzenia regresyjnych perceptronów wielowarstwowych nie chcemy wyznaczać neuronom wyjściowym żadnej funkcji aktywacji, dzięki czemu są one w stanie generować wynik w dowolnym zakresie wartości. Gdybyśmy chcieli sprawić, by był zawsze dodatni, moglibyśmy wprowadzić funkcję aktywacji ReLU w warstwie wyjściowej. Ewentualnie można skorzystać z funkcji aktywacji **softplus**, czyli „wyglądzonego” wariantu funkcji ReLU: $softplus(z) = \log(1+\exp(z))$. Dla ujemnego z przyjmuje wartości bliskie zeru, a w przypadku wartości dodatniej z dąży do z . Jeżeli natomiast chcesz, aby predykcje mieściły się w wyznaczonym zakresie wartości, możesz wprowadzić funkcję logistyczną lub tangensa hiperbolicznego, a następnie przeskalać etykiety do właściwego zakresu: od 0 do 1 dla funkcji logistycznej i od -1 do 1 dla tangensa hiperbolicznego.

Tradycyjnie funkcją straty podczas uczenia jest błąd średniokwadratowy, jeżeli jednak w zestawie danych uczących znajduje się wiele elementów odstających, lepszym rozwiązaniem może się okazać średni błąd bezwzględny. Ewentualnie możesz skorzystać z funkcji Hubera, stanowiącej połączenie obydwu powyższych.



Funkcja Hubera jest kwadratowa dla wartości błędu nieprzekraczającej wartości progowej δ (zazwyczaj 1), ale staje się liniowa dla δ większej od 1. Część liniowa sprawia, że funkcja ta jest mniej wrażliwa na elementy odstające w porównaniu do funkcji średniokwadratowej, natomiast część kwadratowa umożliwia szybsze i precyzyjniejsze uzyskanie zbieżności w stosunku do średniego błędu bezwzględnego.

W tabeli 10.1 podsumowuję typową strukturę regresyjnego perceptronu wielowarstwowego.

Tabela 10.1. Typowa architektura regresyjnego perceptronu MLP

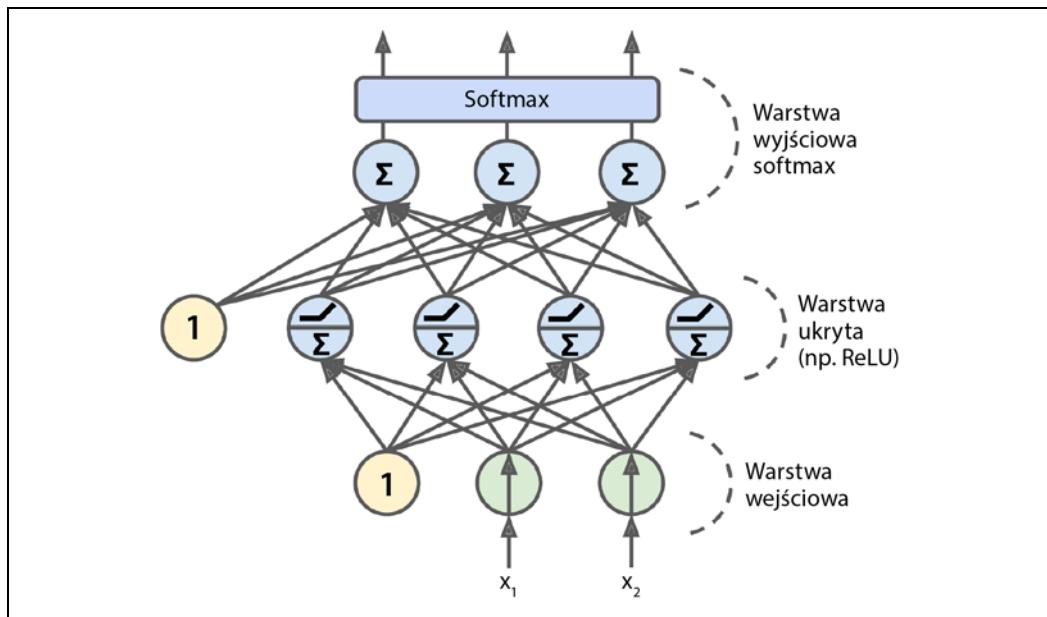
Hiperparametr	Wartość typowa
Liczba neuronów wejściowych	Po jednym na cechę wejściową (np. $28 \times 28 = 784$ dla zestawu danych MNIST)
Liczba warstw ukrytych	W zależności od problemu, ale zazwyczaj od 1 do 5
Liczba neuronów tworzących warstwę ukrytą	W zależności od problemu, ale zazwyczaj od 10 do 100
Liczba neuronów wyjściowych	Po jednym na każdy wymiar predykcji
Funkcja aktywacji w warstwie ukrytej	ReLU (lub SELU) (zob. rozdział 11.)
Funkcja aktywacji w warstwie wyjściowej	Żadna lub ReLU/softplus (dla wyników dodatnich) albo logistyczna/tanh (dla ograniczonego zakresu wartości)
Funkcja straty	MSE lub MAE/Hubera (w przypadku występowania elementów odstających)

Klasyfikacyjne perceptrony wielowarstwowe

Perceptronów wielowarstwowych można również używać w zadaniach klasyfikacji. W przypadku klasyfikacji binarnej wystarczy jeden neuron wyjściowy zawierający funkcję logistyczną — uzyskiwany wynik będzie mieścił się w zakresie od 0 do 1, co można interpretować jako prawdopodobieństwo przynależności do klasy pozytywnej. Oszacowane prawdopodobieństwo przynależności do klasy negatywnej obliczamy, odejmując tę wartość od 1.

Perceptrony wielowarstwowe radzą sobie również z zadaniami wieloklasowej klasyfikacji binarnej (zob. rozdział 3.). Możesz na przykład opracować system klasyfikowania wiadomości e-mail przewidujący, czy wiadomości przychodzące są normalne, czy są spamem, i jednocześnie określający ich priorytet. W takiej sytuacji niezbędne są dwa neurony wyjściowe zawierające logistyczną funkcję aktywacji: pierwszy z nich określałby prawdopodobieństwo przynależności wiadomości do spamu, a drugi prognozałby ilość wiadomości. Mówiąc ogólniej, wyznaczamy po jednym neuronie wyjściowym na każdą klasę pozytywną. Zwróć uwagę, że prawdopodobieństwa przynależności do poszczególnych klas nie muszą dawać w sumie 1. Dzięki temu model jest w stanie dawać w rezultacie dowolną kombinację etykiet: niepełne zwykłe wiadomości, pilne zwykłe wiadomości, niepełny spam, a nawet pełny spam (chociaż w tym ostatnim przypadku mielibyśmy raczej do czynienia z błędem).

Jeżeli każdy przykład może przynależeć tylko do jednej z co najmniej trzech klas (w przypadku klasyfikacji obrazów cyfr mamy do czynienia z klasami od 0 do 9), to na każdą klasę przypada jeden neuron wyjściowy i w całej warstwie wyjściowej należy wprowadzić funkcję aktywacji softmax (rysunek 10.9). Omówiona w rozdziale 4. funkcja softmax gwarantuje, że wszystkie szacowane prawdopodobieństwa mieszczą się w zakresie od 0 do 1, a ich suma wynosi 1 (co jest wymagane, jeżeli klasy mają wykluczać się wzajemnie). Jest to tak zwana klasyfikacja wieloklasowa.



Rysunek 10.9. Współczesny perceptron wielowarstwowy (zawierający funkcje aktywacji ReLU i softmax) służący do klasyfikacji

W kontekście funkcji straty pamiętajmy, że prognozujemy tu rozkłady prawdopodobieństwa, dlatego zazwyczaj dobrym wyborem okazuje się funkcja entropii krzyżowej (zwana także logarytmiczną funkcją straty) (zob. rozdział 4.).

Tabela 10.2 podsumowuje typową architekturę klasyfikacyjnego perceptronu wielowarstwowego.

Tabela 10.2. Typowa struktura klasyfikacyjnego perceptronu wielowarstwowego

Hiperparametr	Klasyfikacja binarna	Binarna klasyfikacja wieloetykietowa	Klasyfikacja wieloklasowa
Warstwy wejściowa i ukryte	Tak samo jak w przypadku regresji	Tak samo jak w przypadku regresji	Tak samo jak w przypadku regresji
Liczba neuronów wyjściowych	Jeden	Po jednym na etykietę	Po jednym na klasę
Funkcja aktywacji na wyjściu	Logistyczna	Logistyczna	Softmax
Funkcja straty	Entropia krzyżowa	Entropia krzyżowa	Entropia krzyżowa



Zanim przejdziemy dalej, zalecam wykonanie ćwiczenia 1., które znajdziesz na końcu rozdziału. Zapoznasz się z różnymi strukturami sieci neuronowych i zwizualizujesz ich rezultaty w środowisku **TensorFlow Playground**. W ten sposób łatwiej będzie Ci zrozumieć perceptrony wielowarstwowe, w tym również wpływ hiperparametrów (liczby warstw i neuronów, rodzaju funkcji aktywacji itd.).

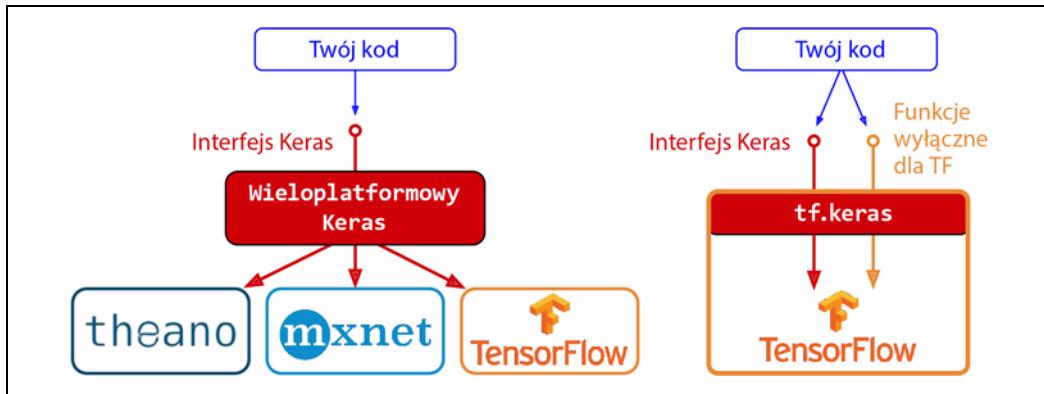
Znasz już wszystkie pojęcia niezbędne, by rozpocząć implementowanie perceptronów wielowarstwowych za pomocą interfejsu Keras.

Implementowanie perceptronów wielowarstwowych za pomocą interfejsu Keras

Keras to wyspecjalizowany, przeznaczony do uczenia głębokiego interfejs API, za pomocą którego możemy z łatwością tworzyć, uczyć, oceniać i uruchamiać wszystkie rodzaje sieci neuronowych. Jego dokumentacja (czy też specyfikacja) dostępna jest pod adresem <https://keras.io/>. Implementacja referencyjna (<https://github.com/keras-team/keras>), również nosząca nazwę Keras, została zaprojektowana przez François Cholleta jako część projektu badawczego¹³, po czym w marcu 2015 roku została upubliczniona w postaci projektu o otwartym kodzie źródłowym. Interfejs ten szybko zyskał popularność dzięki łatwości użycia, elastyczności i znakomitej konstrukcji. Aby można było prowadzić skomplikowane obliczenia wymagane przez sieci neuronowe, ta implementacja oparta jest na platformach uczenia głębokiego. Obecnie w tym celu możemy wykorzystać jedną z trzech popularnych bibliotek o otwartym kodzie źródłowym: TensorFlow, Microsoft Cognitive Toolkit (CNTK) i Theano. Zatem dla uniknięcia wszelkich niejasności będziemy nazywać tę implementację referencyjną **wieloplatformowym interfejsem Keras** (ang. *multibackend Keras*).

Pod koniec 2016 roku udostępniono kolejne implementacje interfejsu Keras. Teraz możemy go uruchamiać jako część środowisk Apache MXNet, Apple Core ML, JavaScript czy TypeScript (do działania z poziomu przeglądarki), a także PlainMD (uruchamiany na dowolnej karcie graficznej, niekonicznie firmy Nvidia). Co więcej, sam w sobie moduł TensorFlow został zaopatrzony we własną implementację Keras, `tf.keras`. Korzysta ona wyłącznie z modułu TensorFlow, ale za to zawiera kilka dodatkowych, bardzo przydatnych funkcji (rysunek 10.10), np. obsługuje interfejs API TensorFlow Data, pozwalający na szybkie i skuteczne wczytywanie i przetwarzanie wstępne danych. Z tego powodu będziemy korzystać z implementacji `tf.keras` w niniejszej książce. W tym rozdziale jednak nie będziemy wykorzystywać żadnych funkcji specyficznych dla modułu TensorFlow, dlatego zaprezentowany kod powinien współpracować także z innymi implementacjami Keras (przynajmniej w Pythonie), a jedyne, co trzeba zrobić, to wprowadzić drobne modyfikacje (np. w wierszach importowania).

¹³ Projekt ONEIROS (ang. *Open-Ended Neuro-Electronic Intelligent Robot Operating System* — otwarty neuroelektroniczny inteligentny system operacyjny robotów).



Rysunek 10.10. Dwie implementacje interfejsu Keras: wieloplatformowy interfejs Keras (po lewej) i tf.keras (po prawej)

Najpopularniejszą biblioteką uczenia głębokiego, zaraz po Keras i TensorFlow, jest stworzona przez firmę Facebook biblioteka PyTorch (<https://pytorch.org/>). Dobra wiadomość jest taka, że jej struktura przypomina interfejs Keras (częściowo dlatego, że inspirację dla obydwu interfejsów stanowiły moduły Scikit-Learn i Chainer: <https://chainer.org/>), więc jeśli kiedyś zechcesz przejść z Kerasa na PyTorch, nie będziesz mieć z tym trudności. Popularność biblioteki PyTorch rośnie wykładniczo od 2018 roku, w dużej mierze dzięki prostocie i znakomitej dokumentacji, które nie stanowiły mocnej strony modułu TensorFlow w wersji 1.x. Obecnie jednak moduł TensorFlow 2 jest prawdopodobnie równie prosty jak PyTorch, a także zaadoptował interfejs Keras jako oficjalną, wyspecjalizowaną bibliotekę, natomiast programiści znacznie oczyściли i uprościły pozostałą część API. Również dokumentacja została zupełnie zmodernizowana i znacznie łatwiej teraz znaleźć interesujące nas zagadnienia. Z kolei największe wady biblioteki PyTorch (m.in. ograniczona przenośność i brak analizy grafów obliczeniowych) zostały wyeliminowane w wersji 1.0. Zdrowa konkurencja jest korzystna dla każdego.

W porządku, czas na zabawę z kodem! Interfejs tf.keras stanowi część modułu TensorFlow, dlatego pierwszym etapem jest jego instalacja.

Instalacja modułu TensorFlow 2

Zakładam, że masz zainstalowane środowisko Jupyter i moduł Scikit-Learn zgodnie z wytycznymi zawartymi w rozdziale 2. Teraz również skorzystamy z polecenia pip. Jeżeli stworzyłaś/stworzyłeś środowisko wirtualne za pomocą aplikacji `virtualenv`, należy je najpierw aktywować:

```
$ cd $ML_PATH          # Twój katalog roboczy uczenia maszynowego (np. $HOME/um)
$ source moje_srod/bin/activate # w Linuksie lub macOS
$ .\moje_srod\Scripts\activate # w Windowsie
```

Zainstaluj teraz moduł TensorFlow 2 (jeżeli nie używasz środowiska wirtualnego, musisz skorzystać z praw administratora lub dodać opcję `--user`):

```
$ python3 -m pip install -U tensorflow
```



Jeżeli chcesz korzystać z mocy karty graficznej, to musisz zainstalować moduł tensorflow-gpu zamiast tensorflow — przynajmniej tak jest w czasie, gdy piszę tę książkę, ale zespół twórców już pracuje nad ujednoliceniem bibliotek tak, aby wystarczyła jedna do obsługiwanego systemów zaopatrzonych zarówno w klasyczne procesory, jak i karty graficzne. Do obsługi GPU wymagane są dodatkowe biblioteki (więcej informacji znajdziesz pod adresem <https://www.tensorflow.org/install>). Przyjrzymy się uważniej obsłudze kart graficznych w rozdziale 19.

Aby zweryfikować poprawność instalacji, otwórz powłokę Python lub notatnik Jupyter, po czym importuj klasy TensorFlow i tf.keras. Teraz możesz sprawdzić ich wersje:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

W drugim przypadku widzimy wersję interfejsu Keras zaimplementowaną w module tf.keras. Zwróć uwagę na dopisek -tf, mówiący nam, że został zaimplementowany interfejs Keras wraz z dodatkowymi funkcjami, specyficznymi dla modułu TensorFlow.

Skorzystajmy w końcu z modułu tf.keras. Zaczniemy od zbudowania prostego klasyfikatora obrazów.

Tworzenie klasyfikatora obrazów za pomocą interfejsu sekwencyjnego

Najpierw musimy wczytać zestaw danych. W tym rozdziale wykorzystamy zestaw Fashion MNIST, stanowiący zamiennik klasycznego zestawu MNIST (omówionego w rozdziale 3.). Ma on dokładnie taki sam format jak MNIST (70 000 czarno-białych obrazów o rozmiarze 28×28 pikseli podzielonych na 10 klas), ale obrazy zawierają nie odręcznie pisane cyfry, lecz elementy ubioru, zatem wszystkie klasy okazują się bardziej zróżnicowane, a problem staje się nagle znacznie trudniejszy do rozwiązania. Na przykład zwykły model liniowy osiąga ok. 92% dokładności dla zestawu danych MNIST, ale tylko ok. 83% dla zestawu Fashion MNIST.

Wczytywanie danych za pomocą interfejsu Keras

Keras zawiera pewne funkcje służące do pobierania i wczytywania popularnych zestawów danych, w tym MNIST, Fashion MNIST czy omówiony w rozdziale 2. California Housing. Wczytajmy Fashion MNIST:

```
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
```

Należy pamiętać podczas wczytywania zestawu danych MNIST lub Fashion MNIST za pomocą modułu Keras, że każdy obraz będzie reprezentowany teraz w postaci tablicy o rozmiarze 28×28 , a nie jednowymiarowej tablicy o rozmiarze 784, jak to miało miejsce w przypadku modułu Scikit-Learn. Ponadto nasycenie kolorami jest wyrażone liczbami całkowitymi (od 0 do 255), a nie zmienno-przecinkowymi (od 0,0 do 255,0). Przyjrzymy się wymiarom i typowi danych tworzących zestaw uczący:

```
>>> X_train_full.shape  
(60000, 28, 28)  
>>> X_train_full.dtype  
dtype('uint8')
```

Zestaw danych jest już rozdzielony na zbiór uczący i testowy, ale nigdzie nie widać zbioru walidacyjnego, dlatego go utworzymy. Ponadto będziemy trenować sieć neuronową za pomocą algorytmu gradientu prostego, więc musimy przeskalać cechy wejściowe. Dla uproszczenia przeskaliujemy nasycenie barwami do zakresu od 0 do 1, dzieląc je przez 255,0 (tym samym przekształcimy te wartości do postaci zmienoprzecinkowej):

```
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

W przypadku zestawu danych MNIST wiemy, że gdy etykieta ma wartość 5, znaczy to, że obraz reprezentuje odręcznie zapisaną cyfrę 5. Proste. Jednak w przypadku zestawu Fashion MNIST potrzebna jest nam lista nazw klas, aby wiedzieć, z czym mamy do czynienia:

```
class_names = ["Koszulka", "Spodnie", "Sweter", "Sukienka", "Płaszcz",  
"Sandał", "Koszula", "Tenisówka", "Torebka", "Trzewik"]
```

Na przykład pierwszy obraz w zestawie danych uczących reprezentuje płaszcz:

```
>>> class_names[y_train[0]]  
'Płaszcz'
```

Rysunek 10.11 prezentuje kilka przykładów z zestawu danych Fashion MNIST.



Rysunek 10.11. Przykłady z zestawu Fashion MNIST

Tworzenie modelu za pomocą interfejsu sekwencyjnego

Zajmijmy się budowaniem sieci neuronowej! Oto klasyfikujący perceptron wielowarstwowy zawierający dwie warstwy ukryte:

```
model = keras.models.Sequential()  
model.add(keras.layers.Flatten(input_shape=[28, 28]))  
model.add(keras.layers.Dense(300, activation="relu"))  
model.add(keras.layers.Dense(100, activation="relu"))  
model.add(keras.layers.Dense(10, activation="softmax"))
```

Przeanalizujmy ten listing wiersz po wierszu:

- W pierwszym wierszu tworzymy model sekwencyjny. Jest to najprostsza forma modelu sieci neuronowej dostępna w interfejsie Keras, składająca się z pojedynczego stosu połączonych sekwencyjnie warstw. To tak zwany interfejs sekwencyjny (ang. *sequential API*).
- Następnie przygotowujemy pierwszą warstwę i dodajemy ją do modelu. Jest to warstwa spłaszczona (`Flatten`), która ma za zadanie przekształcić każdy obraz wejściowy w tablicę jednowymiarową — jeżeli otrzyma dane wejściowe X , przeprowadzi operację $X.reshape(-1, 1)$. Warstwa ta nie zawiera żadnych parametrów, a jej jedynym celem jest proste, wstępne przetwarzanie danych. Skoro mamy tu do czynienia z pierwszą warstwą modelu, powinniśmy wyznaczyć wymiary danych wejściowych (`input_shape`), które nie są równoważne rozmiarowi grupy, lecz wymiarom każdego przykładu. Możesz ewentualnie wstawić tu `keras.layers.InputLayer` jako pierwszą warstwę i wyznaczyć `input_shape=[28, 28]`.
- W dalszej kolejności definiujemy warstwę ukrytą gęstą (`Dense`) składającą się z 300 neuronów. Będzie ona korzystać z funkcji aktywacji `ReLU`. Każda warstwa `Dense` zarządza samodzielnie swoją macierzą wag, zawierającą wszystkie wagi połączeń pomiędzy neuronami a wejściami do nich. Zarządza ona także wektorem obciążzeń (po jednym członie obciążenia na każdy neuron). Gdy docierają do niej sygnały wejściowe, zostają one obliczone zgodnie ze wzorem zaprezentowanym w równaniu 10.2.
- Teraz dodajemy kolejną gęstą warstwę ukrytą, zawierającą tym razem 100 neuronów. Tutaj także wprowadzamy funkcję aktywacji `ReLU`.
- Na koniec wyznaczamy gęstą warstwę wyjściową składającą się z 10 neuronów (po jednym na każdą klasę) z wyznaczoną funkcją aktywacji `softmax` (ponieważ klasy wykluczają się wzajemnie).



Zapis `activation="relu"` jest równoważny zapisowi `activation=keras.activations.relu`. Inne funkcje aktywacji są dostępne w pakiecie `keras.activations` — będziemy korzystać z wielu spośród nich w tej książce. Pełną ich listę znajdziesz na stronie <https://keras.io/activations/>.

Zamiast dodawać warstwy pojedynczo możemy przekazać ich listę podczas tworzenia modelu sekwencyjnego:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Metoda `summary()` wyświetla wszystkie warstwy modelu¹⁴, włącznie z nazwą warstwy (która, jeśli jej nie zdefiniujesz własnoręcznie, zostaje wygenerowana automatycznie), wymiary danych wyjściowych (wartość `None` oznacza, że rozmiar grupy może być dowolny), a także liczbę jego parametrów. Na końcu podsumowania znajduje się całkowita liczba parametrów, w tym modyfikowalnych i niemodyfikowalnych (natrafimy na niemodyfikowalne parametry w rozdziale 11.):

¹⁴ Za pomocą metody `keras.utils.plot_model()` możesz wygenerować obraz modelu.

Korzystanie z przykładów umieszczonych w serwisie keras.io

Przykładowe listingi umieszczone w serwisie keras.io będą współpracować z interfejsem `tf.keras`, należy jednak zmodyfikować instrukcje importowania. Spójrz na przykład z serwisu keras.io:

```
from keras.layers import Dense  
output_layer = Dense(10)
```

Instrukcje importowania należy zmienić następująco:

```
from tensorflow.keras.layers import Dense  
output_layer = Dense(10)
```

Jeśli chcesz, możesz ewentualnie zdefiniować pełną ścieżkę:

```
from tensorflow import keras  
output_layer = keras.layers.Dense(10)
```

To ostatnie rozwiązanie jest bardziej rozwlekłe, ale korzystam z niego w tej książce, aby łatwiej Ci było ustalić, które pakiety są wykorzystywane, i uniknąć mylenia klas standardowych z niestandardowymi. W kodzie przeznaczonym do środowiska produkcyjnego wybieram zazwyczaj pierwsze rozwiązanie. Wiele osób stosuje również zapis `from tensorflow.keras import layers`, a po nim `layers.Dense(10)`.

```
>>> model.summary()  
Model: "sequential"  


| Layer (type)      | Output Shape | Param # |
|-------------------|--------------|---------|
| flatten (Flatten) | (None, 784)  | 0       |
| dense (Dense)     | (None, 300)  | 235500  |
| dense_1 (Dense)   | (None, 100)  | 30100   |
| dense_2 (Dense)   | (None, 10)   | 1010    |

  
Total params: 266,610  
Trainable params: 266,610  
Non-trainable params: 0
```

Zwróć uwagę, że warstwy `Dense` często zawierają **mnóstwo** parametrów. Na przykład pierwsza warstwa ukryta ma 784×300 wag połączeń, a do tego 300 członów obciążen, co daje łącznie 235 500 parametrów! Dzięki temu model uzyskuje olbrzymią swobodę w dopasowywaniu do danych uczących, ale jednocześnie grozi mu ryzyko przetrenowania, zwłaszcza w przypadku korzystania z mniejszych zestawów danych. Jeszcze wróćmy do tego zagadnienia.

Możesz z łatwością wyświetlić listę warstw modelu, a także sprawdzić daną warstwę po jej indeksie lub po nazwie:

```
>>> model.layers  
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,  
<tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,  
<tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,  
<tensorflow.python.keras.layers.core.Dense at 0x13240d240>]  
>>> hidden1 = model.layers[1]  
>>> hidden1.name
```

```
'dense'  
>>> model.get_layer('dense') is hidden1  
True
```

Dostęp do wszystkich parametrów modelu uzyskamy za pomocą metod `get_weights()` i `set_weights()`. W przypadku warstwy Dense oznacza to dostęp zarówno do wag połączeń, jak i do członów obciążenia:

```
>>> weights, biases = hidden1.get_weights()  
>>> weights  
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,  
       0.03859074, -0.06889391],  
      ...,  
      [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,  
       0.00272203, -0.06793761]], dtype=float32)  
>>> weights.shape  
(784, 300)  
>>> biases  
array([0., 0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)  
>>> biases.shape  
(300,)
```

Zauważ, że w warstwie Dense połączenia wag zostały zainicjalizowane losowo (jest to warunek konieczny do złamania symetrii, o czym już wiemy), a obciążenia otrzymały wartość 0, co nie stanowi problemu. Jeżeli chcesz zastosować inną metodę inicjalizacji, możesz wyznaczyć parametry `kernel_initializer` (`kernel`, czyli **jądro**, stanowi inną nazwę macierzy wag połączeń) lub `bias_initializer` na etapie tworzenia warstwy. Do inicjalizatorów wróćmy jeszcze w rozdziale 11., ale jeśli chcesz zobaczyć ich pełną listę, znajdziesz ją pod adresem <https://keras.io/initializers/>.



Rozmiar macierzy wag połączeń zależy od liczby wejść. Z tego właśnie powodu zalecam wyznaczanie parametru `input_shape` podczas definiowania pierwszej warstwy w modelu `Sequential`. Nic nie szkodzi jednak, jeżeli nie określisz rozmiaru wejścia — Keras poczeka cierpliwie, dopóki nie określi tego rozmiaru przed skonstruowaniem właściwego modelu. Zrobi to albo w momencie dostarczenia danych (np. podczas uczenia), albo w momencie wywołania metody `build()`. Dopóki nie zostanie utworzony właściwy model, warstwy nie będą miały przyporządkowanych żadnych wag, a Ty nie będziesz w stanie wykonać pewnych czynności (takich jak wyświetlenie podsumowania modelu czy zapisanie modelu). Jeśli więc znasz rozmiar wejścia już na etapie definiowania modelu, to najlepiej go od razu określić.

Kompilowanie modelu

Po utworzeniu modelu musisz wywołać jego metodę `compile()` po to, aby określić funkcję straty i optymalizator. Ewentualnie możesz również wprowadzić listę dodatkowych wskaźników obliczanych podczas uczenia i oceniania modelu:

```
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer="sgd",  
              metrics=["accuracy"])
```



Wyrażenie `loss="sparse_categorical_crossentropy"` jest równoważne konstrukcji `loss=keras.losses.sparse_categorical_crossentropy`. Podobnie w przypadku zapisu `optimizer="sgd"`, który jest tożsamy ze strukturą `optimizer=keras.optimizers.SGD()`, natomiast `metrics=["accuracy"]` stanowi odpowiednik `metrics=[keras.metrics.sparse_categorical_accuracy]` (podczas korzystanie z tej funkcji straty). W niniejszej książce stosuję wiele innych funkcji straty, optymalizatorów i wskaźników, a pełne ich listy znajdziesz na stronach, odpowiednio: <https://keras.io/losses/>, <https://keras.io/optimizers/> i <https://keras.io/metrics/>.

Powyższy kod wymaga wyjaśnienia. Najpierw wyznaczamy funkcję straty "sparse_categorical_crossentropy", ponieważ mamy do czynienia z etykietami rzadkimi (tzn. dla każdego przykładu występuje tylko indeks klasy docelowej, w naszym przypadku od 0 do 9), a klasy wykluczają się wzajemnie. Gdybyśmy mieli natomiast do czynienia z pojedynczym prawdopodobieństwem przynależności do klasy w każdym przykładzie (np. z wektorami gorącojedynkowymi: [0., 0., 0., 1., 0., 0., 0., 0., 0.] reprezentowałby tu klasę 3.), to użylibyśmy funkcji straty "categorical_crossentropy". W przypadku klasyfikacji binarnej (z co najmniej jedną etykietą binarną) zastąpiliśmy funkcję aktywacji "softmax" w warstwie wyjściowej funkcją "sigmoid" (logistyczną), natomiast w miejscu funkcji straty umieścilibyśmy "binary_crossentropy".



Jeżeli chcesz przekształcić etykiety rzadkie (tzn. indeksy klas) w wektor gorącojedynkowy, użyj funkcji `keras.utils.to_categorical()`. Odwrotną operację wykonasz, stosując funkcję `np.argmax()` z atrybutem `axis=1`.

Optymalizator "sgd" oznacza, że trenujemy model za pomocą algorytmu stochastycznego spadku wzdłuż gradientu. Inaczej mówiąc, interfejs Keras przeprowadzi omówiony wcześniej algorytm propagacji wstecznej (tzn. odwrotne różniczkowanie automatyczne wraz z algorytmem gradientu prostego). W rozdziale 11. poznasz skuteczniejsze optymalizatory (usprawniają one metodę gradientu prostego, a nie różniczkowania automatycznego).



Podczas korzystania z optymalizatora SGD bardzo ważną rolę odgrywa strojenie współczynnika uczenia. Dlatego zazwyczaj korzystamy z konstrukcji `optimizer=keras.optimizers.SGD(lr=???)`, gdzie w miejsce znaków zapytania wstawiamy wartość współczynnika uczenia, zamiast zapisu `optimizer="sgd"`, w którym domyślnie `lr=0.01`.

Konstruujemy tu klasyfikator, dlatego warto zmierzyć jego dokładność ("accuracy") podczas jego uczenia i oceniania.

Trenowanie i ocenianie modelu

Model jest teraz gotowy do uczenia. Wystarczy wywołać metodę `fit()`:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                      validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
```

```
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218 - accuracy: 0.7660
                                - val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327
                                - val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192
                                - val_loss: 0.2999 - val_accuracy: 0.8926
```

Podajemy mu cechy wejściowe (`X_train`) i klasy docelowe (`y_train`), a także liczbę epok (domyślnie wyznaczona jest tylko jedna epoka, co zdecydowanie nie wystarczy do osiągnięcia zbieżności przez model). Wykonujemy również przebieg walidacyjny (nie jest to niezbędne). Keras zmierzy funkcję straty i dodatkowe wskaźniki na końcu każdej epoki, co okazuje się bardzo przydatne przy określeniu rzeczywistej wydajności modelu. Jeżeli skuteczność dla zestawu uczącego przewyższa wydajność dla zestawu walidacyjnego, to model prawdopodobnie uległ przetrenowaniu (albo występuje jakaś usterka, na przykład rozbieżność pomiędzy danymi zbioru uczącego i zbioru walidacyjnego).

I to wszystko! Sieć neuronowa została wytrenowana¹⁵. W każdej epoce uczenia Keras wyświetla liczbę dotychczas przetworzonych przykładów (wraz z paskiem postępu), średni czas uczenia na próbę oraz wartość funkcji straty i dokładność (lub dowolne dodatkowe, wyznaczone przez Ciebie wskaźniki), zarówno dla zestawu uczącego, jak i dla walidacyjnego. Jak widać, wartość funkcji straty zmalała podczas uczenia, co stanowi dobry znak, natomiast dokładność walidacji osiągnęła po 30 epokach wartość 89,26%. Nie odbiega to zbytnio od dokładności dla zestawu uczącego, zatem raczej nie jesteśmy tu świadkami przetrenowania.



Zamiast przekazywać zestaw walidacyjny za pomocą argumentu `validation_data`, możesz wyznaczyć w argumencie `validation_split` odsetek danych uczących, które Keras ma użyć do weryfikacji modelu. Na przykład wartość `validation_split=0.1` sprawi, że Keras wyznaczy ostatnie 10% danych (przed przetasowaniem) na walidację.

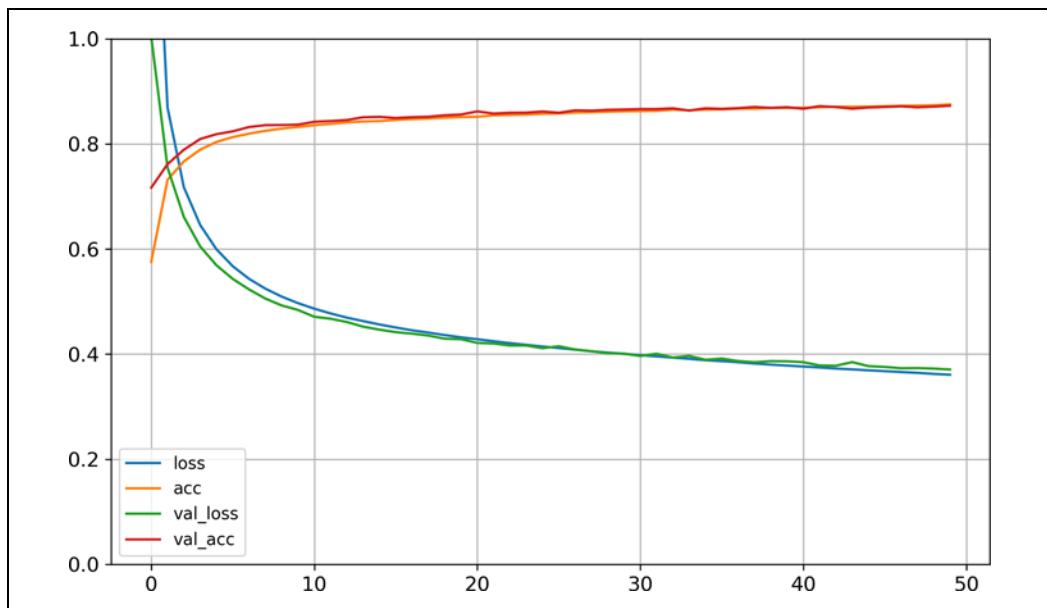
Jeśli zestaw danych uczących jest mocno zniekształccony (czyli jedne klasy są dominujące, a inne są niemal niespotykane), to podczas wywoływania metody `fit()` warto wyznaczyć argument `class_weight` — będzie on nadawał większą wagę klasom mniejszościowym, a klasom dominującym wyznaczy mniejszą wagę. Keras używa tych wag podczas obliczania funkcji straty. Jeżeli wolisz wagi wyznaczone przykładom, wyznacz argument `sample_weight` (po wyznaczeniu obydwu argumentów Keras je pomnoży). Wagi wyznaczone przykładom bywają użyteczne tam, gdzie część przykładów została oznakowana przez ekspertów, a część za pomocą platformy społecznościowej — tym pierwszym należałoby wyznaczyć większą wagę. Możesz także określić wagi przykładów (ale nie wagi klas) w zestawie walidacyjnym poprzez umieszczenie ich jako trzeciego elementu w krotce `validation_data`.

¹⁵ Jeżeli dane uczące lub walidacyjne nie są zgodne z wyznaczonym wymiarem, zostanie wyświetlony wyjątek. Jest to chyba najczęściej występujący błąd, dlatego warto zaznajomić się z treścią komunikatu, który sam w sobie jest zrozumiały — na przykład jeżeli spróbujesz wyuczyć ten model za pomocą tablicy zawierającej „splaszczone” obrazy (`X_train.reshape(-1, 784)`), to ujrzesz następujący wyjątek: `ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)`.

Metoda `fit()` zwraca obiekt `History` zawierający parametry uczenia (`history.params`), listę epok (`history.epoch`) i, co ważniejsze, słownik (`history.history`) z funkcją straty i dodatkowymi wskaźnikami zmierzonymi na końcu każdej epoki w fazie uczenia oraz walidacji (jeżeli występują). Jeżeli za pomocą tego słownika utworzysz obiekt `DataFrame` i wywołasz metodę `plot()`, to zostaną wyświetcone krzywe uczenia zaprezentowane na rysunku 10.12:

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # Wyznacza zakres osi pionowej od 0 do 1
plt.show()
```



Rysunek 10.12. Krzywe uczenia: średnia funkcja straty uczenia (`loss`) i dokładność (`accuracy`) zmierzone w każdej epoce, zarówno dla zestawu uczącego, jak i dla walidacyjnego (`val_loss`, `val_accuracy`)

Jak widać, zarówno dokładność dla zestawu uczącego, jak i dla walidacyjnego stopniowo rosną w miarę uczenia, natomiast błąd uczenia i walidacji maleją. Dobrze! Do tego krzywe dla zestawu walidacyjnego nie odbiegają zbytnio od krzywych dla zestawu uczącego, co oznacza, że nawet jeśli model jest przetrenowany, to nieznacznie. W omawianym przypadku wygląda na to, że na początku treningu model lepiej sprawował się wobec zbioru walidacyjnego niż uczącego. W istocie jest jednak inaczej: wartość błędu walidacji jest obliczana **na końcu** każdej epoki, natomiast wartość błędu uczenia jest uzyskiwana przy użyciu średniej kroczącej **w trakcie** danej epoki. Zatem krzywa dla zestawu uczącego powinna zostać przesunięta o pół epoki w lewo. Jeśli tak zrobisz, to zobaczysz, że na początku uczenia krzywe dla danych uczących i walidacyjnych pokrywają się niemal idealnie.



Gdy tworzysz wykres krzywej dla zestawu uczącego, przesuń go o pół epoki w lewo.

Wydajność dla zestawu danych uczących okazuje się większa w stosunku do wydajności dla zestawu walidacyjnego, co jest całkiem normalne, jeżeli będziesz uczyć model wystarczająco długo. Można zauważać, że model nie uzyskał jeszcze zbieżności, ponieważ funkcja straty dla zestawu walidacyjnego nadal maleje, dlatego prawdopodobnie należałoby kontynuować trenowanie. Wystarczy w tym celu wywołać metodę `fit()`, ponieważ Keras wznowia uczenie od miejsca, w którym zostało przerwane (powinnaś/powinieneś być w stanie zbliżyć się do dokładności walidacyjnej rzędu 89%).

Jeżeli nie zadowala Cię skuteczność modelu, wróć do strojenia hiperparametrów. Pierwszym z nich jest sprawdzenie współczynnika uczenia. Jeśli to nie pomoże, wypróbuj inny optymalizator (zawsze poprawiaj współczynnik uczenia po zmianie dowolnego hiperparametru). Jeśli wydajność dalej pozostawia wiele do życzenia, pobaw się takimi parametrami, jak liczba warstw, liczba neuronów w warstwie czy rodzaj funkcji aktywacji wprowadzanej do każdej warstwy ukrytej. Masz jeszcze do dyspozycji inne parametry, np. rozmiar minigrupy (możesz wyznaczyć go w metodzie `fit()` za pomocą argumentu `batch_size`, którego domyślana wartość wynosi 32). Na końcu rozdziału powrócimy jeszcze do strojenia hiperparametrów. Gdy już uzyskasz wystarczającą dokładność dla zestawu walidacyjnego, sprawdź model na zestawie testowym w celu uzyskania błędu uogólniania przed wdrożeniem modelu do środowiska produkcyjnego. Z łatwością tego dokonasz dzięki metodzie `evaluate()` (obsługuje ona kilka innych argumentów, takich jak `batch_size` czy `sample_weight`; więcej informacji znajdziesz w dokumentacji tej metody).

```
>>> model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

Jak wiesz z rozdziału 2., często skuteczność dla zestawu testowego jest nieco mniejsza niż dla zestawu walidacyjnego, ponieważ parametry są dostrojone do zbioru walidacyjnego, a nie testowego (w naszym przykładzie nie zajmowaliśmy się zbytnio strojeniem hiperparametrów, dlatego niższa wydajność wynika po prostu z nieszczerśliwego przypadku). Pamiętaj, aby zwalczyć pokusę strojenia hiperparametrów dla zestawu testowego, gdyż w przeciwnym razie oszacowanie błędu uogólniania będzie zbyt optymistyczne.

Uzyskiwanie prognoz za pomocą modelu

Skorzystamy teraz z metody `predict()`, dzięki czemu uzyskamy predykcje dla nowych przykładów. Nie dysponujemy niestety prawdziwymi nowymi przykładami, dlatego wykorzystamy po prostu trzy pierwsze przykłady z zestawu testowego:

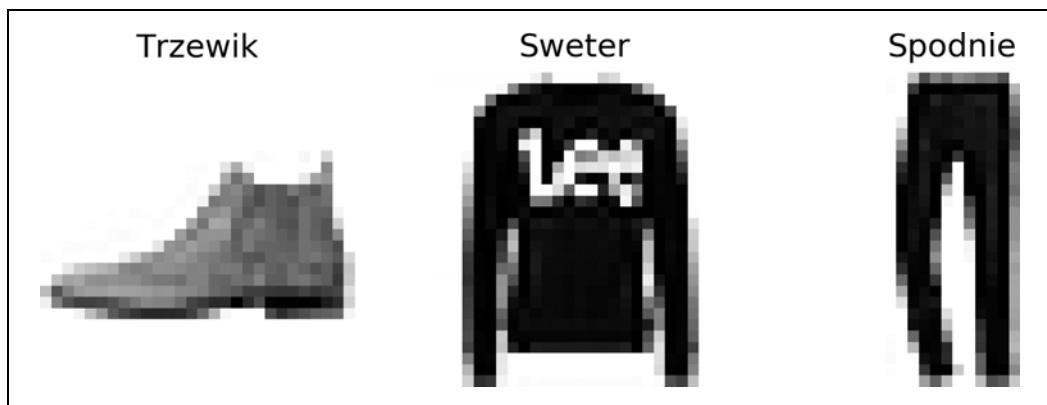
```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

Jak widać, model oszacowuje po jednym prawdopodobieństwie przynależności przykładowu do każdej klasy, od klasy 0. do klasy 9. Na przykład dla pierwszego obrazu zostaje oszacowane 96% prawdopodobieństwa przynależności do klasy 9. (trzewik), 3% dla klasy 5. (sandał) i 1% dla klasy 7. (tenisówka), natomiast wartości prawdopodobieństwa dla pozostałych klas są pomijalne. Inaczej mówiąc, model „wierzy”, że pierwszy obraz symbolizuje obuwie, najprawdopodobniej trzewiki, ale możliwe, że sandały lub tenisówki. Jeżeli interesuje Cię tylko klasa o największym oszacowanym prawdopodobieństwie (nawet jeśli mimo wszystko nie jest ono duże), to możesz wybrać metodę `predict_classes()`:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Trzewik', 'Sweter', 'Spodnie'], dtype='<U11')
```

W tym przypadku klasyfikator w istocie prawidłowo rozpoznał wszystkie trzy obrazy (rysunek 10.13):

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```



Rysunek 10.13. Prawidłowo sklasyfikowane obrazy z zestawu danych Fashion MNIST

Potrafisz już tworzyć, uczyć, oceniać i wykorzystywać klasyfikacyjny perceptron wielowarstwowy oparty na interfejsie sekwencyjnym. Jak to wygląda w przypadku zadań regresji?

Tworzenie regresyjnego perceptronu wielowarstwowego za pomocą interfejsu sekwencyjnego

Przejdźmy do zestawu danych California Housing i użyjmy go w regresyjnej sieci neuronowej. Dla uproszczenia wczytamy dane za pomocą funkcji `fetch_california_housing()` stanowiącej część modułu Scikit-Learn. Uzyskamy dostęp do prostszego zestawu danych w porównaniu do użytego w rozdziale 2., gdyż będzie zawierał wyłącznie cechy numeryczne (czyli jest pozbawiony cechy `ocean_proximity`) i nie ma żadnych brakujących wartości. Po wczytaniu zestawu danych rozdzielimy go na zbiory uczący, walidacyjny i testowy, a także skalujemy wszystkie cechy:

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

```

from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

```

Etapy konstruowania, uczenia, oceniania i używania regresyjnego perceptronu wielowarstwowego do uzyskiwania prognoz za pomocą interfejsu Sequential nie różnią się bardzo od tych etapów w przypadku klasyfikacyjnego perceptronu wielowarstwowego. Podstawową różnicą jest fakt, że w warstwie wyjściowej znajduje się tylko jeden neuron (ponieważ chcemy przewidywać tylko jedną wartość), w którym nie definiujemy funkcji aktywacji, natomiast na funkcję straty wyznaczamy błąd średniokwadratowy. Zestaw danych jest dość mocno zaszumiony, dlatego aby uniknąć przetrenowania, wprowadzimy tylko jedną warstwę ukrytą, składającą się z mniejszej liczby neuronów:

```

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                      validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3] # Udajemy, że są to nowe przykłady
y_pred = model.predict(X_new)

```

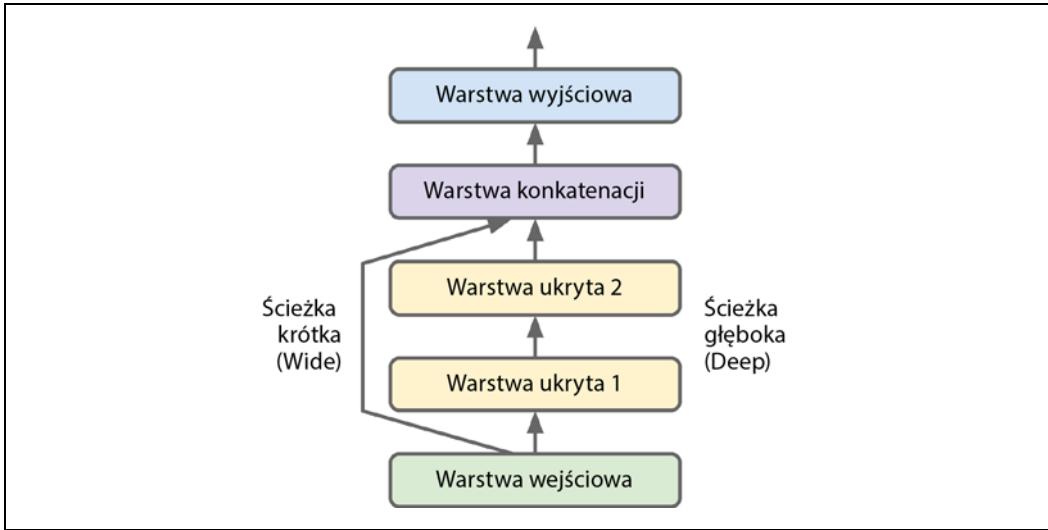
Jak widać, interfejs Sequential jest stosunkowo prosty w użyciu. Jednak mimo że modele sekwencyjne są niezwykle powszechnie, niekiedy lepiej poświęcić czas na stworzenie sieci neuronowych o bardziej skomplikowanej topologii lub zawierających wiele wejść/wyjść. Posłuży nam do tego interfejs funkcyjny.

Tworzenie złożonych modeli za pomocą interfejsu funkcyjnego

Jednym z przykładów niesekwencyjnej sieci neuronowej jest **Wide & Deep**. Architektura ta została zaproponowana w artykule Heng-Tze Chenga i in. z 2016 roku (<https://arxiv.org/abs/1606.07792>)¹⁶. Wszystkie (lub niektóre) wejścia zostają połączone z warstwą wyjściową (rysunek 10.14). Dzięki takiej strukturze sieć neuronowa może poznawać zarówno wzorce głębokie (ścieżka głęboka), jak i proste reguły (poprzez ścieżkę krótką)¹⁷. Dla porównania: w klasycznym perceptronie wielowarstwowym wszystkie dane muszą przechodzić przez wszystkie warstwy sieci, zatem proste wzorce mogą zostać zasłonięte przez sekwencję przekształceń.

¹⁶ Heng-Tze Cheng i in., *Wide & Deep Learning for Recommender Systems*, „Proceedings of the First Workshop on Deep Learning for Recommender Systems” (2016), s. 7 – 10.

¹⁷ Za pomocą ścieżki krótkiej można również wprowadzać do sieci neuronowej sztucznie zaprojektowane cechy.



Rysunek 10.14. Sieć neuronowa Wide & Deep

Przygotujmy taką sieć służącą do rozwiązywania problemu California Housing:

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])
```

Przeanalizujmy poszczególne wiersze kodu:

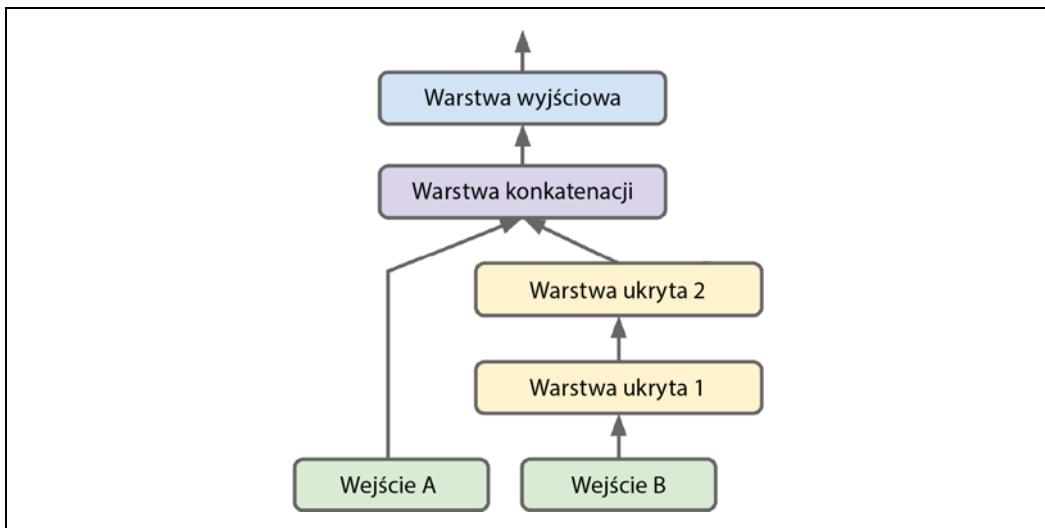
- Musimy najpierw utworzyć obiekt Input¹⁸. Wyznaczamy w nim rodzaj danych wejściowych dostarczanych do modelu, w tym ich rozmiar (shape) i typ (dtype). Jak się niebawem przekonasz, model może zawierać wiele wejść.
- Teraz tworzymy warstwę gęstą (Dense) zawierającą 30 neuronów i wyznaczamy funkcję aktywacji ReLU. Zwróc uwagę, że tuż po jej utworzeniu wywołujemy ją jak funkcję i przekazujemy jej sygnał wejściowy. Dlatego właśnie interfejs ten nazywamy funkcyjnym (ang. *functional API*). Definiujemy tu wyłącznie sposób łączenia warstw; żadne nie są jeszcze przetwarzane.
- Konstruujemy następnie drugą warstwę ukrytą, którą również traktujemy jak funkcję. Przekazujemy jej sygnał wyjściowy z pierwszej warstwy ukrytej.
- Kolejnym etapem jest przygotowanie warstwy konkatenacji (Concatenate) — jej także od razu użyjemy jak funkcji, dzięki której połączymy sygnał wejściowy z wynikiem drugiej warstwy ukrytej. Niektórzy wolą funkcję keras.layers.concatenate(), która tworzy warstwę konkatenacji i od razu wywołuje ją z wyznaczonymi danymi.
- Czas na utworzenie warstwy wyjściowej, zawierającej tylko jeden neuron i pozbawionej funkcji aktywacji. Ją także wywołujemy jako funkcję i przekazujemy jej wynik warstwy konkatenacji.
- Na koniec tworzymy obiekt Model i wyznaczamy, które wejścia i wyjścia mają zostać użyte.

¹⁸ Nazwa `input_` została użyta po to, aby odróżnić ją od domyślnej funkcji `input()` Pythona.

Po skonstruowaniu modelu Keras pozostałe etapy wyglądają identycznie jak w przypadku interfejsu sekwencyjnego, dlatego nie ma potrzeby powtarzać ich opisu: musisz skompilować, wytrenować, ocenić model i użyć go do prognozowania wyników.

A gdybyśmy chcieli wysłać podzbior cech ścieżką krótką, a inny podzbior (być może zawierający część wspólną) ścieżką głęboką (rysunek 10.15)? W takim przypadku możemy wprowadzić wiele wejść. Założymy na przykład, że chcemy przesłać pięć cech ścieżką krótką (cechy od 0. do 4.), a sześć ścieżką głęboką (od 2. do 7.):

```
input_A = keras.layers.Input(shape=[5], name="wejscie_krotkie")
input_B = keras.layers.Input(shape=[6], name="wejscie_glebokie")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="wyjście")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```



Rysunek 10.15. Obsługa wielu wejść

Listing powinien być w pełni zrozumiałym. Należy nadawać nazwy przynajmniej najistotniejszym warstwom, zwłaszcza gdy masz do czynienia ze skomplikowanym modelem. Zwróć uwagę, że podczas tworzenia modelu zdefiniowaliśmy `inputs=[input_A, input_B]`. Możemy teraz tradycyjnie skompilować model, ale w trakcie wywoływania funkcji `fit()` musimy przekazać parę macierzy, (`X_train_A, X_train_B`), po jednej na każde wejście¹⁹. To samo dotyczy `X_valid`, a także `X_test` i `X_new` podczas wywoływania funkcji `evaluate()` lub `predict()`:

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))

X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
```

¹⁹ Możesz ewentualnie przekazać słownik odwzorowujący nazwy wejść na dane wejściowe, np. `{"wejscie_krotkie": X_train_A, "wejscie_glebokie": X_train_B}`. Przydaje się to zwłaszcza w przypadku dużej liczby wejść, gdy można z łatwością pomylić kolejność.

```

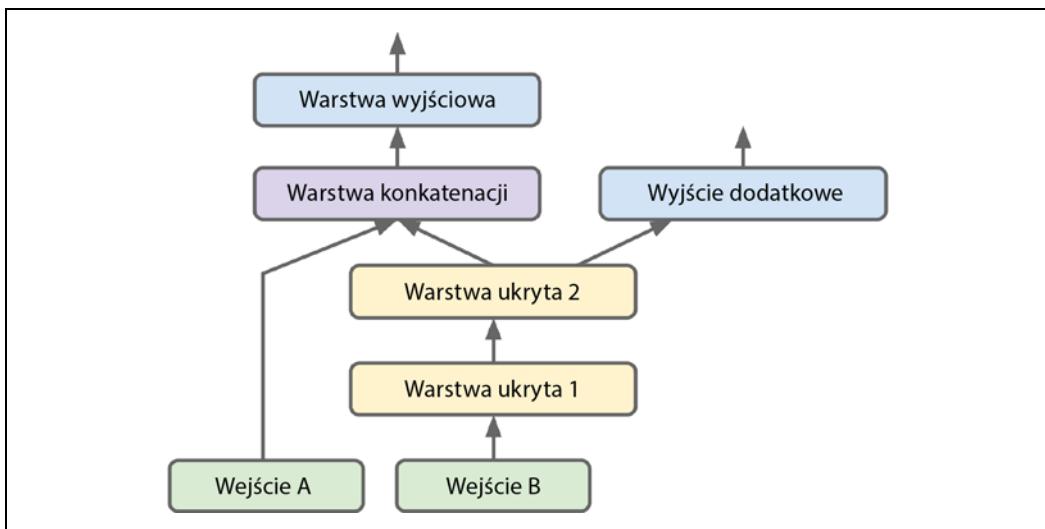
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]

history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))

```

Istnieją różne sytuacje, w których zależy nam na wprowadzeniu wielu wyjść:

- Może tego wymagać dane zadanie. Na przykład musisz znaleźć i zaklasyfikować główny obiekt na zdjęciu. Jest to jednocześnie zadanie regresyjne (określenie współrzędnych środka obiektu, a także jego szerokości i wysokości) i klasyfikacyjne.
- Możesz mieć również do czynienia z wieloma niezależnymi od siebie zadaniami bazującymi na tych samych danych. Jasne, możesz do każdego zadania wytrenować osobną sieć neuronową, ale w wielu przypadkach osiągniesz lepsze rezultaty we wszystkich zadaniach, jeśli wyuczysz sieć neuronową, w której na każde wyjście przypadałoby jedno zadanie. Wynika to z faktu, że sieć neuronowa jest w stanie poznawać cechy danych przydatne w różnych typach zadań. Przykładowo możesz przeprowadzić **klasyfikację wielozadaniową** zdjęć twarzy, w której na jednym wyjściu byłaby rozpoznawana mimika (uśmiech, zaskoczenie itd.), natomiast drugie wyjście odpowiadałoby za określanie, czy dana osoba ma założone okulary.
- Kolejnym zastosowaniem jest technika regularizacji (tzn. ograniczenie nauki, którego celem jest zmniejszenie przetrenowania, a zatem poprawianie zdolności uogólniania). Możesz na przykład dołączyć do struktury sieci dodatkowe wyjście (rysunek 10.16), dzięki czemu głębsza część sieci może rozpoznawać jakieś wzorce bez udziału pozostałej części sieci.



Rysunek 10.16. Obsługa wielu wyjść — w tym przypadku wprowadzamy dodatkowe wyjście w celu regularyzacji

Wprowadzanie dodatkowych wyjść jest całkiem proste: wystarczy podłączyć je do odpowiednich warstw i dodać do listy wyjść modelu. Przykładowo ten oto listing konstruuje sieć zaprezentowaną na rysunku 10.16:

```
[...] # To samo co wcześniej, aż do warstwy wyjścia głównego  
output = keras.layers.Dense(1, name="wyjscie_glowne")(concat)  
aux_output = keras.layers.Dense(1, name="wyjscie_dodatkowe")(hidden2)  
model = keras.Model(inputs=[input_A, input_B], outputs=[output, aux_output])
```

Każde wyjście wymaga zdefiniowania osobnej funkcji straty. Zatem na etapie komplikacji modelu powinniśmy przekazać listę funkcji straty²⁰ (jeżeli przekażemy tylko jedną funkcję straty, Keras uzna, że ma być ona używana we wszystkich warstwach wyjściowych). Domyślnie Keras oblicza wszystkie te funkcje straty i sumuje je, aby uzyskać końcową wartość końcowej funkcji straty modelu. Zależy nam znacznie bardziej na głównej warstwie wyjściowej (gdyż dodatkowa warstwa wyjścia służy jedynie do regularyzacji), dlatego wyznaczmy jej funkcji straty o wiele większą wagę. Na szczęście w trakcie komplikacji modelu możemy zdefiniować wszystkie wagi funkcji straty:

```
model.compile(loss=["mse", "mse"], loss_weights=[0.9, 0.1], optimizer="sgd")
```

Teraz podczas uczenia modelu musimy do każdego wyjścia doprowadzić etykiety. W naszym przykładzie zarówno główna, jak i dodatkowa warstwa wyjściowa powinny próbować prognozować to samo, zatem będą wykorzystywać te same etykiety. Dlatego musimy przekazać nie `y_train`, lecz (`y_train, y_train`) (to samo dotyczy `y_valid` i `y_test`):

```
history = model.fit(  
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,  
    validation_data=(X_valid_A, X_valid_B), [y_valid, y_valid]))
```

Podczas oceniania modelu Keras zwróci całkowitą funkcję straty, a także jej poszczególne składowe:

```
total_loss, main_loss, aux_loss = model.evaluate(  
    [X_test_A, X_test_B], [y_test, y_test])
```

Podobnie metoda `predict()` zwróci prognozy dla każdego wyjścia:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Jak widać, za pomocą interfejsu funkcyjnego można z łatwością tworzyć dowolną strukturę sieci neuronowej. Pozostał nam do omówienia jeszcze jeden sposób konstruowania modeli w bibliotece Keras.

Tworzenie modeli dynamicznych za pomocą interfejsu podklasowego

Zarówno interfejs sekwencyjny, jak i funkcyjny są deklaratywne: musisz najpierw zadeklarować warstwy i ich połączenia, a dopiero potem możesz dostarczać do modelu dane służące do jego uczenia lub uzyskiwania prognoz. Rozwiążanie to ma wiele zalet: taki model można łatwo zapisywać, kopiować i udostępniać, jego strukturę można wyświetlać i analizować i umożliwia ona sprawdzanie

²⁰ Ewentualnie możesz przekazać słownik odwzorowujący każdą nazwę warstwy wyjściowej na odpowiadającą mu funkcję straty. Podobnie jak w analogicznym przypadku z warstwami wejściowymi rozwiązanie to przydaje się przy dużej liczbie wyjść, gdy można łatwo pomylić kolejność. Za pomocą słowników możemy również wyznaczać wagi funkcji straty i wskaźniki (którymi zajmiemy się już niebawem).

wymiarów i typów danych, dzięki czemu wcześniej można wychwytywać błędy (tzn. jeszcze zanim dane przepłyną przez model). Dość prosto również można usuwać błędy z takiego modelu, ponieważ stanowi on po prostu statyczny układ warstw. Na tym jednak polega jego największa wada: jest statyczny. Niektóre modele wymagają użycia pętli, danych różnowymiarowych, rozgałęzień warunkowych i innych mechanizmów dynamicznych. W takich przypadkach lub jeżeli wolisz bardziej imperatywny styl programowania, zastosowanie znajduje **interfejs podklasowy** (ang. *subclassing API*).

Wystarczy utworzyć podklasę klasy `Model`, przygotować warstwy dla konstruktora i wykorzystać je w metodzie `call()` do przeprowadzenia obliczeń. Na przykład wystąpienie poniższej klasy `WideAndDeepModel` stanowi odpowiednik modelu, który skonstruowaliśmy za pomocą interfejsu funkcyjnego. Możesz go skompilować, ocenić i wykorzystać do wyznaczania prognoz tak samo jak wcześniej:

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # Obsługuje standardowe argumenty (np. name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

Przykład ten jest bardzo podobny do modelu utworzonego za pomocą interfejsu funkcyjnego, tutaj jednak nie musimy deklarować warstw wejściowych — wystarczy użyć argumentu `input` w metodzie `call()` i oddzielić fazę budowania warstw²¹ w konstruktorze od fazy ich wykorzystania w metodzie `call()`. Największa różnica polega na tym, że uzyskujesz olbrzymią swobodę wewnętrz metody `call()`: masz dostęp do pętli `for`, instrukcji `if`, ogólnych operacji TensorFlow — jedyne ograniczenie stanowi Twoja wyobraźnia (zob. rozdział 12.)! Oznacza to, że interfejs podkласowy jest znakomitym rozwiązaniem dla badaczy eksperymentujących z nowymi koncepcjami.

Ta dodatkowa swoboda ma swoją cenę: struktura modelu jest ukryta wewnątrz metody `call()`, zatem Keras nie może jej łatwo sprawdzić, nie można jej zapisać ani powieścić, natomiast po wywołaniu metody `summary()` otrzymujemy wyłącznie listę warstw bez informacji na temat ich połączeń. Co więcej, Keras nie jest w stanie sprawdzić wymiarów i typów danych przed uruchomieniem modelu, dlatego łatwiej o pomyłkę. Jeśli więc dodatkowa funkcjonalność nie jest niezbędna, lepiej pozostać przy interfejsie sekwencyjnym lub funkcjonalnym.

²¹ Modele Keras zawierają atrybut `output`, nie możemy więc jej użyć jako nazwy głównej warstwy wyjściowej, dlatego przemianowaliśmy ją na `main_output`.



Modele Keras mogą być traktowane jak pojedyncze warstwy, dlatego możesz z łatwością łączyć je w skomplikowane struktury.

Skoro potrafisz już tworzyć i trenować sieci neuronowe za pomocą interfejsu Keras, będziesz chciał/a chciał/a zapisać.

Zapisywanie i odczytywanie modelu

Jeżeli korzystasz z interfejsu sekwencyjnego lub funkcyjnego, to zapisywanie wytrenowanego modelu nie może być łatwiejsze:

```
model = keras.models.Sequential([...]) # Lub keras.Model([...])
model.compile([...])
model.fit([...])
model.save("moj_model.keras.h5")
```

Keras wykorzystuje format HDF5 do zapisywania zarówno struktury modelu (w tym hiperparametrów każdej warstwy), jak i wartości parametrów modelu w każdej warstwie (tzn. wag połączeń i obciążień). Zapisany zostaje również optymalizator (włącznie z jego hiperparametrami i stanami, jeśli występują).

Zazwyczaj będziesz korzystać ze skryptu uczącego i zapisującego model oraz z co najmniej jednego skryptu (lub usługi sieciowej) wczytującego model i używającego go do uzyskania prognoz. Wczytywanie modelu jest również proste:

```
model = keras.models.load_model("moj_model.keras.h5")
```



Metoda ta działa podczas korzystania z interfejsu sekwencyjnego lub funkcyjnego, lecz okazuje się bezużyteczna w przypadku interfejsu podkласowego. Możesz przy najmniej zapisać (`save_weights()`) i wczytać (`load_weights()`) parametry modelu, ale wszystkie pozostałe elementy musisz własnoręcznie zapisać i wczytać.

Ale co, jeśli uczenie trwa kilka godzin? Taka sytuacja jest powszechnie spotykana, zwłaszcza w przypadku dużych zestawów danych. Należy nie tylko zapisać model po zakończeniu trenowania, lecz również zapisywać punkty kontrolne w równych odstępach czasu podczas uczenia, aby uniknąć utraty wszystkich postępów w razie awarii komputera. Jak jednak sprawić, aby metoda `fit()` zapisywała punkty kontrolne? Użyj wywołań zwrotnych.

Stosowanie wywołań zwrotnych

Metoda `fit()` przyjmuje argument `callbacks` pozwalający określić listę obiektów, które Keras będzie wywoływać na początku i na końcu procesu uczenia, na początku i na końcu każdej epoki, a nawet przed przetwarzaniem i po przetworzeniu każdej grupy danych. Na przykład wywołanie zwrotne `ModelCheckpoint` zapisuje punkty kontrolne modelu w regularnych odstępach czasu w trakcie uczenia, domyślnie na końcu każdej epoki:

```
[...] # Budowanie i kompilowanie modelu
checkpoint_cb = keras.callbacks.ModelCheckpoint("moj_model.keras.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

Ponadto jeżeli korzystasz ze zbioru walidacyjnego w trakcie uczenia, możesz wyznaczyć `save_best_only=True` podczas tworzenia wywołania `ModelCheckpoint`. Oznacza to, że zostanie zapisany tylko ten model, który osiągnie największą skuteczność na zestawie walidacyjnym. W ten sposób nie musisz martwić się o to, że proces uczenia trwa zbyt długo i że model ulegnie przetrenowaniu — wystarczy wczytać ostatni zachowany model po zakończeniu treningu, gdyż będzie to model, który uzyskał najlepsze wyniki. Oto prosta implementacja techniki wczesnego zatrzymywania (zob. rozdział 4.):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("moj_model_keras.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb])
model = keras.models.load_model("moj_model_keras.h5") # Wraca do najlepszego modelu
```

Innym sposobem implementacji wczesnego zatrzymywania jest wprowadzenie wywołania zwrotnego `EarlyStopping`. Przerwie ono proces uczenia w momencie, gdy przez określoną liczbę epok (zdefiniowaną za pomocą argumentu `patience`) nie wykryje poprawy skuteczności na zbiorze walidacyjnym, a dodatkowo może także powrócić do najlepszego dotychczasowego modelu. Możesz połączyć obydwa wywołania zwrotne po to, aby zapisywać punkty kontrolne modelu (na wypadek awarii komputera) i wcześnie przerywać naukę w razie braku postępów (oszczędność czasu i zasobów):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                    restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

Możesz wyznaczyć dużą liczbę epok, gdyż proces uczenia zostanie zakończony automatycznie w przypadku braku postępów. W takiej sytuacji nie ma potrzeby wczytywania najlepszego zapisanego modelu, ponieważ wywołanie zwrotne `EarlyStopping` będzie śledzić najlepsze wagи i wczyta je po zakończeniu treningu.



W pakiecie `keras.callbacks` znajdziesz wiele innych wywołań zwrotnych (<https://keras.io/callbacks/>).

Jeżeli potrzebujesz dodatkowej kontroli, możesz bez problemu stworzyć własne wywołania zwrotne. Przykładowo zaprezentowane poniżej niestandardowe wywołanie zwrotne będzie wyświetlać stosunek pomiędzy funkcją straty dla zestawu walidacyjnego a funkcją straty dla zestawu uczącego w fazie uczenia (np. w celu wykrycia przetrenowania):

```
class PrintValTrainRatioCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("\nwalidacja/uczenie: {:.2f}".format(logs["val_loss"] / logs["loss"]))
```

Jak łatwo przewidzieć, możesz zaimplementować wywołania zwrotne `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()` i `on_batch_end()`. W razie potrzeby (np. podczas usuwania błędów) można je również stosować w fazach oceniania i wnioskowania. Na etapie oceniania należy implementować `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()` lub `on_test_batch_end()` (wywoływanie przez `evaluate()`), natomiast podczas prognozowania —

`on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()` lub `on_predict_batch_end()` (wywoływanie przez `predict()`).

Przyjrzyjmy się jeszcze jednemu narzędziu, które trzeba znać, korzystając z interfejsu `tf.keras`: `TensorBoard`.

Wizualizacja danych za pomocą narzędzia `TensorBoard`

`TensorBoard` to znakomite narzędzie do interaktywnej wizualizacji, umożliwiające obserwację krzywych uczenia podczas trenowania modelu, porównywanie krzywych uczenia pomiędzy wieloma przebiegami, wizualizowanie grafów obliczeniowych, analizowanie statystyk uczenia, przeglądanie obrazów generowanych przez model, wizualizowanie złożonych, wielowymiarowych danych, automatycznie rzutowanych na wykres trójwymiarowy i poddawanych analizie skupień, a to tylko część możliwości! Narzędzie to zostaje automatycznie zainstalowane wraz z modułem `TensorFlow`, więc znajduje się już na Twoim komputerze.

Aby móc z niego skorzystać, musisz tak zmodyfikować swoją aplikację, aby zapisywała dane do zwizualizowania w specjalnych, binarnych plikach dziennika zwanych **plikami zdarzeń** (ang. *event files*). Każdy rekord danych binarnych nosi nazwę **podsumowania** (ang. *summary*). Serwer `TensorBoard` monitoruje katalog dziennika zdarzeń, automatycznie wychwytuje zmiany i aktualizuje wizualizacje — dzięki temu możesz wizualizować dane na bieżąco (z nieznacznym opóźnieniem), takie jak krzywe uczenia w trakcie trenowania modelu. Zasadniczo należy wskazać serwerowi `TensorBoard` główny katalog dzienników zdarzeń i tak skonfigurować swoją aplikację, aby przy każdym uruchomieniu zapisywała pliki zdarzeń do osobnych podkatalogów. W ten sposób ta sama instancja serwera `TensorBoard` będzie w stanie wizualizować i porównywać dane z wielu przebiegów programu bez ryzyka pomieszania danych.

Zaczniemy od zdefiniowania katalogu głównego, w którym będziemy przechowywać dzienniki zdarzeń `TensorFlow`, a także niewielkiej funkcji generującej ścieżkę podkatalogu za pomocą bieżącej daty i godziny, dzięki czemu będzie ona za każdym razem inna. Możesz dodać dodatkowe informacje do ścieżki katalogu, takie jak testowane wartości hiperparametrów modelu, co ułatwi Ci rozeznanie w zalewie informacji:

```
import os
root_logdir = os.path.join(os.curdir, "moje_dzienniki")

def get_run_logdir():
    import time
    run_id = time.strftime("przebieg_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # Np. './moje_dzienniki/przebieg_2019_06_07-15_15_22'
```

Dobra informacja jest taka, że Keras zawiera przydatne wywołanie zwrotne `TensorBoard()`:

```
[...] # Budowanie i kompilowanie modelu
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                      validation_data=(X_valid, y_valid),
                      callbacks=[tensorboard_cb])
```

I to tyle! Nie może już być prościej. Po uruchomieniu tego listingu wywołanie zwrotne `TensorBoard()` zajmie się tworzeniem katalogu dzienników zdarzeń (w razie potrzeby wraz z katalogami nadzrodnymi), a w trakcie uczenia modelu zajmie się generowaniem plików zdarzeń i zapisywaniem w nich podsumowań. Po uruchomieniu programu drugi raz (być może zmieniłaś/zmieniłeś wartość jakiegoś hiperparametru) zostanie utworzona mniej więcej taka struktura katalogowa:

```
moje_dzienniki/
└── przebieg_2019_06_07-15_15_22
    ├── train
    │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
    │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
    │   └── plugins/profile/2019-06-07_15-15-32
    │       └── local.trace
    ├── validation
    │   └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
    └── przebieg_2019_06_07-15_15_49
        └── [...]
```

Na każdy przebieg zostaje przeznaczony osobny katalog, w którego skład wchodzą podkatalogi zawierające dzienniki zdarzeń uczenia i walidacji. W obydwu tych podkatalogach mieszą się pliki zdarzeń, ale dzienniki zdarzeń uczenia zawierają także śledzenie profilowania — w ten sposób `TensorBoard` może sprawdzić, ile czasu model poświęcił na poszczególne etapy na różnych urządzeniach, co stanowi doskonałe narzędzia wykrywania zatorów.

Należy teraz uruchomić serwer `TensorBoard`. Możesz to zrobić za pomocą odpowiedniego polecenia w terminalu. Jeżeli masz zainstalowany `TensorFlow` w środowisku wirtualnym, uaktywnij je teraz. Następnie wpisz następującą komendę w katalogu głównym projektu (lub gdziekolwiek indziej, pod warunkiem że określiłeś właściwy katalog dzienników zdarzeń):

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/ (Press CTRL+C to quit)
```

Jeżeli powłoka nie jest w stanie znaleźć skryptu `tensorboard`, musisz zaktualizować zmienną środowiskową `PATH` tak, aby zawierała katalog, w którym został zainstalowany ten skrypt (ewentualnie możesz zastąpić `tensorboard` w wierszu polecenia komendą `python3 -m tensorflow.main`). Po uruchomieniu serwera wystarczy otworzyć przeglądarkę i wprowadzić adres `http://localhost:6006`.

Możesz także wykorzystać `TensorBoard` bezpośrednio w notatniku Jupyter za pomocą poniższych poleceń. Pierwszy wiersz wczytuje rozszerzenie `TensorBoard`, natomiast drugi uruchamia serwer `TensorBoard`, udostępniając porty 6006 (jeżeli nie został jeszcze uruchomiony) i się z nim łączy:

```
%load_ext tensorboard
%tensorboard --logdir=./moje_dzienniki --port=6006
```

W każdym przypadku Twoim oczom powinien ukazać się interfejs sieciowy `TensorBoard`. Kliknij zakładkę `SCALARS`, aby ujrzeć wykres krzywych uczenia (rysunek 10.17). W lewym dolnym rogu wybierz interesujące Cię dzienniki (np. dzienniki uczenia z trzeciego i czwartego przebiegu) i kliknij skalar `epoch_loss`. Zwróć uwagę, że funkcja straty uczenia ładnie mała w obydwu przebiegach, ale w drugim proces ten trwał krócej. Rzeczywiście, wprowadziliśmy tu współczynnik uczenia równy 0,05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) zamiast 0,001.



Rysunek 10.17. Wizualizowanie krzywych uczenia w TensorBoard

Możesz także zwizualizować cały graf, wyuczone wagi (rzutowane na wykres trójwymiarowy) lub śledzenie profilowania. Wywołanie zwrotne `TensorBoard()` zawiera również opcje pozwalające na śledzenie dodatkowych danych, takich jak wektory właściwościowe (zob. rozdział 13.).

Oprócz tego TensorFlow zawiera ogólny interfejs API w pakiecie `tf.summary`. Poniższy listing tworzy obiekt `SummaryWriter` za pomocą funkcji `create_file_writer()`; obiekt ten nadaje kontekst dziennikom zdarzeń przechowującym skalary, histogramy, obrazy, dźwięki i teksty, które można następnie zwizualizować w środowisku TensorBoard (przekonaj się sama/sam!):

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("moj_skalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100 # Jakiś dane losowe
        tf.summary.histogram("moj_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3) # Losowe obrazy 32x32 RGB
        tf.summary.image("moje_obrazy", images * step / 1000, step=step)
        texts = ["Skok wynosi " + str(step), "Jego kwadrat to " + str(step**2)]
        tf.summary.text("moj_tekst", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("moj_dzwiek", audio, sample_rate=48000, step=step)
```

W rzeczywistości jest to bardzo przydatne narzędzie wizualizujące, którego zastosowanie wykracza poza moduł TensorFlow czy nawet uczenie głębokie.

Podsumujmy wiedzę zdobytą dotychczas dzięki lekturze tego rozdziału. Przedstawiłem odrobinę historii sieci neuronowych, a także omówiłem perceptrony wielowarstwowe i ich zastosowanie w zadaniach klasifikacji i regresji. Pokazałem, jak budować perceptrony wielowarstwowe za pomocą interfejsu sekwencyjnego stanowiącego część modułu `tf.keras`, a także jak tworzyć bardziej zaawansowane struktury modeli przy użyciu interfejsów funkcyjnego i podklasowego. Umiesz już zapisywać i wczytywać model oraz wykorzystywać wywoływanie zwrotne do zapisywania punktów kontrolnych modelu, wczesnego zatrzymywania itd. Wiesz również, jak korzystać z narzędzia TensorBoard do

wizualizowania danych. Już teraz jesteś w stanie rozwiązać wiele problemów za pomocą sieci neuronowych! Być może jednak zastanawiasz się, w jaki sposób dobierać liczbę warstw ukrytych, liczbę neuronów tworzących sieć oraz wszelkie inne hiperparametry. Teraz właśnie zajmiemy się tym zagadnieniem.

Doszajanie hiperparametrów sieci neuronowej

Elastyczność sieci neuronowych stanowi jednocześnie jedną z ich największych wad: konieczne jest wyznaczenie olbrzymiej liczby hiperparametrów. Możesz nie tylko stworzyć dowolnie rozbudowaną sieć neuronową, ale nawet w prostym perceptronie wielowarstwowym masz możliwość modyfikowania liczby warstw, neuronów w każdej warstwie, rodzaju funkcji aktywacji używanej w każdej warstwie, logiki inicjalizowania wag itd. Skąd mamy wiedzieć, jaka kombinacja hiperparametrów jest najlepsza do realizacji określonego zadania?

Jednym z rozwiązań jest wypróbowanie wielu kombinacji hiperparametrów i sprawdzenie, które z nich dają najlepsze wyniki na zbiorze walidacyjnym (lub wykorzystania k-krotnego sprawdzianu krzyżowego). Na przykład możemy użyć GridSearchCV lub RandomizedSearchCV do przeszukiwania przestrzeni hiperparametrów, podobnie jak to robiliśmy w rozdziale 2. W tym celu musimy umieścić modele Keras w obiektach przypominających standardowe regresory Scikit-Learn. Najpierw utworzymy funkcję tworzącą i komplilującą model Keras dla zadanego zbioru hiperparametrów:

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

Funkcja ta tworzy prosty model sekwencyjny regresji jednoczynnikowej (tylko jeden neuron wyjściowy) przy zadany wymiarze danych wejściowych i liczbie warstw ukrytych i neuronów, a następnie kompiluje go za pomocą optymalizatora SGD o określonym współczynniku uczenia. Dobrym rozwiązaniem jest wprowadzanie rozsądnych wartości domyślnych przy jak największej liczbie hiperparametrów, podobnie jak to robi moduł Scikit-Learn.

Utwórzmy teraz KerasRegressor oparty na tej funkcji build_model():

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

Obiekt KerasRegressor jest klasą opakowującą model Keras utworzony za pomocą funkcji build_model(). Nie zdefiniowaliśmy tu żadnych hiperparametrów, dlatego zostaną użyte domyślne parametry zawarte wewnętrz funkcji build_model(). Teraz możemy użyć tego obiektu jak zwykłego regresora Scikit-Learn — wytrenujemy go za pomocą metody fit(), następnie ocenimy przy użyciu metody score(), a potem możemy uzyskać prognozy, korzystając z metody predict():

```
keras_reg.fit(X_train, y_train, epochs=100,
               validation_data=(X_valid, y_valid),
               callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Zauważ, że każdy dodatkowy parametr przekazany metodzie `fit()` trafi do modelu Keras. Do tego wynik będzie stanowił przeciwność błędu średniokwadratowego, ponieważ moduł Scikit-Learn akceptuje wyniki, a nie funkcje straty (im wyższy, tym lepszy).

Nie chcemy uczyć i oceniać modelu w ten sposób, lecz zależy nam na sprawdzeniu setek jego wariantów i wybraniu tego, który będzie miał najlepszą skuteczność dla zbioru walidacyjnego. Mamy do czynienia z mnóstwem hiperparametrów, dlatego lepiej zastosować wyszukiwanie losowe zamiast przeszukiwania siatki (zob. rozdział 2.). Spróbujmy określić liczbę warstw ukrytych i neuronów oraz współczynnik uczenia:

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distrib = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distrib, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                   validation_data=(X_valid, y_valid),
                   callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

To samo widzieliśmy już w rozdziale 2., tutaj jednak wprowadzamy jeszcze dodatkowe parametry do metody `fit()`, a one zostają przekazane do modeli Keras. Zwróć uwagę, że `RandomizedSearchCV` wykorzystuje k-krotny sprawdzian krzyżowy, dlatego nie są tu używane `X_valid` i `y_valid`, które mają zastosowanie jedynie we wcześnieym zatrzymywaniu.

W zależności od sprzętu, rozmiaru zestawu danych, złożoności modelu oraz wartości `n_iter` i `cv` poszukiwania mogą trwać wiele godzin. Po ich zakończeniu możesz sprawdzić optymalne wartości parametrów, najlepszy wynik i wytrenowany model Keras:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

Mögesz teraz zapisać model, ocenić go na zestawie testowym i, jeżeli jesteś zadowolona/zadowolony z wyników, umieścić go w środowisku produkcyjnym. Korzystanie z przeszukiwania losowego nie jest zbyt skomplikowane i okazuje się skuteczne w przypadku wielu dość prostych problemów. Jeżeli jednak proces uczenia przebiega powoli (np. masz do czynienia z bardziej złożonymi problemami i dużymi zestawami danych), to technika ta przeanalizuje jedynie niewielki obszar przestrzeni hiperparametrów. Mogesz częściowo rozwiązać ten problem poprzez ręczne wspomaganie przeszukiwania: najpierw uruchom szybkie przeszukiwanie losowe z wyznaczonym dużym zakresem wartości hiperparametrów, po czym wykonaj następne przeszukiwanie na mniejszym zakresie wartości wyśrodkowanych na najlepszych wartościach wyznaczonych podczas pierwszego przebiegu itd. W ten sposób powinno Ci się udać znaleźć dobry zbiór hiperparametrów. Rozwiązanie to jest jednak bardzo czasochłonne i raczej wolisz wykorzystać ten czas inaczej.

Na szczęście istnieje wiele technik przeszukiwania przestrzeni hiperparametrów znacznie skuteczniejszych od przeszukiwania losowego. Ich podstawowa zasada jest prosta: jeżeli jakiś obszar przestrzeni okazuje się dobry, należy go dokładniej przeanalizować. Techniki te wykorzystują zjawisko „ogniskowania” i dzięki niemu uzyskują lepsze wyniki w znacznie krótszym czasie. Oto lista niektórych bibliotek Pythona służących do optymalizowania hiperparametrów:

Hyperopt (<https://github.com/hyperopt/hyperopt>)

Popularna biblioteka optymalizującą wszystkie typy złożonych przestrzeni przeszukiwania (w tym przestrzeni wartości rzeczywistych, takich jak wyznaczające współczynnik uczenia, i przestrzeni wartości dyskretnych, określających np. liczbę warstw).

Hyperas (<https://github.com/maxpumperla/hyperas>), **kopt** (<https://github.com/Avsecz/kopt>) lub **Talos** (<https://github.com/autonomio/talos>)

Przydatne biblioteki, które służą do optymalizowania hiperparametrów w modelach Keras (pierwsze dwie bazują na bibliotece Hyperopt).

Keras Tuner (<https://www.youtube.com/watch?v=Un0JDL3i5Hg&t=24s>)

Stworzona przez firmę Google przystępna biblioteka optymalizacji hiperparametrów, przeznaczona do modeli Keras. Zawiera usługę wizualizacji i analizy.

Scikit-Optimize, inaczej **skopt** (<https://scikit-optimize.github.io/>)

Ogólna biblioteka optymalizująca. Klasa BayesSearchCV przeprowadza optymalizację bayesowską za pomocą interfejsu przypominającego GridSearchCV.

Spearmint (<https://github.com/JasperSnoek/spearmint>)

Biblioteka optymalizacji bayesowskiej.

Hyperband (<https://github.com/zygmuntz/hyperband>)

Szybka biblioteka strojenia hiperparametrycznego bazująca na informacjach zawartych w niedawno opublikowanym artykule *Hyperband* Lishy Li i in. (<https://arxiv.org/abs/1603.06560>)²².

Sklearn-Deap (<https://github.com/rsteca/sklearn-deap>)

Biblioteka optymalizacji hiperparametrów oparta na algorytmach genetycznych, zawierająca interfejs przypominający GridSearchCV.

Ponadto coraz więcej firm oferuje usługi optymalizowania hiperparametrów. W rozdziale 19. poznasz taką usługę dostępną na Cloud AI Platform firmy Google (<https://cloud.google.com/ml-engine/docs/using-hyperparameter-tuning>). Innymi przykładami są usługi firmy Arimo (<https://arimo.com/>), SigOpt (<https://sigopt.com/>) czy usługa Oscar firmy CallDesk (<http://oscar.calldesk.ai/>).

Strojenie hiperparametrów stanowi obecnie dział uczenia maszynowego, któremu badacze poświęcają mnóstwo uwagi, a do tego algorytmy ewolucyjne powoli wracają do łask. Zapoznaj się na przykład

²² Lisha Li i in., *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*, „Journal of Machine Learning Research” 18 (April 2018), s. 1 – 52.

z rewelacyjnym artykułem firmy DeepMind z 2017 roku (<https://arxiv.org/abs/1711.09846>)²³, w którym autorzy optymalizują jednocześnie populację modeli i ich hiperparametrów. Firma Google również wykorzystała strategię ewolucyjną nie tylko w procesie przeszukiwania hiperparametrów, lecz również wyznaczania najlepszej struktury sieci neuronowej do rozwiązania danego problemu; pakiet AutoML jest już dostępny jako usługa sieciowa (<https://cloud.google.com/automl/>). Może wkrótce nie będziemy już musieli własnoręcznie budować sieci neuronowych? Warto zapoznać się z wpisem firmy Google (<https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html>) na ten temat. W rzeczywistości algorytmy ewolucyjne były już stosowane z powodzeniem do uczenia pojedynczych sieci neuronowych i zastępowały wszechobecną technikę gradientu prostego! Dowód na to znajdziesz chociażby we wpisie firmy Ubon z 2017 roku (<https://eng.uber.com/deep-neuroevolution/>), w którym autorzy zaprezentowali technikę **neuroewolucji głębokiej** (ang. *Deep Neuroevolution*).

Jednak pomimo wszystkich tych ekscytujących postępów, narzędzi i usług warto wiedzieć, jakie są rozsądne wartości każdego hiperparametru, co pozwoli szybko stworzyć prototyp i ograniczyć przestrzeń przeszukiwania. W dalszej części rozdziału znajdziesz wytyczne dotyczące doboru liczby warstw ukrytych i neuronów w perceptronie wielowarstwowym oraz wyznaczania dobrych wartości niektórych podstawowych hiperparametrów.

Liczba warstw ukrytych

W przypadku wielu problemów warto rozpocząć od pojedynczej warstwy ukrytej, gdyż może ona dawać dobre rezultaty. Perceptron wielowarstwowy zawierający tylko jedną warstwę ukrytą może w teorii modelować każdą skomplikowaną funkcję pod warunkiem, że zawiera wystarczająco wiele neuronów. Jednak w przypadku skomplikowanych problemów sieci głębokie mają znacznie większą **efektywność parametryczną** (ang. *parameter efficiency*) w porównaniu do sieci płytowych: są one w stanie modelować złożone funkcje za pomocą wykładniczo mniejszej liczby neuronów, co oznacza, że mogą osiągnąć dużo lepszą wydajność przy użyciu takiej samej ilości danych uczących.

Aby to zrozumieć, założmy, że poproszono Cię o narysowanie lasu w jakimś programie graficznym, ale zabroniono korzystać z funkcji kopiowania i klejenia. Zadanie to wymagałoby olbrzymiej ilości czasu: trzeba by rysować każde drzewo osobno, gałąź po gałęzi, liść za liściem. Gdyby można było narysować jeden liść, skopiować go wielokrotnie, aby zapełnić gałąź, a następnie skopiować wielokrotnie tę gałąź, powierzone zadanie udałoby się ukończyć w mgnieniu oka. Rzeczywiste dane są często uporządkowane w taki hierarchiczny sposób i sieci neuronowe mogą natychmiastowo wykorzystać tę właściwość: niższe warstwy ukryte modelują ogólne struktury (np. linie zarysów i kierunki ułożenia), warstwy środkowe łączą te cechy ogólne w struktury pośrednie (np. kwadraty, okręgi), a z tych pośrednich składników górne warstwy ukryte tworzą szczegółowe struktury (np. twarze).

Taka struktura hierarchiczna nie tylko ułatwia osiąganie zbieżności przez sieci DNN, lecz także zwiększa ich zdolność uogólniania do nowych zestawów danych. Na przykład jeżeli już wytrenowałeś/wytrenowałeś model do rozpoznawania twarzy na zdjęciach i chcesz teraz wyuczyć kolejną sieć, rozpoznającą tym razem fryzury, możesz wykorzystać dolne warstwy pierwszej sieci. Nie musisz inicjalizować losowo wag i obciążać kilku pierwszych warstw nowej sieci, lecz możesz wykorzy-

²³ Max Jaderberg i in., *Population Based Training of Neural Networks*, arXiv preprint arXiv:1711.09846 (2017).

stać wartości dobrane w dolnych warstwach sieci rozpoznającej twarze. Nowa sieć nie będzie musiała poznawać od nowa ogólnych struktur wspólnych dla większości zdjęć, lecz będzie musiała się nauczyć rozpoznawać jedynie bardziej szczegółowe struktury (w tym przypadku fryzury). Jest to tak zwane **uczenie transferowe** (ang. *transfer learning*).

Podsumowując, w przypadku wielu problemów wystarczy na początku zaprojektować jedną lub dwie warstwy ukryte i sieć neuronowa powinna spisywać się jak należy. Przykładowo jesteś w stanie z łatwością osiągnąć dokładność przekraczającą 97% dla zestawu MNIST za pomocą zaledwie jednej warstwy ukrytej składającej się z kilkuset neuronów i ponad 98% po wprowadzeniu dwóch warstw ukrytych o takiej samej całkowitej liczbie neuronów i dla zbliżonego czasu uczenia. W przypadku bardziej skomplikowanych problemów możesz zwiększyć liczbę warstw aż do przetrenowania względem zestawu uczącego. Bardzo złożone zadania, takie jak klasyfikacja dużych obrazów lub rozpoznawanie mowy, wymagają zazwyczaj sieci składających się z dziesiątek warstw (a nawet setek, które jednak nie są w pełni połączone, o czym przekonasz się w rozdziale 14.) oraz olbrzymich zestawów danych. Rzadko trzeba trenować takie sieci od podstaw — o wiele częściej mamy do czynienia z wielokrotnym wykorzystywaniem uprzednio wyuczonej, nowoczesnej sieci przeznaczonej do realizacji podobnego zadania. Dzięki temu uczenie będzie przebiegać znacznie szybciej i będzie wymagać o wiele mniej danych (wróćmy do tego tematu w rozdziale 11.).

Liczba neuronów w poszczególnych warstwach ukrytych

Liczbę neuronów w warstwie wejściowej i warstwie wyjściowej określa typ danych wejściowych i wyjściowych wymaganych przez zadanie. Na przykład klasyfikacja obrazów MNIST wymaga $28 \times 28 = 784$ neurony wejściowe i 10 neuronów wyjściowych.

Warstwy ukryte tworzono kiedyś na wzór piramidy: w każdej kolejnej warstwie występowało coraz mniej neuronów. Tłumaczono to hipotezą, zgodnie z którą wiele cech ogólnych łączy się w znacznie mniejszą liczbę cech szczegółowych. Typowa sieć klasyfikująca obrazy MNIST mogła składać się z trzech warstw ukrytych, z których pierwsza zawierała 300, druga 200, a trzecia 100 neuronów. Porzucono jednak to rozwiązanie, ponieważ zauważono, że w większości przypadków taka sama liczba neuronów w każdej warstwie daje takie same, a nawet lepsze rezultaty; do tego zamiast trzech hiperparametrów (po jednym na warstwę) wystarczy dostroić jeden. Mimo to czasami, w zależności od zestawu danych, warto przygotować pierwszą warstwę ukrytą większą od pozostałych.

Możesz, podobnie jak w przypadku warstw sieci, stopniowo zwiększać liczbę neuronów w każdej warstwie aż do pojawienia się zjawiska przetrenowania. W praktyce jednak jest dużo prościej i skuteczniej wybrać model zawierający większą liczbę warstw i neuronów, niż jest wymagane w danym zadaniu, a następnie zastosować wczesne zatrzymywanie lub inną technikę regularyzacyjną do uniknięcia przetrenowania. Vincent Vanhoucke, naukowiec z firmy Google, nazwał to rozwiązanie strategią „rozciągliwych portek” (ang. *stretch pants approach*): nie marnuj czasu na znalezienie portek w odpowiednim rozmiarze, tylko użyj rozciągliwych portek, które można dopasować do właściwego rozmiaru. W ten sposób możesz uniknąć warstw „zatorowych”, które mogą zrujnować cały model. Z drugiej strony, jeśli warstwa będzie miała zbyt mało neuronów, nie będzie dysponować wystarczającą siłą reprezentacyjną, która pozwoli zachowywać wszystkie niezbędne informacje z danych wejściowych (tzn. warstwa zawierająca dwa neurony może przekazywać jedynie sygnał dwuwymiarowy,

więc jeżeli otrzyma dane trójwymiarowe, część informacji zostanie utraconych). Bez względu na rozmiar i możliwości pozostałej części sieci nie da się już odzyskać tych utraconych informacji.



Zasadniczo bardziej opłaca się zwiększać liczbę warstw niż neuronów w poszczególnych warstwach.

Współczynnik uczenia, rozmiar grupy i pozostałe hiperparametry

Liczba warstw ukrytych i neuronów to nie jedyne hiperparametry, które można dostroić w perceptronie wielowarstwowym. Oto najważniejsze z nich, a także kilka porad, w jaki sposób dobierać ich wartości:

Współczynnik uczenia

Współczynnik uczenia jest bodaj najważniejszym hiperparametrem. Zasadniczo optymalny współczynnik uczenia stanowi mniej więcej połowę maksymalnego współczynnika uczenia (czyli wartości, dla której algorytm uczenia staje się rozbieżny z rozwiązaniem) (zob. rozdział 4.). Jednym ze sposobów określenia odpowiedniego współczynnika uczenia jest uczenie modelu przez kilkaset iteracji, począwszy od małej wartości tego współczynnika (np. 10^{-5}), i jego stopniowe zwiększanie aż do osiągnięcia bardzo dużej wartości (np. 10). Można tego dokonać dzięki mnożeniu współczynnika uczenia o jakiś stały czynnik w każdej następnej iteracji (np. aby w trakcie 500 iteracji osiągnąć wartość od 10^{-5} do 10, należy pomnożyć współczynnik uczenia przez $\exp(\log(10^6)/500)$). Jeżeli utworzysz wykres zależności funkcji straty od współczynnika uczenia (wyznaczywszy współczynnikowi uczenia skalę logarytmiczną), to na początku funkcja straty będzie malała. Po pewnym czasie jednak wartości współczynnika uczenia staną się zbyt duże, co spowoduje wzrost funkcji straty: optymalna wartość współczynnika uczenia znajduje się tuż przed punktem, w którym funkcja straty zaczyna rosnąć (najczęściej wartość ta jest mniej więcej dziesięciokrotnie mniejsza od punktu zwrotnego). Możesz następnie znów zainicjalizować model i wyczyścić go normalnie przy użyciu optymalnej wartości współczynnika uczenia. W rozdziale 11. omówię inne techniki określania współczynnika uczenia.

Optymalizator

Również całkiem istotny jest dobór optymalizatora lepszego od starego, dobrego algorytmu schodzenia po gradiencie z minigrupami (wraz z dostrajaniem jego hiperparametrów). W rozdziale 11. poznasz kilka zaawansowanych optymalizatorów.

Rozmiar grupy danych

Rozmiar grupy danych może mieć znaczący wpływ na skuteczność modelu i czas uczenia. Największa zaleta korzystania z dużych grup wiąże się z faktem, że są wydajnie przetwarzane przez akceleratory sprzętowe, takie jak karty graficzne (zob. rozdział 19.), zatem algorytm uczący będzie „widział” więcej przykładów na sekundę. Z tego powodu wielu badaczy i adeptów uczenia maszynowego zaleca wyznaczanie maksymalnego rozmiaru grupy, jaki zmieści się w pamięci operacyjnej procesora graficznego. Jest tu jednak pewien haczyk: w praktyce grupy o dużych rozmiarach zmniejszają stabilność procesu uczenia, zwłaszcza na początku, a uzyskany model

może nie radzić sobie z uogólnianiem tak dobrze jak model wytrenowany na małych grupach. W kwietniu 2018 roku Yann LeCun umieścił nawet wpis na Twitterze o treści: „Przyjaciele nie pozwolą Ci korzystać z minigrup większych niż 32 przykłady”, która została zacytowana z artykułu Dominica Mastersa i Carlo Luschiego z 2018 roku (<https://arxiv.org/abs/1804.07612>)²⁴. Autorzy stwierdzili w nim, że preferowane jest korzystanie z małych grup (których rozmiar mieści się w zakresie od 2 do 32), ponieważ prowadzi do lepszych modeli w krótszym czasie uczenia. Można jednak natrafić na przeciwnie opinie: w 2017 roku Elad Hoffer i in. (<https://arxiv.org/abs/1705.08741>)²⁵ oraz Priya Goyal i in. (<https://arxiv.org/abs/1706.02677>)²⁶ udowodnili, że możliwe jest używanie grup o bardzo dużych rozmiarach (aż do 8192 przykładów) za pomocą różnorodnych technik, takich jak rozgrzewanie współczynnika uczenia (tzn. rozpoczęcie treningu z bardzo małym współczynnikiem uczenia, który następnie jest zwiększały) (zob. rozdział 11.). Prowadzi to do znacznego skrócenia czasu uczenia, bez żadnego ubytku w jakości uogólnienia. Warto więc najpierw wypróbować uczenie za pomocą dużej grupy danych przy użyciu techniki rozgrzewania współczynnika uczenia, a jeżeli proces uczenia jest niestabilny lub rezultaty pozostawiają wiele do życzenia, można zmniejszyć rozmiar grupy.

Funkcja aktywacji

Wcześniej w tym rozdziale omówiłem sposób doboru funkcji aktywacji: zasadniczo dobrym wyborem domyślnym dla wszystkich warstw ukrytych jest funkcja ReLU. W przypadku warstwy wyjściowej największe znaczenie ma rodzaj zadania.

Liczba iteracji

W większości przypadków nie trzeba modyfikować liczby przebiegów uczących: w razie potrzeby zaimplementuj wczesne zatrzymywanie.



Optymalna wartość współczynnika uczenia zależy od pozostałych hiperparametrów, zwłaszcza rozmiaru grupy, zatem po zmodyfikowaniu jakiegokolwiek hiperparametru nie zapomnij zaktualizować również współczynnika uczenia.

Więcej porad dotyczących strojenia hiperparametrów sieci neuronowych znajdziesz w znakomitym artykule Leslie Smith z 2018 roku (<https://arxiv.org/abs/1803.09820>)²⁷.

Na tym zakończymy wprowadzenie do sztucznych sieci neuronowych i ich implementacji za pomocą interfejsu Keras. Kilka kolejnych rozdziałów poświęciłem omówieniu technik uczenia bardzo głębokich sieci neuronowych. Nauczysz się również dostosowywać modele za pomocą ogólnego

²⁴ Dominic Masters i Carlo Luschi, *Revisiting Small Batch Training for Deep Neural Networks*, arXiv preprint arXiv:1804.07612 (2018).

²⁵ Elad Hoffer i in., Train Longer, Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks, „Proceedings of the 31st International Conference on Neural Information Processing Systems” (2017), s. 1729 – 1739.

²⁶ Priya Goyal i in., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, arXiv preprint arXiv: 1706.02677 (2017).

²⁷ Leslie N. Smith, A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 — Learning Rate, Batch Size, Momentum, and Weight Decay, arXiv preprint arXiv:1803.09820 (2018).

interfejsu TensorFlow, a także skutecznie wczytywać i wstępnie przetwarzanie dane przy użyciu interfejsu Data. Zapoznasz się też z innymi popularnymi architekturami sieci neuronowych: sieciami splotowymi służącymi do przetwarzania obrazów, sieciami rekurencyjnymi zajmującymi się danymi sekwencyjnymi, autokoderami przeznaczonymi do uczenia reprezentacyjnego oraz generatywnymi sieciami przeciwnymi, które modelują i generują dane²⁸.

Ćwiczenia

1. TensorFlow Playground (<https://playground.tensorflow.org/>) to przydatny symulator sieci neuronowej stworzony przez zespół TensorFlow. W tym ćwiczeniu za pomocą zaledwie kilku kliknięć wyuczyszmy kilka klasyfikatorów binarnych, a także zmodyfikujemy strukturę modelu i jego hiperparametry, aby lepiej zrozumieć mechanizm działania sieci neuronowych oraz wpływ hiperparametrów. Poświęć trochę czasu na następujące zagadnienia:
 - a. Wzorce poznawane przez sieć neuronową. Spróbuj wytrenować domyślną sieć neuronową poprzez kliknięcie przycisku *Run/Pause* (w lewym górnym rogu). Zwróć uwagę, jak szybko wyszukuje optymalne rozwiązanie dla zadania klasyfikacji. Neurony w pierwszej warstwie ukrytej wykryły proste wzorce, natomiast neurony z drugiej warstwy ukrytej nauczyły się łączyć te proste wzorce w bardziej skomplikowane elementy. Zasadniczo im więcej warstw ukrytych umieścimy, tym wzorce będą bardziej skomplikowane.
 - b. Funkcje aktywacji. Spróbuj zastąpić funkcję tanh funkcją ReLU i ponownie wytrenuj sieć. Model wykrywa rozwiązanie jeszcze szybciej, tym razem jednak granice są liniowe. Powodem jest kształt funkcji ReLU.
 - c. Ryzyko natrafienia na minima lokalne. Zmodyfikuj sieć tak, aby w jej strukturze mieściła się tylko jedna warstwa ukryta składająca się z trzech neuronów. Wyucz ją kilkakrotnie (aby wyzerować wagę połączeń, kliknij przycisk *Reset the Network* po lewej stronie przycisku *Run/Pause*). Czas uczenia jest teraz za każdym razem inny, a czasami model zatrzymuje się w minimum lokalnym.
 - d. Działanie zbyt małych sieci neuronowych. Usuń jeden neuron (powinny zostać dwa). Teraz sieć neuronowa nie jest w stanie znaleźć dobrego rozwiązania, nawet jeżeli będziemy wielokrotnie ją uruchamiać. Model ma za mało parametrów i ciągle pozostaje niedotrenowany.
 - e. Działanie odpowiednio dużych sieci neuronowych. Dodaj w warstwie ukrytej sześć neuronów (powinno być ich teraz osiem) i wytrenuj sieć kilka razy. Teraz jest ona stabilnie szybka i nie pozostaje już w minimach lokalnych. To jest ważny wniosek w teorii sieci neuronowych: duże sieci neuronowe prawie nigdy nie pozostają w minimach lokalnych, a nawet jeśli, to takie minima lokalne są niemal tak samo dobre jak optimum globalne. Mimo to mogą one utknąć na długich wypłaszczeniach przez dłuższy czas.
 - f. Ryzyko zanikania gradientów w sieciach głębokich. Wybierz spiralny zestaw danych (prawy dolny w panelu *DATA*) i zmodyfikuj strukturę sieci tak, aby zawierała cztery warstwy ukryte z ośmioma neuronami w każdej z nich. Proces uczenia trwa teraz znacznie dłużej i często zatrzymuje się na wypłaszczeniach przez długi czas. Zwróć

²⁸ Kilka dodatkowych sieci ANN zostało opisanych w dodatku E.

również uwagę, że neurony w najwyższej warstwie (po prawej) ewoluują szybciej od neuronów z warstw najniższych (po lewej). Problem ten, zwany problemem zanikających gradientów, można zredukować poprzez lepsze inicjalizowanie wag i inne techniki, optymalizatory (takie jak AdaGrad czy Adam) lub normalizację wsadową (zob. rozdział 11.).

- g.** Idź dalej. Poświęć mniej więcej godzinę na modyfikowanie innych parametrów i zrozum ich wpływ po to, aby lepiej pojąć sieci neuronowe.
- 2.** Przy użyciu oryginalnych neuronów sztucznych (takich jak na rysunku 10.3) stwórz sztuczną sieć neuronową obliczającą operację $A \oplus B$ (gdzie \oplus symbolizuje operację XOR). Podpowiedź: $A \oplus B = (A \square B \square (A \square B))$.
- 3.** Dlaczego lepiej korzystać z klasyfikatora regresji logistycznej niż z klasycznego perceptronu (tzn. pojedynczej warstwy jednostek logicznych z funkcją progową wytrenowanych za pomocą algorytmu uczenia perceptronu)? W jaki sposób można zmodyfikować perceptron, aby stał się równoważny klasyfikatorowi regresji logistycznej?
- 4.** Dlaczego podczas uczenia pierwszych perceptronów wielowarstwowych kluczowym składnikiem była logistyczna funkcja aktywacji?
- 5.** Wymień trzy popularne funkcje aktywacji. Narysuj ich wykresy.
- 6.** Powiedzmy, że dysponujesz perceptronem wielowarstwowym składającym się z jednej warstwy wejściowej zawierającej 10 neuronów przechodniczych połączonych z warstwą ukrytą utworzoną z 50 neuronów, które łączą się trójneuronową warstwą wyjściową. Wszystkie neurony wykorzystują funkcję aktywacji ReLU.

 - a.** Jaki jest rozmiar macierzy wejściowej X ?
 - b.** Jaki jest rozmiar wektora wag W_u i wektora obciążen b_u warstwy ukrytej?
 - c.** Jaki jest rozmiar wektora wag W_{wy} i wektora obciążen b_{wy} warstwy wyjściowej?
 - d.** Jaki jest rozmiar macierzy wyjściowej Y ?
 - e.** Napisz równanie obliczające macierz wyjściową Y jako funkcję X , W_u , b_u , W_{wy} i b_{wy} .
- 7.** Ile neuronów należy umieścić w warstwie wyjściowej, aby odróżniać normalne wiadomości e-mail od spamu? Jaką funkcję aktywacji należy zdefiniować w warstwie wyjściowej? Gdybyśmy chcieli klasyfikować zestaw danych MNIST, ile neuronów należałoby dodać lub odjąć w warstwie wyjściowej oraz z jakiej funkcji aktywacji skorzystać? Jak wyglądałaby sytuacja w przypadku przeprowadzania cen domów za pomocą zestawu Housing opisanego w rozdziale 2.?
- 8.** Czym jest propagacja wsteczna i jaki jest mechanizm jej działania? Jaka jest różnica między propagacją wsteczną a odwrotnym różniczkowaniem automatycznym?
- 9.** Czy potrafisz wymienić wszystkie hiperparametry modyfikowane w podstawowym perceptronie wielowarstwowym? W jaki sposób je zmodyfikujesz, aby wyeliminować przetrenowanie modelu?

- 10.** Wytrenuj głęboki perceptron wielowarstwowy na zestawie danych MNIST (możesz go wczytać za pomocą funkcji keras.datasets.mnist.load_data()). Spróbuj uzyskać precyzyję przekraczającą 98%. Postaraj się wyszukać optymalny współczynnik uczenia za pomocą techniki opisanej w niniejszym rozdziale (tzn. wykładnicze zwiększenie wartości współczynnika uczenia w każdej iteracji, utworzenie wykresu funkcji straty i znalezienie punktu, w którym wartość tej funkcji zaczyna rosnąć). Spróbuj dodać inne funkcje, takie jak zapisywanie punktów kontrolnych, wczesne zatrzymywanie czy generowanie wykresów krzywych uczenia przy użyciu narzędzia TensorBoard.

Rozwiązań tych ćwiczeń znajdziesz w dodatku A.

Uczenie głębokich sieci neuronowych

W rozdziale 10. omówiłem pojęcie sztucznej sieci neuronowej i wytrenowaliśmy nasze pierwsze głębokie sieci neuronowe. Nie były one jednak zbyt głębokie, gdyż zawierały zaledwie kilka warstw ukrytych. A gdybyśmy na przykład musieli zajmować się bardzo złożonym problemem, np. wykrywaniem setek typów obiektów na zdjęciach o wysokiej rozdzielcości? Wymagałoby to wyuczenia znacznie głębszej sieci neuronowej, być może składającej się z (powiedzmy) 10 warstw, z których każda zawierałaby setki neuronów i setki tysięcy połączeń. To nie byłaby bułka z masłem. Oto kilka problemów, na jakie możesz natrafić:

- Musisz poradzić sobie z niełatwym problemem **zanikających gradientów** lub ściśle powiązany z nim problemem **eksplodujących gradientów**. Występują one wtedy, gdy gradienty stopniowo maleją lub rosną na etapie propagacji wstecznej podczas uczenia sieci neuronowej. Obydwia te problemy znacznie utrudniają uczenie niższych warstw sieci.
- Możesz mieć niewystarczającą ilość danych uczących na tak rozbudowaną sieć lub ich oznaczanie może być zbyt kosztowne.
- Uczenie tak rozbudowanej sieci może być bardzo czasochłonne.
- Model zawierający miliony parametrów jest w znacznym stopniu narażony na przetrenowanie, zwłaszcza w przypadku niewystarczającej liczby przykładów uczących lub ich zbytniego za-szumienia.

W tym rozdziale przyjrzymy się uważnie każdemu z wymienionych zagadnień i zademonstrujemy sposoby rozwiązymania tych problemów. Zaczniemy od wyjaśnienia problemów zanikających/eksplodujących gradientów i dowiemy się, jak możemy im zapobiec. Następnie przyjrzymy się uczeniu transferowemu i nienadzorowanemu uczeniu wstępнемu, co często pomaga w realizowaniu skomplikowanych zadań nawet w przypadku dysponowania niewielką ilością oznakowanych danych. W dalszej części rozdziału przeanalizujemy różne optymalizatory znacznie przyspieszające uczenie dużych modeli. Na koniec omówię kilka popularnych technik regularyzacji rozbudowanych sieci neuronowych.

Dzięki tym narzędziom będziesz w stanie trenować nawet bardzo głębokie sieci neuronowe. Witaj w świecie uczenia głębokiego!

Problemy zanikających/eksplodujących gradientów

Wiemy już z rozdziału 10., że algorytm propagacji wstecznej przebiega od warstwy wyjściowej do wejściowej i rozprowadza po drodze wartość gradientu błędu. Po wyliczeniu gradientu funkcji kosztu w odniesieniu do każdego parametru sieci algorytm ten wykorzystuje obliczone gradienty do zaktualizowania każdego parametru za pomocą gradientu prostego.

Niestety wartości gradientów często maleją wraz z przebiegiem algorytmu do niższych warstw sieci. Wskutek tego aktualizacja gradientu prostego praktycznie nie powoduje żadnych zmian w wagach połączeń tych warstw, a model nie osiąga optymalnej zbieżności. Jest to tak zwany problem **zanikających gradientów** (ang. *vanishing gradients*). W niektórych sytuacjach możemy natrafić na przeciwnie zjawisko: gradienty stale rosną, przez co w wielu warstwach wagi są aktualizowane w szalonym tempie, a algorytm staje się rozbieżny. Mamy w tym przypadku do czynienia z problemem **eksplodujących gradientów** (ang. *exploding gradients*), który najczęściej występuje w rekurencyjnych sieciach neuronowych (zob. rozdział 15.). Ogólnie mówiąc, głębokie sieci neuronowe cierpią na syndrom niestabilnych gradientów; poszczególne warstwy mogą uczyć się z diametralnie różnymi szybkościami.

Chociaż to niefortunne zjawisko było empirycznie znane już od dawna i stanowiło jeden z powodów, przez które głębokie sieci neuronowe zostały w większości porzucone na początku XXI wieku. Nie znano przyczyn niestabilności gradientów podczas uczenia głębokich sieci neuronowych, a pewne przesłanki pojawiły się dopiero w artykule z 2010 roku (<http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>), którego autorami są Xavier Glorot i Yoshua Bengio¹. Wykryli oni kilku „podejrzanych” o taki stan rzeczy, między innymi kombinację popularnej logistycznej, sigmoidalnej funkcji aktywacji, a także najczęściej używaną w tamtym czasie technikę inicjalizacji wag (mianowicie losową inicjalizację przy użyciu rozkładu normalnego o średniej równej 0 i odchyleniu standardowym równym 1). Krótko mówiąc, autorzy udowodnili, że z powodu tej funkcji aktywacji i strategii inicjalizacji wag wariancja na wyjściach każdej warstwy jest znacznie większa niż wariancja na ich wejściach. Podczas przebiegu naprzód wariancja wzrasta wraz z każdą następną warstwą aż do momentu nasycenia funkcji aktywacji w górnych warstwach. Sprawę pogarsza fakt, że średnia funkcji logistycznej ma wartość 0,5, a nie 0 (funkcja tangensa hiperbolicznego ma średnią o wartości 0 i w sieciach neuronowych sprawuje się nieco lepiej od funkcji logistycznej).

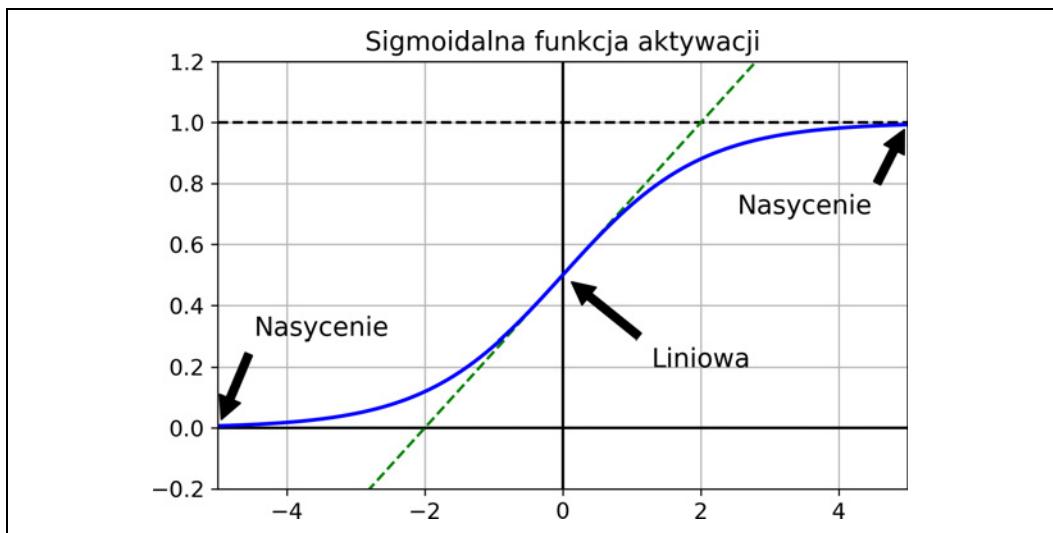
Jeśli przyjrzymy się logistycznej funkcji aktywacji (rysunek 11.1), zauważymy, że gdy dane wejściowe przyjmują duże wartości (ujemne lub dodatnie), funkcja ta ulega nasyceniu w punktach, odpowiednio, 0 i 1, zaś jej pochodna zbliża się znacznie do 0. Dlatego w momencie rozpoczęcia propagacji wstecznej praktycznie nie ma gradientu, który można byłoby rozprowadzić po sieci, natomiast ta reszta gradientu, jaką mamy do dyspozycji, ulega dalszemu „rozcieńczaniu” podczas schodzeniu do początkowych warstw, dlatego właściwie żaden gradient nie dociera do tych niższych warstw.

Inicjalizacje wag Glorota i He

We wspomnianym już artykule Glorot i Bengio zaproponowali mechanizm w dużej mierze rozwiązuający problem niestabilnych gradientów. Musimy w prawidłowy sposób określić przepływ sygnału

¹ Xavier Glorot i Yoshua Bengio, *Understanding the Difficulty of Training Deep Feedforward Neural Networks*, „Proceedings of the 13th International Conference on Artificial Intelligence and Statistics” (2010), s. 249 – 256.

w obydwu kierunkach: do przodu podczas wyliczania prognoz oraz w przeciwnym kierunku na etapie wstępnej propagacji gradientów. Nie chcemy, żeby sygnał zaniknął ani żeby eksplodował i spowodował nasycenie funkcji. Autorzy stwierdzają, że w celu właściwego sygnalizowania kierunku przepływu informacji wariancja wyjść w danej warstwie musi być równa wariancji jej wejść², a do tego gradienty muszą mieć taką samą wariancję przed przejściem i po przejściu przez warstwę w odwrotnym kierunku (matematyczny opis tej techniki znajdziesz w samym artykule). W rzeczywistości nie da się jednocześnie spełnić obydwu warunków, jeśli dana warstwa zawiera różną liczbę wejść i wyjść (jest to tzw. **obciążalność wejściowa** — ang. *fan-in* — i **obciążalność wyjściowa** — ang. *fan-out*), ale Glorot i Bengio zaproponowali dobry kompromis, który w istocie sprawdza się całkiem skutecznie: wagi połączeń w każdej warstwie muszą być losowo zainicjalizowane, co zostało pokazane w równaniu 11.1, gdzie $obc_{sr} = (obc_{we} + obc_{wy})/2$. Taka strategia inicjalizowania wag często jest nazywana **inicjalizacją Xavierem** (od imienia autora; ang. *Xavier initialization*) lub czasami **inicjalizacją Glorota**.



Rysunek 11.1. Nasycenie logistycznej funkcji aktywacji

Równanie 11.1. Inicjalizacja Xavier'a (podczas stosowania logistycznej funkcji aktywacji)

$$\text{Rozkład normalny ze średnią o wartości } 0 \text{ i odchyleniem standardowym } \sigma^2 = \frac{1}{obc_{sr}}$$

$$\text{Lub rozkład jednorodny pomiędzy } -r \text{ i } r, \text{ gdzie } r = \sqrt{\frac{3}{obc_{sr}}}$$

² Możemy skorzystać tu z następującej analogii: jeżeli ustawisz wartość pokrętła wzmacniacza mikrofonu blisko zera, to słuchacze nie będą słyszeć Twojego głosu, ale jeżeli ustawisz wartość zbyt blisko maksimum, Twój głos stanie się „nasycony” i nikt nie zrozumie, co mówisz. Teraz wyobraź sobie cały szereg takich wzmacniaczy: należy prawidłowo ustawić wzmacnienie na każdym z nich, aby Twój głos docierał do słuchaczy głośno i wyraźnie. Dźwięk Twojego głosu musi mieć na wyjściu każdego wzmacniacza taką samą amplitudę, jak na jego wejściu.

Jeżeli w równaniu 11.1 zastąpimy wartość obc_{sr} wartością obc_{we} , otrzymamy strategię inicjalizowania zaproponowaną przez Yanna LeCuna w latach 90. ubiegłego wieku i nazwaną od jego nazwiska **inicjalizacją LeCuna** (ang. *LeCun initialization*). Genevieve Orr i Klaus-Robert Müller zalecali jej stosowanie w swojej książce z 1998 roku pod tytułem *Neural Networks: Tricks of the Trade* (Springer). Inicjalizacja LeCuna jest równoważna inicjalizacji Xaviera, gdy $obc_{we} = obc_{wy}$. Musiała upływać ponad dekada, aby badacze zrozumieli znaczenie tej sztuczki. Inicjalizacja Glorota może istotnie przyspieszyć proces uczenia i stanowi jedną z podwalin sukcesu uczenia głębokiego.

W niektórych artykułach³ zaproponowano podobne strategie dla innych funkcji aktywacji. Strategie te różnią się wyłącznie skalą wariancji i użyciem obc_{we} lub obc_{sr} , co zaprezentowałem w tabeli 11.1 (w przypadku rozkładu jednostajnego wystarczy obliczyć $r = \sqrt{3\sigma^2}$). Strategia inicjalizacji (<https://arxiv.org/abs/1502.01852>) dla funkcji aktywacji ReLU (oraz jej odmian, np. aktywacji ELU, do której powróćmy wkrótce) jest czasami nazywana **inicjalizacją He** (od nazwiska twórcy; ang. *He initialization*). W dalszej części rozdziału omówię także funkcję aktywacji SELU. Należy ją stosować wraz z inicjalizacją LeCuna (jak się przekonamy, najlepiej o rozkładzie normalnym).

Tabela 11.1. Parametry inicjujące dla różnych funkcji aktywacji

Inicjalizacja	Funkcja aktywacji	σ^2 (Rozkład normalny)
Glorota	brak, tanh, logistyczna, softmax	$1/obc_{sr}$
He	ReLU i jej odmiany	$2/obc_{we}$
LeCuna	SELU	$1/obc_{we}$

Domyślnie interfejs Keras wykorzystuje inicjalizację Glorota z rozkładem jednostajnym. Możesz to zmienić na inicjalizację He w czasie tworzenia warstwy poprzez wyznaczenie parametrów `kernel_initializer="he_uniform"` lub `kernel_initializer="he_normal"`:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Jeżeli zależy Ci na inicjalizacji He o rozkładzie jednostajnym, ale bazującej na obc_{sr} , a nie obc_{we} , możesz wprowadzić inicjalizator `VarianceScaling`:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                    distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

Nienasycające funkcje aktywacji

Jednym ze spostrzeżeń, jakich dokonali Glorot i Bengio w swoim artykule z 2010 roku, jest fakt, że problemy zanikających/eksplodujących gradientów wynikają w pewnym stopniu z niewłaściwego doboru funkcji aktywacji. Wcześniej uważano, że skoro Matka Natura wybrała sigmoidalnopodobne funkcje aktywacji do przetwarzania sygnałów w neuronach biologicznych, to muszą być one najlepsze. Okazuje się jednak, że w przypadku głębokich sieci neuronowych inne funkcje aktywacji sprawdzają

³ Na przykład Kaiming He i in., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, „Proceedings of the 2015 IEEE International Conference on Computer Vision” (2015), s. 1026 – 1034.

się znacznie lepiej, w szczególności zaś funkcja ReLU, głównie z tej przyczyny, że nie ulega ona nasyceniu dla wartości dodatnich (a także z powodu jej szybkości obliczeniowej).

Niestety funkcja ReLU nie jest idealna. Znany jest problem nazywany **śmiercią ReLU** (ang. *dying ReLUs*): w czasie uczenia niektóre neurony trwale „giną”, co oznacza, że jedynym przesyłanym przez nie sygnałem jest 0. W pewnych sytuacjach okazuje się, że połowa sieci przestała być aktywna, zwłaszcza jeżeli opisuje ją duży współczynnik uczenia. Jeżeli w trakcie uczenia wagi neuronu zostaną zaktualizowane tak, że suma ważona sygnałów wejściowych dla tego neuronu przyjmie wartość ujemną, to zacznie on wysyłać sygnał 0. Jeżeli zdarzy się taka sytuacja, martwy neuron będzie wysyłał same zera, a algorytm gradientu prostego przestanie na niego wpływać, ponieważ gradient funkcji ReLU wynosi 0 przy ujemnych wartościach na wejściu⁴.

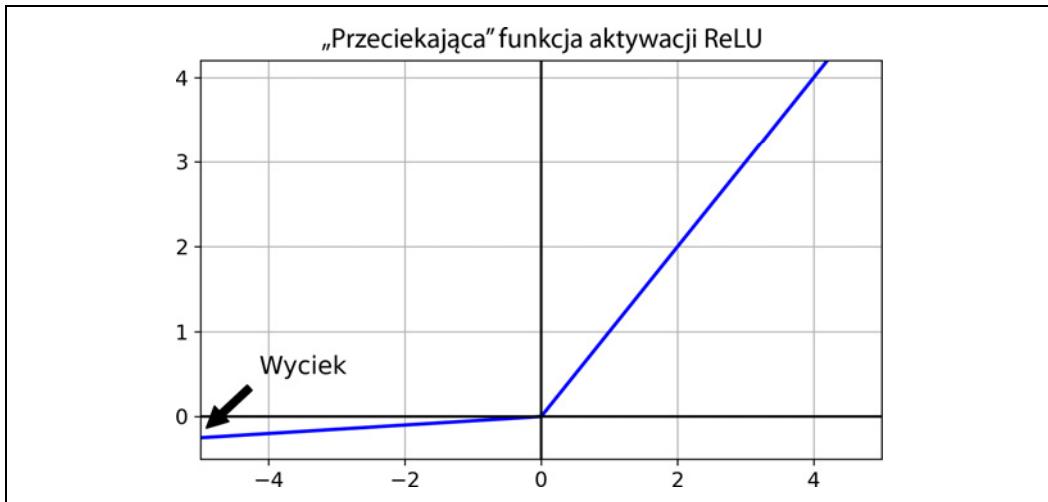
W celu uniknięcia tego problemu możesz zastosować wariant funkcji ReLU nazywany **przeciekającą funkcją ReLU** (ang. *leaky ReLU*). Definiujemy ją jako PrzeciekającaReLU $_{\alpha}(z) = \max(\alpha z, z)$ (rysunek 11.2). Hiperparametr α określa stopień „przeciekania” funkcji: wyznacza on nachylenie funkcji dla $z < 0$ i zazwyczaj przyjmuje wartość 0,01. Dzięki takiemu niewielkiemu nachyleniu „przeciekające” neurony nigdy nie „giną”; mogą zapaść w długą „śpiączkę”, ale zawsze istnieje szansa, że się „przebudzą”. W artykule z 2015 r. (<https://arxiv.org/pdf/1505.00853.pdf>)⁵ porównano kilka wariantów funkcji aktywacji ReLU i jednym z wniosków jest stwierdzenie, że „przeciekające” wersje za każdym razem okazywały się lepsze od standardowej funkcji ReLU. Rzeczywiście, wydaje się, że wyznaczenie wartości $\alpha = 0,2$ (duży wyciek) zapewnia większą wydajność niż wartość $\alpha = 0,01$ (mały wyciek). Została również oceniona **losowa, przeciekająca funkcja ReLU** (ang. *randomized leaky ReLU* — RReLU), w której wartość hiperparametru α jest podczas nauki losowo dobierana w określonym zakresie, a w trakcie testowania zostaje wyznaczona jej uśrednia wartość. Takie rozwiązanie jest również skuteczne i wydaje się, że dobrze pełni rolę regularyzatora (zmniejsza ryzyko przetrenowania wobec zbioru uczącego). Na koniec została również przeanalizowana **parametryczna, przeciekająca funkcja ReLU** (ang. *parametric leaky ReLU* — PReLU), w której parametr α również „uczy się” wraz z całym modelem (przystaje być hiperparametrem i, podobnie jak pozostałe parametry, może być modyfikowany w ramach propagacji wstecznej). Okazuje się, że ta wersja funkcji ReLU znacznie przewyższa wydajnością klasyczną odmianę funkcji w przypadku dużych zbiorów obrazów, ale przy mniejszych zestawach danych pojawia się ryzyko przetrenowania modelu.

Warto również zapoznać się z publikacją Djorka-Arnégo Cleverta i in. z 2015 roku (<https://arxiv.org/abs/1511.07289>)⁶, w której zaproponowano nowy rodzaj funkcji aktywacji nazwanej **jednostką wykładniczo-liniową** (ang. *exponential linear unit* — ELU), która w przeprowadzonych doświadczeniach okazała się lepsza od wszystkich odmian funkcji ReLU: czas nauki był krótszy, a sama sieć neuronowa sprawowała się lepiej wobec zbioru testowego. Jej wykres został ukazany na rysunku 11.3, natomiast równanie 11.2 zawiera definicję tej funkcji.

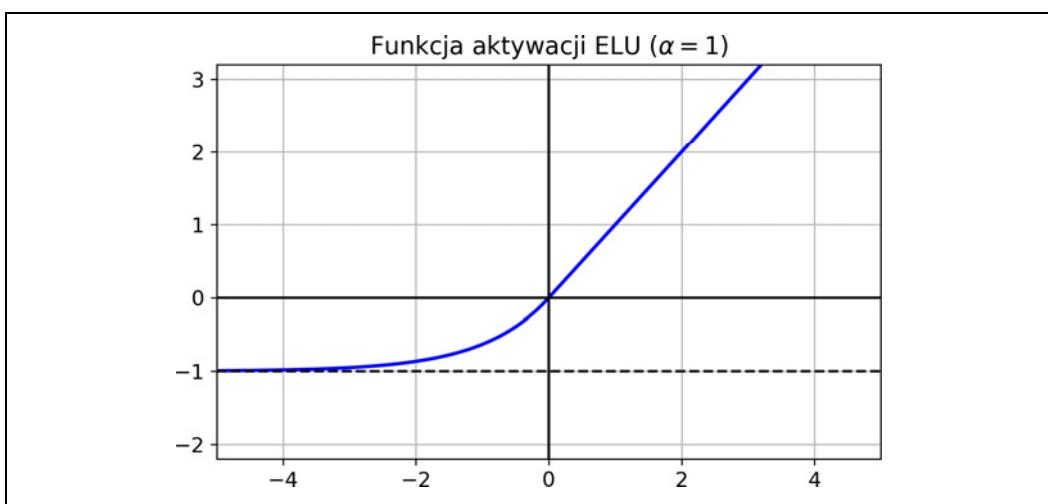
⁴ Jeżeli martwy neuron nie stanowi elementu pierwszej warstwy ukrytej, to czasami zdarzają się przypadki jego „zmartwychwstania”: algorytm gradientu prostego może rzeczywiście tak zmodyfikować niższe warstwy, że suma ważona sygnału na wejściach martwego neuronu znowu będzie dodatnia.

⁵ Bing Xu i in., *Empirical Evaluation of Rectified Activations in Convolutional Network*, arXiv preprint arXiv:1505.00853 (2015).

⁶ Djork-Arne Clevert i in., *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*, „Proceedings of the International Conference on Learning Representations” (2016).



Rysunek 11.2. Przeciekająca funkcja ReLU: przypomina klasyczną funkcję ReLU, ale zawiera niewielkie nachylenie dla wartości ujemnych



Rysunek 11.3. Funkcja aktywacji ELU

Równanie 11.2. Funkcja aktywacji ELU

$$ELU_{\alpha}(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{jeśli } z < 0 \\ z & \text{jeśli } z \geq 0 \end{cases}$$

Przypomina ona zwykłą funkcję ReLU, możemy jednak dostrzec kilka zasadniczych różnic:

- Po pierwsze, przyjmuje ona ujemne wartości dla $z < 0$, dzięki czemu średnia wartość na wyjściu jednostki jest bardziej zbliżona do zera. W ten sposób rozwiążymy omówiony wcześniej problem zanikających gradientów. Hiperparametr α definiuje wartość, do jakiej funkcja ma się zbliżać

dla dużych ujemnych wartości z . Zazwyczaj wyznaczamy jego wartość 1, ale w razie potrzeby możemy go zmodyfikować tak jak wszystkie hiperparametry.

- Po drugie, dla $z < 0$ ma ona niezerowy gradient, co stanowi rozwiązańe problemu „umierających” neuronów.
- Po trzecie, jeżeli wartość α jest równa 1, to funkcja jest gładka w każdym punkcie, również w $z = 0$, co pozwala przyspieszyć metodę gradientu prostego, gdyż nie skacze ona tak bardzo na boki w okolicy punktu $z = 0$.

Główna wada funkcji ELU polega na tym, że jest ona wolniej wyliczana od standardowej funkcji ReLU i jej odmian (z powodu wykorzystania funkcji wykładniczej). W trakcie uczenia jest to rekompensowane lepszym współczynnikiem zbieżności, jednakże w trakcie testowania sieć ELU będzie wolniejsza od sieci ReLU.

Z kolei w artykule z 2017 roku (<https://arxiv.org/abs/1706.02515>)⁷ Günter Klambauer i in. zaproponowali **skalowaną funkcję aktywacji ELU** (ang. *Scaled ELU* — SELU) — jak wskazuje nazwa, mamy tu do czynienia ze skalowaną wersją funkcji ELU. Autorzy udowodnili, że jeśli stworzyć sieć neuronową zbudowaną wyłącznie ze stosu warstw gęstych oraz jeśli we wszystkich warstwach ukrytych będzie zaimplementowana funkcja aktywacji ELU, to sieć ta będzie podlegać **samonormalizacji** (ang. *self-normalize*): sygnał wyjściowy w każdej warstwie będzie zachowywał średnią 0 i odchylenie standardowe równe 1 w trakcie uczenia, co rozwiązuje problem zanikających/eksplodujących gradientów. W konsekwencji funkcja SELU często osiąga znacznie lepsze rezultaty od pozostałych funkcji aktywacji w tego typu sieciach neuronowych (zwłaszcza głębokich). Należy jednak spełnić kilka warunków, aby umożliwić samonormalizację (dowód matematyczny znajdziesz we wspomnianym artykule):

- Cechy wejściowe muszą być znormalizowane (średnia równa 0 i odchylenie standardowe równe 1).
- Wagi każdej warstwy ukrytej muszą być inicjalizowane za pomocą inicjalizacji LeCuna z rozkładem normalnym. W interfejsie Keras oznacza to wyznaczenie parametru `kernel_initializer="lecun_normal"`.
- Struktura sieci musi być sekwencyjna. Niestety jeżeli spróbujesz użyć funkcji SELU w sieciach niesekwencyjnych, takich jak sieci rekurencyjne (zob. rozdział 15.) czy sieci zawierające **połączenia pomijające** (tzn. połączenia omijające część warstw, na przykład w sieciach Wide & Deep), to samonormalizacja może nie nastąpić, przez co funkcja SELU może nie osiągnąć lepszych rezultatów w stosunku do innych funkcji aktywacji.
- W artykule została opisana samonormalizacja wyłącznie w przypadku warstw gęstych, ale niektórzy badacze zauważyli, że funkcja aktywacji SELU może również zwiększyć skuteczność splotowych sieci neuronowych (zob. rozdział 14.).

⁷ Günter Klambauer i in., *Self-Normalizing Neural Networks*, „Proceedings of the 31st International Conference on Neural Information Processing Systems” (2017), s. 972 – 981.



Zatem jaką funkcję aktywacji należy zastosować w warstwach ukrytych Twoich sieci głębokich? Liczba przebiegów może się różnić w poszczególnych sieciach, ale generalnie kolejność wygląda następująco: funkcja SELU > ELU > przeciekająca ReLU (wraz z odmianami) > ReLU > tangensa hiperbolicznego > logistyczna. Jeżeli dana sieć nie podlega samonormalizacji, to funkcja ELU może dawać lepsze rezultaty niż funkcja SELU (gdyż funkcja SELU nie jest gładka w punkcie $z = 0$). Jeśli bardziej zależy Ci na szybkości, możesz wybrać przeciekającą funkcję ReLU. Jeżeli nie masz ochoty dostrajać kolejnego parametru, możesz użyć domyślnych wartości α zdefiniowanych w interfejsie Keras (np. 0,3 dla przeciekającej funkcji ReLU). Z kolei jeśli masz nadmiar czasu i mocy obliczeniowej, możesz ocenić skuteczność innych funkcji aktywacji za pomocą sprawdzianu krzyżowego, zwłaszcza funkcji RReLU, jeśli Twój model ulega przetrenowaniu, lub PReLU dla dużych zbiorów uczących. Pamiętaj, że (obecnie) ReLU jest najczęściej wykorzystywana funkcją aktywacji, dlatego wiele bibliotek i akceleratorów sprzętowych zostało zoptymalizowanych pod nią, a zatem jeśli szybkość stanowi priorytet, to najlepszym wyborem może być funkcja ReLU.

Aby skorzystać z przeciekającej funkcji ReLU, utwórz warstwę LeakyReLU i umieść ją tuż za warstwą, która ma tej funkcji używać:

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

W przypadku funkcji PReLU zastąp parametr LeakyReLU(alpha=0.2) funkcją PReLU(). Obecnie nie istnieje oficjalna implementacja funkcji RReLU w interfejsie Keras, ale można dość łatwo stworzyć własną implementację (nauczysz się tego z ćwiczeń umieszczonego na końcu rozdziału 12.).

Funkcję aktywacji SELU uzyskasz poprzez wyznaczenie parametrów activation="selu" i kernel_initializer="lecun_normal" podczas tworzenia warstwy:

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Normalizacja wsadowa

Mimo że stosowanie inicjalizacji He wraz z funkcją ELU (lub dowolną odmianą funkcji ReLU) może znacznie zredukować problem zanikających/eksplodujących gradientów na początku procesu uczenia, nie gwarantuje ochrony przed nim po upływie czasu.

W publikacji z 2015 roku (<https://arxiv.org/abs/1502.03167>)⁸ Sergey Ioffe i Christian Szegedy zaproponowali technikę zwany **normalizacją wsadową** (ang. *batch normalization* — BN) jako rozwiązanie tego problemu. Technika ta polega na wstawianiu operacji do modelu tuż przed funkcją aktywacji w każdej warstwie. W operacji tej dane wejściowe zostają wyśrodkowane i znormalizowane, a jej wynik podlega przeskalowaniu i przesunięciu za pomocą dwóch nowych wektorów parametrów przypadają-

⁸ Sergey Ioffe i Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, „Proceedings of the 32nd International Conference on Machine Learning” (2015), s. 448 – 456.

jących na każdą warstwę: jeden odpowiada za skalowanie, drugi za przesunięcie. Innymi słowy, operacja ta pozwala modelowi określić optymalną skalę i średnią danych wejściowych dla każdej warstwy. W wielu przypadkach po umieszczeniu warstwy BN na samym początku sieci neuronowej nie musisz standaryzować zestawu danych uczących (np. za pomocą obiektu `StandardScaler`) — warstwa normalizacji wsadowej wykona tę operację za Ciebie (mniej więcej, gdyż za każdym razem wykorzystuje tylko jedną grupę, a do tego może przeskalać i przesunąć każdą cechę wejściową).

W celu wyśrodkowania i znormalizowania danych wejściowych algorytm musi oszacować średnią i odchylenie standardowe każdego sygnału wejściowego. Dokonuje tego poprzez wyliczenie tych wartości dla bieżącej minigrupy (stąd nazwa „normalizacja wsadowa”). Cały proces został zaprezentowany w równaniu 11.3.

Równanie 11.3. Algorytm normalizacji wsadowej

$$\begin{aligned} 1. \quad \boldsymbol{\mu}_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\ 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2 \\ 3. \quad \hat{\mathbf{x}}^{(i)} &= \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ 4. \quad \mathbf{z}^{(i)} &= \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta \end{aligned}$$

W tym algorytmie:

- $\boldsymbol{\mu}_B$ jest wektorem średnich sygnałów wejściowych, wyliczonych dla całej minigrupy B (zawiera po jednej średniej na każde wejście).
- σ_B jest wektorem odchyлеń standardowych sygnałów wejściowych, również wyliczonych dla całej minigrupy B (zawiera po jednym odchyleniu standardowym na każde wejście).
- m_B jest liczbą przykładów tworzących minigrupę.
- $\hat{\mathbf{x}}^{(i)}$ jest wektorem wyśrodkowanych i znormalizowanych wejść dla i -tego przykładu.
- γ jest wektorem parametrów skalowania w danej warstwie (zawiera po jednym parametrze skali na każde wejście).
- \otimes symbolizuje operację mnożenia po elementach (każdy sygnał wejściowy jest mnożony przez odpowiadający mu wynikowy parametr skali).
- β jest wektorem parametrów przesunięcia w danej warstwie (zawiera po jednej wartości przesunięcia na każde wejście). Każde wejście zostaje przesunięte o odpowiadający mu parametr przesunięcia.
- ϵ jest niewielką liczbą służącą do uniknięcia operacji dzielenia przez 0 (zazwyczaj ma wartość 10^{-5}). Jest to tzw. **człon wygładzający** (ang. *smoothing term*).
- $\mathbf{z}^{(i)}$ jest wynikiem operacji BN — przeskalowaną i przesuniętą wersją danych wejściowych.

Zatem podczas uczenia algorytm BN standaryzuje dane wejściowe, a następnie je przeskalał i przesuwa. Dobrze! A jak sytuacja wygląda w czasie testowania? Tutaj już nie jest tak łatwo. W istocie być może chcemy czasami uzyskiwać prognozy dla pojedynczych przykładów, a nie całych ich grup — w takiej sytuacji nie ma możliwości obliczenia średniej i odchylenia standardowego każdego wejścia. Co więcej, nawet w przypadku dysponowania grupą przykładów może być ona zbyt mała lub przykłady mogą być niezależne i o jednakowym rozkładzie, więc obliczanie statystyk takiej grupy danych może się okazać nierzetelne. Jednym z rozwiązań jest zaczekać do końca treningu, a następnie przepuścić cały zestaw uczący przez sieć neuronową i obliczyć średnią oraz odchylenie standarde každego wejścia w warstwie BN. Takie „ostateczne” średnie i odchylenia standardowe wejść można byłoby wykorzystać zamiast średnich i odchylen standardowych grup podczas uzyskiwania prognoz. Jednak większość implementacji normalizacji wsadowej oszacowuje te statystyki końcowe podczas uczenia za pomocą średniej kroczącej ze średnich i odchylen standardowych wejść tej warstwy. Jest to automatyczny mechanizm działania interfejsu Keras po wstawieniu warstwy BatchNormalization. Podsumowując: w każdej warstwie normalizacji wsadowej poznawane są cztery wektory parametrów: γ (wyjściowy wektor skali) i β (wyjściowy wektor przesunięcia) poznawane są w wyniku standardowej propagacji wstecznej, natomiast μ (końcowy wektor średniej wejść) i σ (końcowy wektor odchylenia standardowego wejść) są szacowane przy użyciu wykładniczej średniej kroczącej. Zwróć uwagę, że μ i σ są szacowane w trakcie uczenia, ale zostają użyte wyłącznie po jego zakończeniu (zastępują wejściowe średnie i odchylenia standardowe grupy w równaniu 11.3).

Ioffe i Szegedy udowodnili, że ta technika bardzo usprawniła wszystkie głębokie sieci neuronowe, na których prowadzili eksperymenty, i znacząco udoskonaliła zadanie klasyfikacyjne ImageNet (jest to olbrzymi zestaw obrazów podzielonych na wiele klas; często służy do oceniania systemów widzenia komputerowego). Problem zanikających gradientów został mocno ograniczony — do takiego stopnia, że można było zacząć używać nasycających funkcji aktywacji, takich jak tangens hiperboliczny, a nawet logistyczna funkcja aktywacji. Sieci okazywały się również o wiele mniej wrażliwe na inicjalizację wag. Można było wprowadzać znaczne większe wartości współczynnika uczenia, co przyspieszało w olbrzymim stopniu cały proces uczenia. Autorzy zauważali zwłaszcza, że:

Normalizacja wsadowa po dołączeniu do nowoczesnego modelu klasyfikacji obrazów osiąga tę samą dokładność przy 14 razy mniejszej liczbie przebiegach uczących oraz uzyskuje o wiele lepsze wyniki od oryginalnego modelu. [...] Po zastosowaniu zespołu normalizowanych wsadowo sieci osiągamy rezultaty jeszcze lepsze od najlepszych wyników klasyfikacji ImageNet: uzyskujemy wartość 4,9% błędu walidacji typu top-5 (a także 4,8% błędu testowego), przez co uzyskujemy wyższą dokładność od ludzkich analityków.

Jakby tego było mało, normalizacja wsadowa sprawdza się również jako regularyzator i zmniejsza zapotrzebowanie na stosowanie innych technik regularyzacji (np. porzucania, opisanego w dalszej części rozdziału).

Normalizacja wsadowa dodaje jednak pewien poziom skomplikowania do modelu (z drugiej strony, niweluje konieczność normalizowania danych, o czym już wiemy). Co więcej, metoda ta spowalnia działanie modelu: wyliczanie prognoz zajmuje sieci neuronowej więcej czasu z powodu dodatkowych obliczeń wykonywanych w każdej warstwie. Na szczęście istnieje możliwość scalenia warstwy BT z warstwą wcześniejszą, co pozwala uniknąć spowolnienia działania modelu. Dokonujemy tego poprzez zaktualizowanie wag i obciążień tej wcześniejszej warstwy tak, że będzie ona

bezpośrednio generowała wyniki o odpowiedniej skali i odpowiednim przesunięciu. Na przykład jeżeli poprzednia warstwa oblicza $XW + b$, to warstwa BN będzie obliczać $\gamma \otimes (XW + b - \mu) / \sigma + \beta$ (pomijamy tu człon wygładzający w mianowniku). Jeżeli zdefiniujemy $W' = \gamma \otimes W / \sigma$ i $b' = \gamma \otimes (b - \mu) / \sigma + \beta$, to równanie uprości się do postaci $XW' + b'$. Jeśli więc zastąpimy wagę (W) i obciążenia (b) we wcześniejszej warstwie aktualizowanymi wagami (W') i obciążeniami (b'), to pozbędziemy się warstwy BN (optymalizator w module TFLite wykonuje to automatycznie; zob. rozdział 19.).



Proces uczenia może wydawać się powolny, ponieważ każda epoka trwa znacznie dłużej po zastosowaniu algorytmu normalizacji wsadowej. Zazwyczaj równoważy to fakt, że model staje się zbieżny z dobrym rozwiązaniem dużo szybciej, dlatego do osiągnięcia optymalnej wydajności wymaganych jest mniej epok. Ostatecznie **rzeczywisty czas** (ang. *wall time*) działania algorytmu, czyli czas wskazywany przez zegar ścienny, powinien być krótszy.

Implementacja normalizacji wsadowej za pomocą interfejsu Keras

Implementacja normalizacji wsadowej jest, podobnie jak większość rzeczy w interfejsie Keras, prosta i intuicyjna. Wystarczy dodać warstwę BatchNormalization na początku lub na końcu funkcji aktywacji w każdej warstwie ukrytej, a dodatkowo można również dodać warstwę BN jako pierwszą warstwę modelu. Na przykład poniższy model zawiera warstwę normalizacji wsadowej jako pierwszą warstwę modelu (po spłaszczeniu obrazów wejściowych) oraz po każdej warstwie ukrytej:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

I to wszystko! W tak prostym przykładzie zawierającym zaledwie dwie warstwy ukryte prawdopodobnie normalizacja wsadowa nie będzie miała istotnego wpływu, ale w głębszych sieciach neuronowych może zrobić ogromną różnicę.

Spójrzmy na podsumowanie modelu:

```
>>> model.summary()
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
batch_normalization_v2 (BatchNormalization)	(None, 784)	3136
dense_50 (Dense)	(None, 300)	235500
batch_normalization_v2_1 (BatchNormalization)	(None, 300)	1200
dense_51 (Dense)	(None, 100)	30100

<i>batch_normalization_v2_2</i> (<i>Ba</i> (<i>None</i> , 100))		400
<i>dense_52</i> (<i>Dense</i>)	(<i>None</i> , 10)	1010
<hr/>		
<i>Total params:</i>	271,346	
<i>Trainable params:</i>	268,978	
<i>Non-trainable params:</i>	2,368	

Jak widać, każda warstwa BN dodaje cztery parametry na każde wejście: γ , β , μ i σ (na przykład pierwsza warstwa BN dodaje 3136 parametrów, czyli 4×784). Dwa ostatnie parametry, μ i σ , to średnie kroczące — propagacja wsteczna nie ma na nie żadnego wpływu, dlatego w terminologii Keras są one „niemodyfikowalne”⁹ (ang. *non-trainable*; jeżeli zsumujemy całkowitą liczbę parametrów normalizacji wsadowej: $3136+1200+400$ i podzielimy wynik przez 2, otrzymamy 2368 niemodyfikowalnych parametrów w tym modelu).

Sprawdźmy parametry pierwszej warstwy BN. Dwa są modyfikowalne (przez algorytm propagacji wstecznej), a dwa nie są:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Gdy teraz utworzymy warstwę BN w interfejsie Keras, wraz z nią zostaną przygotowane dwie operacje wywoływane w każdej iteracji uczenia. Operacje te odpowiadają za aktualizowanie średnich kroczących. Korzystamy z zaplecza TensorFlow, dlatego będą to operacje TF (zob. rozdział 12.):

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

Twórcy algorytmu BN sugerują, żeby warstwę normalizacji wsadowej umieszczać raczej przed funkcjami aktywacji, a nie za nimi (tak jak my to zrobiliśmy). Nie ma co do tego zgodności, gdyż wydaje się, że znaczenie ma rodzaj zadania; możesz samodzielnie przetestować obydwa rozwiązania, aby przekonać się, które sprawuje się lepiej w przypadku Twojego zestawu danych. Żeby móc dodać warstwy BN przed funkcjami aktywacji, należy usunąć funkcje aktywacji z warstw ukrytych i wstawić je jako osobne warstwy po warstwach normalizacji wsadowej. Ponadto warstwa BN zawiera po jednym parametrze przesunięcia na każde wejście, dlatego możesz usunąć człon obciążenia z warstwy poprzedzającej (wystarczy przekazać `use_bias=False` podczas tworzenia tej warstwy):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

⁹ Zostają one jednak oszacowane w trakcie trenowania na podstawie danych uczących, zatem prawdopodobnie są modyfikowalne. W kontekście interfejsu Keras „niemodyfikowalne” w rzeczywistości oznaczają „nietkniete przez algorytm propagacji wstecznej”.

Klasa BatchNormalization zawiera sporo hiperparametrów do strojenia. Wartości domyślne powinny zazwyczaj wystarczyć, ale sporadycznie może być konieczne zmodyfikowanie hiperparametru momentum. Jest on używany przez warstwę BatchNormalization podczas aktualizowania wękladniczych średnich kroczących; dla danej nowej wartości v (np. nowego wektora wejściowych średnich lub odchyлеń standardowych obliczonych za pomocą bieżącej grupy) warstwa aktualizuje średnią kroczącą \hat{v} w następujący sposób:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Dobra wartość momentum zazwyczaj jest bliska 1, np. 0,9, 0,99, 0,999 (im większy zestaw danych i mniejsza grupa, tym więcej należy wstawiać dziewiątek).

Innym ważnym parametrem jest axis: za jego pomocą wyznaczamy oś, która ma zostać znormalizowana. Jego domyślna wartość to -1, co oznacza, że normalizowana będzie ostatnia oś (za pomocą średnich i odchyłeń standardowych obliczanych ze wszystkich **pozostałych** osi). Jeżeli grupa wejściowa jest dwuwymiarowa (tzn. jej wymiary to [rozmiar grupy, cechy]), oznacza to, że każda cecha wejściowa będzie normalizowana na podstawie średniej i odchylenia standardowego obliczonych ze wszystkich przykładów tworzących grupę. Na przykład pierwsza warstwa BN w powyższym listingu będzie niezależnie normalizować (a także skalować i przesuwać) każdą z 784 cech wejściowych. Jeżeli przeniesiemy warstwę normalizacji wsadowej przed warstwę Flatten, to grupy wejściowe będą trójwymiarowe ([rozmiar grupy, wysokość, szerokość]), zatem warstwa BN będzie obliczać 28 średnich i tyle samo odchyłeń standardowych (po jednym parametrze każdego typu na kolumnę pikseli obliczonym dla wszystkich przykładów tworzących grupę i dla wszystkich rzędów w kolumnie) i znormalizuje wszystkie piksele w danej kolumnie za pomocą tej samej średniej i tego samego odchylenia standardowego. Występować będzie także 28 parametrów skalowania i 28 parametrów przesunięcia. Gdybyśmy chcieli przetwarzać każdy z 784 pikseli niezależnie od pozostałych, powinniśmy wyznaczyć axis=[1, 2].

Zwrót uwagę, że w warstwie BN nie są przeprowadzane takie same obliczenia przed treningiem i po jego zakończeniu: w trakcie uczenia wykorzystywane są statystyki grupy, natomiast później zostają użyte statystyki „ostateczne” (np. wartości końcowe średnich kroczących). Zajrzymy do kodu źródłowego tej klasy i przekonajmy się, jak jest obsługiwana:

```
class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

Za obliczenia odpowiedzialna jest metoda call() — jak widać, zawiera ona dodatkowy argument training, który ma domyślnie wyznaczoną wartość None, ale metoda fit() zmienia ją na 1 w fazie uczenia. Gdybyśmy musieli kiedykolwiek napisać niestandardową warstwę, która miałaby działać inaczej w trakcie uczenia i testowania, należy dodać argument training do metody call() i określić za jego pomocą, co ma być obliczane¹⁰ (w rozdziale 12. zajmiemy się dokładniej warstwami niestandardowymi).

¹⁰ Interfejs Keras definiuje również funkcję keras.backend.learning_phase(), która powinna zwracać 1 w fazie uczenia i 0 w każdej innej sytuacji.

Warstwa BatchNormalization zyskała tak olbrzymią popularność w głębszych sieciach neuronowych, że często jest pomijana na diagramach, gdyż zakłada się, że zostaje umieszczona po każdej warstwie. Może to jednak zmienić niedawno opublikowany artykuł (<https://arxiv.org/abs/1901.09321>)¹¹, w którym Hongyi Zhang i in. opisują autorską technikę **inicjalizacji wag stałymi** (ang. *fixed update initialization — fixup*), dzięki której byli w stanie wytrenować bardzo głębsze sieci (10 000 warstw!) bez normalizacji wsadowej i osiągać niebywałą wydajność w skomplikowanych zadaniach klasyfikacji obrazów. Są to jednak bardzo świeże badania, dlatego lepiej poczekać na kolejne doniesienia, zanim zdecydujesz się zrezygnować z normalizacji wsadowej.

Obcinanie gradientu

Inną popularną techniką służącą ograniczaniu problemu eksplodujących gradientów jest obcinanie gradientów na etapie propagacji wstecznej w taki sposób, żeby nigdy nie przekraczały określonego progu. Proces ten nosi nazwę **obcinania gradientu** (ang. *gradient clipping*)¹². Technikę tę najczęściej stosuje się w rekurencyjnych sieciach neuronowych, ponieważ, jak się przekonasz w rozdziale 15., trudno jest wprowadzać w nich algorytm normalizacji wsadowej. W pozostałych typach sieci normalizacja wsadowa powinna zazwyczaj zupełnie wystarczyć.

W interfejsie Keras implementacja obcinania gradientu sprowadza się do wyznaczenia argumentu `clipvalue` lub `clipnorm` podczas tworzenia optymalizatora:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Optymalizator ten obcina każdą składową wektora gradientów do wartości z przedziału od -1,0 do 1,0. Oznacza to, że wszystkie pochodne cząstkowe funkcji straty (w odniesieniu do każdego parametru modyfikowalnego) zostaną przycięte do wartości z zakresu od -1,0 do 1,0. Wartość progowa jest hiperparametrem, który możesz dostroić. Zwróć uwagę, że kierunek wektora gradientów może ulec zmianie. Na przykład jeżeli pierwotny wektor gradientów ma wartości [0,9, 100,0], to skierowany jest głównie stronę drugiej osi, ale po przycięciu gradientów otrzymujemy [0,9, 1,0], przez co wektor ułoży się teraz po przekątnej niemal idealnie pomiędzy obydwoma osiami. W praktyce rozwiązanie to sprawdza się dobrze. Jeżeli nie chcesz, aby obcinanie gradientów zmieniło kierunek wektora, użyj argumentu `clipnorm`, który obcina gradienty na podstawie normy. Gradient zostanie obcięty wtedy, gdy jego norma ℓ_2 jest większa od wyznaczonej wartości progowej. Przykładowo jeśli wyznaczysz `clipnorm=1.0`, to wektor [0,9, 100] zostanie przycięty do wartości [0,00899964, 0,9999595], co spowoduje zachowanie kierunku przy jednoczesnym niemal całkowitym wyeliminowaniu pierwszej składowej. Jeżeli zauważysz, że gradienty eksplodują w trakcie uczenia (możesz śledzić wartości gradientów w aplikacji TensorBoard), wypróbuj obcinanie gradientów za pomocą wartości i normy dla różnych wartości progowych i sprawdź, które rozwiązanie sprawdza się najlepiej w przypadku zestawu walidacyjnego.

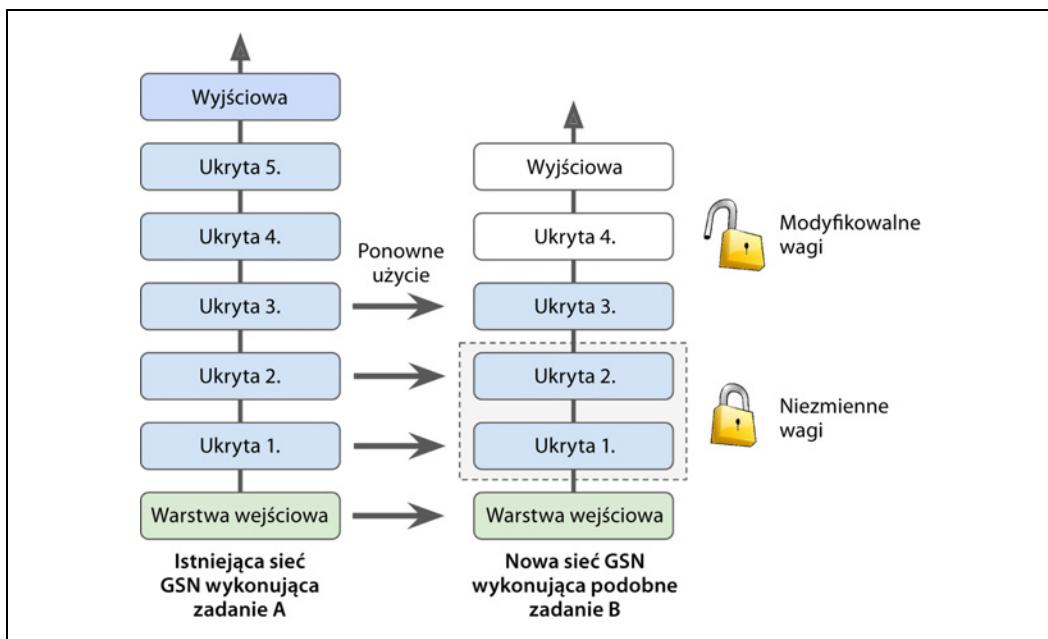
¹¹ Hongyi Zhang i in., *Fixup Initialization: Residual Learning Without Normalization*, arXiv preprint arXiv: 1901.09321 (2019).

¹² Razvan Pascanu i in., *On the Difficulty of Training Recurrent Neural Networks*, „Proceedings of the 30th International Conference on Machine Learning” (2013): 1310 – 1318, <https://arxiv.org/abs/1211.5063>.

Wielokrotne stosowanie gotowych warstw

Zazwyczaj uczenie bardzo dużych sieci GSN nie jest zbyt dobrym pomysłem i zawsze lepiej poszukać jakiejś istniejącej sieci dostosowanej do podobnego zadania (w rozdziale 14. opiszę sposób ich wyszukiwania), jakim się zajmujesz, a następnie po prostu wykorzystać jej niższe warstwy — jest to tak zwane **uczenie transferowe** (ang. *transfer learning*). Nie tylko znaczco przyspiesza ono proces uczenia, lecz także wymaga dużo mniej danych uczących.

Załóżmy na przykład, że mamy dostęp do głębokiej sieci neuronowej, która została wyuczona do klasyfikowania obrazów pomiędzy 100 różnych kategorii, takich jak zwierzęta, rośliny, pojazdy oraz przedmioty. Chcemy teraz wytrenować sieć do klasyfikowania określonych typów pojazdów. Zadania te są bardzo do siebie podobne, często nawet pokrywają się częściowo, dlatego warto byłoby wykorzystać elementy już istniejącej sieci (rysunek 11.4).



Rysunek 11.4. Ponowne wykorzystanie gotowych warstw



Jeśli zdjęcia wykorzystywane w Twoim zadaniu mają inny rozmiar od używanych w pierwotnym zadaniu, musisz dodać etap wstępnego przetwarzania danych i dopasować ich wymiary do wymiarów oczekiwanych przez istniejący model. Zasadniczocześnie transferowe będzie działać najlepiej wtedy, gdy ogólne cechy danych wejściowych będą do siebie podobne.

Zazwyczaj należy zastąpić warstwę wyjściową pierwotnego modelu, gdyż najprawdopodobniej nie będzie zupełnie dostosowana do nowego zadania, a może nawet zawierać niewłaściwą liczbę wyjść.

Podobnie wygląda sytuacja z górnymi warstwami: nie będą one raczej tak przydatne jak warstwy niższe, ponieważ cechy szczegółowe przydatne w danym zadaniu mogą wyglądać zupełnie inaczej

w porównaniu do cech użytecznych w zadaniu pierwotnym. Musisz określić odpowiednią liczbę zapożyczonych warstw.



Im bardziej podobne są zadania, tym więcej warstw chcesz pożyczyć (począwszy od najniższych). W przypadku bardzo podobnych zadań próbuj pozostawić wszystkie warstwy ukryte i zastąpić tylko warstwę wyjściową.

Spróbuj najpierw zamrozić wszystkie przejmowane warstwy (tzn. zablokuj możliwość modyfikowania wag przez algorytm gradientu prostego), a potem wytrenuj model i sprawdź, jak się spisuje. Następnie odmroź jedną albo dwie górne warstwy ukryte, poczekaj, aż zostaną zmodyfikowane przez algorytm propagacji wstecznej, i porównaj jego wydajność. Im więcej masz danych uczących, tym więcej warstw możesz odmrozić. Warto również zmniejszyć współczynnik uczenia podczas odmrażania wielokrotnie używanych warstw — nie zrzucaj w ten sposób ich dostrojonych wag.

Jeżeli ciągle masz problem z uzyskaniem dobrej wydajności i dysponujesz niewielką ilością danych, spróbuj porzucić co najmniej jedną górną warstwę ukrytą i ponownie zamroź pozostałe warstwy ukryte. Powtarzaj tę czynność aż do uzyskania właściwej liczby wielokrotnie używanych warstw. Jeżeli masz dużo danych uczących, możesz zastępować, a nawet dodawać kolejne warstwy ukryte zamiast je usuwać,

Uczenie transferowe w interfejsie Keras

Załóżmy, że zestaw danych Fashion MNIST zawiera tylko osiem klas (na przykład wszystkie oprócz przedstawiających sandały i koszulki). Ktoś zbudował i wytrenował model Keras na takim zestawie danych, dzięki czemu uzyskiwał całkiem dobre rezultaty (dokładność przekraczająca 90%). Nazwijmy go modelem A. Być może dostajesz teraz inne zadanie: dysponujesz zdjęciami sandałów i koszulek, a Twoim celem jest wytrenowanie klasyfikatora binarnego (klasa pozytywna = koszulka, negatywna = sandał). Zestaw danych jest raczej mały: masz tylko 200 oznakowanych zdjęć. Podczas trenowania nowego modelu (nazwijmy go modelem B) o takiej samej strukturze jak model A, na tych danych, osiągasz dobrą wydajność (97,2% dokładności), ale skoro jest to prostsze zadanie (występują w nim tylko dwie klasy), spodziewałaś/spodziewałeś się więcej. Delekując się poranną kawą, uświadamiasz sobie nagle, że zadania wyznaczone modelem A i B są w istocie bardzo podobne, dlatego zastanawiasz się, czy może się tu przydać uczenie transferowe. Sprawdźmy to!

Musisz najpierw wczytać model A i utworzyć na jego podstawie nowy model wykorzystujący jego warstwy. Użyjmy ponownie wszystkich warstw oprócz warstwy wyjściowej:

```
model_A = keras.models.load_model("moj_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Teraz model_A i model_B_on_A mają pewne warstwy takie same. Proces uczenia modelu model_B_on_A będzie wpływać również na model_A. Jeśli chcesz tego uniknąć, musisz sklonować model_A, zanim zaczniesz korzystać wielokrotnie z jego warstw. W tym celu **klonujemy** strukturę modelu A za pomocą funkcji `clone_model()`, a następnie kopujemy jego wag (gdyż funkcja `clone_model()` nie jest za to odpowiedzialna):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Możesz teraz wytrenować model_B_on_A w wykonywaniu zadania B, ale nowa warstwa wyjściowa została zainicjalizowana losowo, dlatego będzie popełniać duże błędy (przynajmniej w ciągu kilku pierwszych epok), zatem wystąpią duże gradienty błędu, które mogą całkowicie popsuć wagę w pozytywnych warstwach. Możemy tego uniknąć, zamrażając te warstwy przez kilka pierwszych epok, dzięki czemu damy czas nowej warstwie na poznanie rozsądnych wartości wag. Wyznaczamy więc w każdej warstwie parametr trainable z wartością False i komplilujemy model:

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = False  
  
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",  
                      metrics=["accuracy"])
```



Należy zawsze kompilować model po zamrożeniu lub rozmrożeniu warstw.

Wytrenuj teraz model przez kilka epok, rozmroź zamrożone warstwy (będzie to wymagać ponownej komplikacji modelu) i kontynuuj trenowanie modelu w wykonywaniu zadania B. Po rozmrożeniu warstw warto zmniejszyć współczynnik uczenia, co pomoże ochronić ich wagę:

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,  
                            validation_data=(X_valid_B, y_valid_B))  
  
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(lr=1e-4) # Domyślna wartość lr wynosi 1e-2  
model_B_on_A.compile(loss="binary_crossentropy", optimizer=optimizer,  
                      metrics=["accuracy"])  
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                            validation_data=(X_valid_B, y_valid_B))
```

Jaki jest więc końcowy werdykt? Dokładność tego modelu dla zestawu testowego wynosi 99,25%, co oznacza, że uczenie transferowe zmniejszyło stopę błędu z 2,8% do niemal 0,7%, czyli czterokrotnie!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)  
[0.06887910133600235, 0.9925]
```

Przekonało Cię to? Nie powinno, bo oszukiwałem! Wypróbowałem wiele konfiguracji, dopóki nie znalazłem wykazującej znaczną poprawę. Jeżeli zmienisz klasy lub ziarno losowości, to wydajność najprawdopodobniej spadnie albo wręcz zaniknie. Zastosowałem tu technikę zwaną „torturowaniem danych aż do uzyskania odpowiedzi”. Gdy jakiś artykuł wydaje się zbyt entuzjastyczny, to należy zachować podejrzliwość: prawdopodobnie wychwalana nowa technologia nie okazuje się za bardzo przydatna (a czasami może wręcz zaszkodzić), ale autorzy wypróbowali wiele wariantów i opisali wyłącznie najlepsze rezultaty (które mogą być jedynie kwestią szczęścia), nie chwaląc się liczbą niepowodzeń, jakim musieli stawić czoła. Zazwyczaj nie ma to negatywnego wpływu, ale stanowi jeden z powodów faktu, że tak wielu wyników w nauce nie da się odtworzyć.

Dlaczego oszukałem? Okazuje się, że uczenie transferowe nie działa zbyt dobrze na małych, gęstych sieciach, prawdopodobnie dlatego, że małe sieci poznają niewiele wzorców, a sieci gęste uczą się określonych wzorców, które prawdopodobnie okażą się bezużyteczne w innych zadaniach. Uczenie

transferowe jest najskuteczniejsze w głębkich sieciach splotowych, które często pełnią rolę ogólnych wykrywaczy cech (dotyczy to zwłaszcza niższych warstw). W rozdziale 14. powrócimy jeszcze do uczenia transferowego i skorzystamy z właśnie poznanych technik (obiecuje, że obędzie się bez oszukiwania!).

Nienadzorowane uczenie wstępne

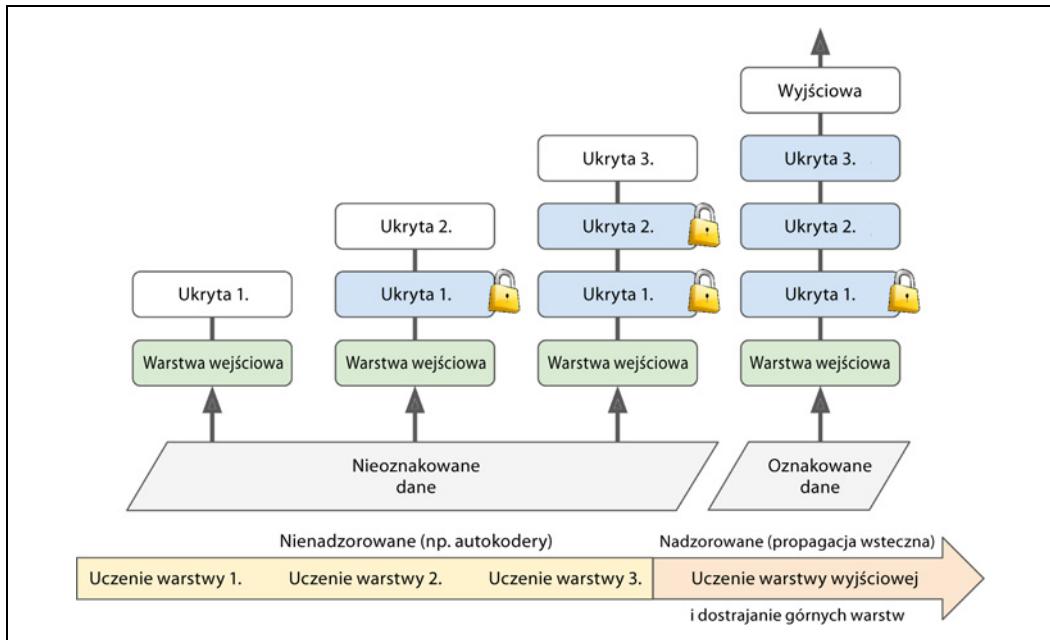
Załóżmy, że chcesz zajmować się skomplikowanym zadaniem, dla którego nie masz wielu oznaczonych danych uczących, ale jednocześnie nie jesteś w stanie znaleźć modelu wyuczonego do podobnego zadania. Nie trać nadziei! Oczywiście najpierw należy spróbować zdobyć jak najwięcej oznakowanych przykładów, ale jeśli jest to zbyt trudne lub kosztowne, pozostaje Ci jeszcze **nienadzorowane uczenie wstępne** (ang. *unsupervised pretraining*) (rysunek 11.5). Istotnie, pozyskanie nieoznakowanych przykładów uczących jest tanie w porównaniu do kosztów ich oznakowania. Jeżeli masz wiele nieoznakowanych danych uczących, możesz spróbować trenować model nienadzorowany, taki jak autokoder czy generatywna sieć przeciwnastawna (zob. rozdział 17.). Następnie możesz wykorzystać dolne warstwy autokodera lub dyskryminatora sieci przeciwnastawnej, umieścić na szczytce warstwę wyjściową dostosowaną do Twojego zadania i dostroić model końcowy za pomocą uczenia nadzorowanego (tzn. za pomocą przykładów oznakowanych).

To właśnie za pomocą tej techniki Geoffrey Hinton i jego zespół ponownie wzbudzili zainteresowanie sieciami neuronowymi i sukcesem uczenia głębowego w 2006 roku. Nienadzorowane uczenie wstępne, zazwyczaj wraz z ograniczonymi maszynami Boltzmanna (zob. dodatek E), aż do 2010 roku stanowiło nieodzowny element sieci głębkich, a dopiero po rozwiązaniu problemu zanikających gradienłów został znacznie upowszechniony mechanizm uczenia sieci głębkich w sposób czysto nadzorowany. Nienadzorowane uczenie wstępne (w którym obecnie częściej są wykorzystywane autokodery lub sieci GAN zamiast ograniczonych maszyn Boltzmanna) do teraz stanowią dobry wybór w przypadku konieczności rozwiązania skomplikowanego zadania, a nie możesz znaleźć podobnego modelu do użycia oraz gdy dysponujesz niewielką liczbą danych oznakowanych, a za to wieloma danymi nieoznakowanymi.

Pamiętaj, że bardzo trudno było uczyć pierwsze modele głębokie, dlatego z chęcią stosowano technikę zwaną **zachłannym uczeniem wstępny warstwa po warstwie** (ang. *greedy layer-wise pretraining*) (rysunek 11.5). Najpierw trenowana była pierwsza warstwa modelu nienadzorowanego, najczęściej ograniczona maszyną Boltzmanna, którą następnie zamrażano i dodawano na jej końcu kolejną warstwę; model był uczyony ponownie (a właściwie tylko dodana warstwa), po czym dołączona warstwa była również zamrażana, dodawano kolejną warstwę itd. Obecnie sytuacja wygląda znacznie prościej: zazwyczaj za jednym zamachem jest uczyony cały model nienadzorowany (trzeci etap na rysunku 11.5), a ograniczone maszyny Boltzmanna zostają zastąpione autokoderami lub generatywnymi sieciami przeciwnastawnymi.

Uczenie wstępne za pomocą dodatkowego zadania

Jeżeli brakuje Ci danych oznakowanych, ostatnią możliwością jest uczenie pierwszej sieci neuronowej za pomocą dodatkowego zadania, dla którego można z łatwością pozyskać lub wygenerować oznakowane dane uczące, a następnie wykorzystanie dolnych warstw tej sieci do przetwarzania



Rysunek 11.5. W uczeniu nienadzorowanym model jest trenowany na danych nieoznakowanych (lub na wszystkich danych) za pomocą techniki uczenia nienadzorowanego, a następnie zostaje dostrojony do zadania końcowego przy użyciu techniki uczenia nadzorowanego. Jak zostało pokazane, część nienadzorowana może uczyć po jednej warstwie lub może wytrenować bezpośrednio pełny model

właściwego zadania. Dolne warstwy neuronowe pierwszej sieci będą korzystały z wykrywaczy cech, które prawdopodobnie zostaną użyte w drugiej sieci.

Przykładowo jeśli chcesz stworzyć system rozpoznawania twarzy, możesz mieć dostęp tylko do kilku zdjęć każdej osoby — zdecydowanie za mało, żeby wyuczyć dobry klasyfikator. Pozyskanie setek zdjęć każdej osoby byłoby z kolei niepraktyczne. Jednak jesteśmy w stanie pobrać z internetu mnóstwo zdjęć losowych ludzi i wytrenować pierwszą sieć neuronową do wykrywania, czy na dwóch różnych zdjęciach znajduje się ta sama osoba. Tego typu sieć mogłaby wyuczyć skuteczne detektory cech dla twarzy, zatem ponowne wykorzystanie jej niższych warstw pozwoliłoby wytrenować skuteczny klasyfikator twarzy za pomocą niewielkiej liczby danych uczących.

W przypadku zadań **przetwarzania języka naturalnego** (ang. *Natural Language Processing* — NLP) możesz pobrać korpus składający się z milionów dokumentów i automatycznie wygenerować z niego oznakowane dane. Na przykład możesz losowo „zakryć” pewne słowa i wytrenować model do prognozowania tych brakujących wyrazów (np. powinien przewidzieć, że brakującym słowem w pytaniu „_____ mi coś powiedzieć?” jest prawdopodobnie „chcesz”, „chciałeś/chciałaś” albo „chciałbyś/chciałabyś”). Jeżeli wyuczysz model tak, aby osiągał dobre rezultaty, to będzie cechował się całkiem niezłą znajomością języka i zdecydowanie warto go będzie wypróbować we właściwym zadaniu i dostroić do danych oznakowanych (więcej informacji na temat zadań uczenia wstępnego znajdziesz w rozdziale 15.).



Uczenie samonadzorowane (ang. *self-supervised learning*) występuje wtedy, gdy automatycznie generujesz etykiety z danych, a następnie trenujesz model za pomocą tak „oznakowanych” danych przy użyciu technik uczenia nadzorowanego. W takim rozwiązaniu człowiek nie musi oznaczać danych, dlatego najlepiej uznać je za odmianę uczenia nienadzorowanego.

Szybsze optymalizatory

Uczenie bardzo dużych sieci DNN może być niezwykle czasochłonne. Do tej pory omówiłem cztery sposoby przyspieszenia procesu uczenia (i uzyskania lepszych wyników): wybranie odpowiedniej strategii inicjalizowania wag połączeń, zastosowanie dobrej funkcji aktywacji, wprowadzenie normalizacji wsadowej i wielokrotne stosowanie składników już istniejącej sieci (być może opartej na dodatkowym zadaniu lub stworzonej za pomocą uczenia nienadzorowanego). Kolejnym elementem mającym olbrzymi wpływ na szybkość nauki jest dobór optymalizatora szybszego od standardowego algorytmu gradientu prostego. W tym podrozdziale zajmiemy się najpopularniejszymi rodzajami optymalizatorów: momentum, algorytmem Nesterova (przyspieszony spadek wzduż gradientu), AdaGrad, RMSProp, a także algorytmami Adam i Nadam.

Optymalizacja momentum

Wyobraź sobie kulę do kręgla toczącą się z niewielkiego pagórka: najpierw będzie przesuwała się powoli, ale szybko nabierze rozpędu aż do osiągnięcia prędkości granicznej (wynikającej z tarcia lub oporu powietrza). Jest to koncepcja stanowiąca podstawę **optymalizacji momentum** (ang. *momentum optimization*), zaproponowana przez Borisa Polyaka w 1964 roku (https://www.researchgate.net/publication/243648538_Some_methods_of_speeding_up_the_convergence_of_iteration_methods)¹³. W porównaniu do tej metody algorytm gradientu prostego wykonuje małe krocze wzduż nachylenia funkcji, dlatego osiągnięcie minimum zajmuje w tym przypadku znacznie więcej czasu.

Przypominam, że w metodzie gradientu prostego wagi θ są aktualizowane poprzez odejmowanie gradientu funkcji kosztu $J(\theta)$ od wag ($\nabla_{\theta} J(\theta)$) pomnożonych przez współczynnik uczenia η . Wzór ten wygląda następująco: $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$. Wcześniej sze gradiensty nie są tu brane pod uwagę. Jeżeli lokalny gradient jest niewielki, proces przebiega bardzo powoli.

Optymalizacja momentum z kolei bierze pod uwagę wcześniejsze gradiensty: w każdym przebiegu lokalny gradient jest odejmowany od **wektora momentu m** (pomnożonego przez współczynnik uczenia η), a wagi są aktualizowane poprzez dodanie tego wektora (równanie 11.4). Innymi słowy, gradient jest tu wykorzystywany jako przyspieszenie, a nie jako prędkość. W celu symulowania zjawiska tarcia zapobiegającego nadmiernemu wzrostowi wartości pędu został wprowadzony nowy hiperparametr β , zwany po prostu **momentem** (albo **pędem**) — jego wartość mieści się w zakresie pomiędzy 0 (duże tarcie) a 1 (brak tarcia). Typowa wartość momentu to 0,9.

¹³ Boris T. Polyak, *Some Methods of Speeding Up the Convergence of Iteration Methods*, „USSR Computational Mathematics and Mathematical Physics” 4, no. 5 (1964), s. 1 – 17.

Równanie 11.4. Algorytm optymalizacji momentum

$$1. \quad \mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

Możemy łatwo sprawdzić, że przy stałym gradiencie prędkość graniczna (tj. maksymalny rozmiar aktualizacji wag) będzie równa iloczynowi tego gradientu, współczynnika uczenia i wyrażenia $\frac{1}{1-\beta}$ (znak nie ma znaczenia). Przykładowo jeśli $\beta = 0,9$, to prędkość graniczna będzie wynosiła $10 \times \text{gradient} \times \text{współczynnik uczenia}$, zatem optymalizacja momentum okazuje się dziesięciokrotnie szybsza od algorytmu gradientu prostego! Dzięki temu optymalizacja momentum jest w stanie znacznie szybciej „uciekać” z wypłaszczeń funkcji. Dotyczy to zwłaszcza sytuacji takich, jak opisana w rozdziale 4., gdzie mieliśmy do czynienia z funkcją kosztu o kształcie wydłużonego dzwonu (rysunek 4.7). Algorytm gradientu prostego całkiem szybko schodzi po stromych odcinkach, ale przejście po „dolinie” zabiera mu znacznie więcej czasu. Z kolei algorytm optymalizacji momentum będzie „stała” się coraz szybciej aż do osiągnięcia dna (optimum). W głębokich sieciach neuronowych niewykorzystujących normalizacji wsadowej górne warstwy bardzo często mają wejścia zdefiniowane w różnych skalach, dlatego optymalizacja momentum okazuje się bardzo pomocna. Ułatwia ona również „ucieczkę” z lokalnych minimów.



Z powodu pędu optymalizator może nieznacznie ominąć minimum globalne, cofnąć się i znowu je ominąć i tak oscylować aż do osiągnięcia dna. Jest to jedna z przyczyn, dla których warto wyznaczyć niewielką wartość tarcia: redukujemy w ten sposób oscylacje i przyśpieszamy osiągnięcie zbieżności.

Implementacja optymalizacji momentum w interfejsie Keras to fraszka: wystarczy, że wyznaczysz optymalizator SGD wraz z hiperparametrem momentum, a następnie możesz leżeć i pachnieć!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

Największa wada optymalizacji momentum polega na tym, że dodaje ona kolejny hiperparametr do strojenia. Jednak jego wartość 0,9 sprawdza się dobrze w większości sytuacji i niemal zawsze gwarantuje działanie szybsze od gradientu prostego.

Przyspieszony spadek wzduż gradientu (algorytm Nesterova)

Jedna niewielka modyfikacja optymalizacji momentum, zaproponowana w 1983 roku przez Yuriiego Nesterova (<https://homl.info/55>)¹⁴, jest niemal zawsze szybsza od standardowej wersji algorytmu. Konsepcją kryjącą się za **optymalizacją momentum Nesterova** (ang. *Nesterov momentum optimization*), zwaną również **algorytmem Nesterova** lub **przyspieszonym spadkiem wzduż gradientu** (ang. *Nesterov accelerated gradient* — NAG), jest pomiar gradientu funkcji kosztu nie w lokalnej pozycji θ , ale nieco z przodu w kierunku pędu $\theta + \beta \mathbf{m}$ (równanie 11.5).

¹⁴ Yurii Nesterov, *A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k²)*, „Doklady AN USSR” 269 (1983), s. 543 – 547.

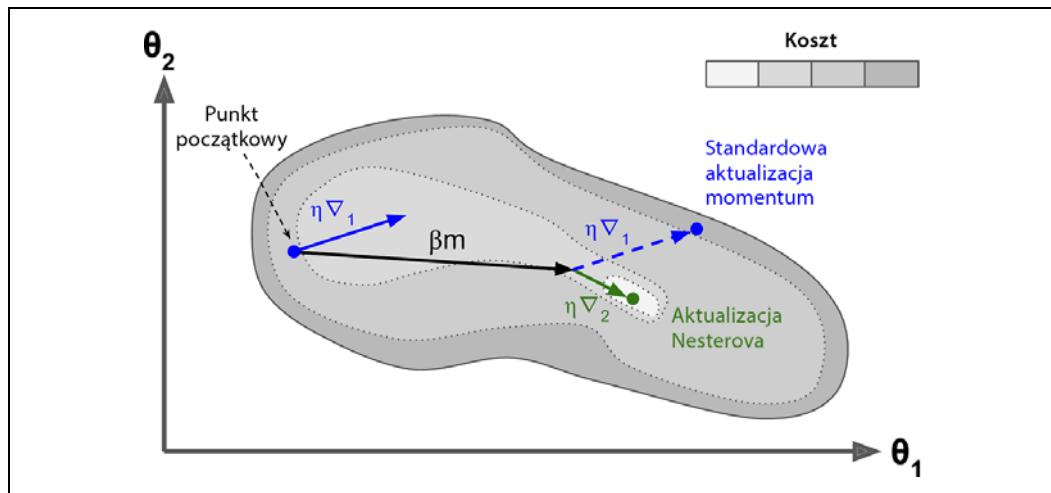
Równanie 11.5. Algorytm przyspieszonego spadku wzduż gradientu

$$1. \quad \mathbf{m} \leftarrow \beta\mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta\mathbf{m})$$

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

Ta modyfikacja jest skuteczna, ponieważ zazwyczaj wektor pędu będzie skierowany we właściwym kierunku (tj. w stronę optimum), zatem wykorzystanie gradientu znajdującego się nieco z przodu będzie nieznacznie dokładniejsze od użycia gradientu znajdującego się w lokalnie analizowanym punkcie, co zostało ukazane na rysunku 11.6 (gdzie ∇_1 reprezentuje gradient funkcji kosztu zmierzony w punkcie początkowym θ , natomiast ∇_2 oznacza gradient zmierzony dla punktu $\theta + \beta\mathbf{m}$).

Jak widać, aktualizacja Nesterova trafia nieco bliżej optimum. Po pewnym czasie te drobne różnice sumują się i algorytm NAG okazuje się znacznie szybszy od standardowej optymalizacji momentum. Ponadto zwróci uwagę, że gdy pęd kieruje się na drugą stronę doliny, ∇_1 nie zmienia kierunku, natomiast ∇_2 zwraca ku minimum. Ograniczamy w ten sposób oscylacje, dzięki czemu model jeszcze szybciej osiąga zbieżność.



Rysunek 11.6. Porównanie optymalizacji momentum z algorytmem Nesterova: w tej pierwszej gradienty są obliczane przed fazą momentu, natomiast w drugim algorytmie gradienty są stosowane po obliczeniu momentów

Algorytm NAG niemal zawsze przyśpiesza proces uczenia w porównaniu do standardowej optymalizacji momentum. Żeby z niego skorzystać, wystarczy wprowadzić parametr nesterov=True podczas tworzenia klasy SGD:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad

Wróćmy jeszcze na chwilę do problemu funkcji o kształcie wydłużonego dzwonu: model gradientu prostego najpierw szybko schodzi po największej pochyłości, która nie zawsze jest skierowana ku optimum globalnemu, po czym bardzo powoli zmierza ku minimum globalnemu. Byłoby miło, gdyby algorytm był w stanie wykrywać odpowiednio wcześniej zmianę kierunku i korygować przebieg bardziej w stronę minimum. Algorytm **AdaGrad**¹⁵ (<http://jmlr.org/papers/v12/duchi11a>) dokonuje tego poprzez stopniowe zmniejszanie wektora gradientów wzduż najbardziej stromych kierunków (równanie 11.6).

Równanie 11.6. Algorytm AdaGrad

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\boldsymbol{\theta}) \otimes \nabla_{\theta} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\theta} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

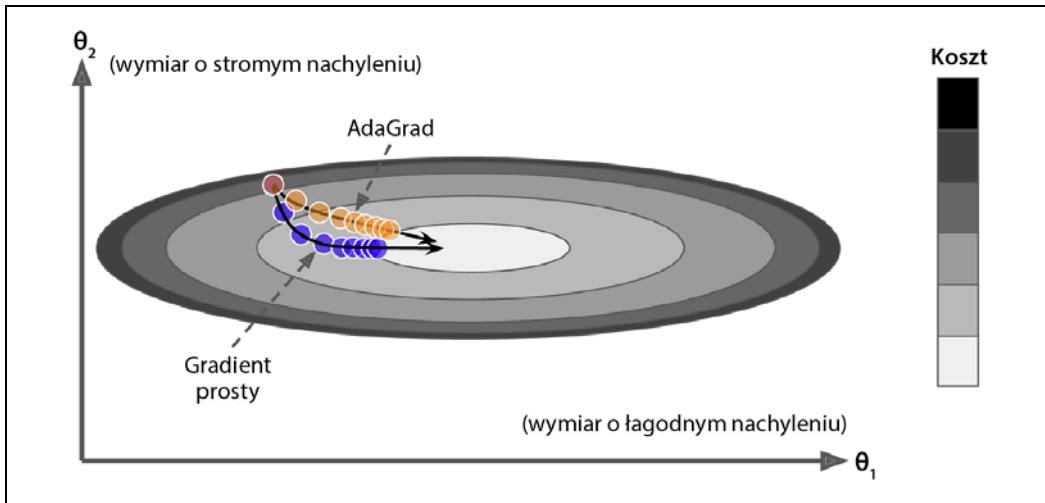
W pierwszym równaniu gromadzimy kwadrat gradientów w wektorze \mathbf{s} (symbol \otimes oznacza mnożenie poszczególnych elementów). Ta zwektryzowana postać jest równoważna obliczeniu $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$ elementu s_i w wektorze \mathbf{s} ; innymi słowy, każdy element s_i kumuluje kwadraty pochodnej częściowej funkcji kosztu przy uwzględnieniu parametru θ_i . Jeżeli funkcja kosztu jest stromo nachylona w i -tym wymiarze, to przy każdym przebiegu element s_i będzie miał coraz większą wartość.

Drugi etap niemal przypomina algorytm gradientu prostego, jednak widzimy tu jedną dużą różnicę: wektor gradientu jest zmniejszany o czynnik $\sqrt{\mathbf{s} + \epsilon}$ (symbol \oslash oznacza dzielenie przez poszczególne elementy, a parametr $\sqrt{\mathbf{s} + \epsilon}$ jest członem wygładzającym, służącym do unikania operacji dzielenia przez 0; zazwyczaj przyjmuje on wartość 10^{-10}). Ta zwektryzowana forma jest równoznaczna z operacją $\theta_i \leftarrow \theta_i - \eta \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{s_i + \epsilon}$ dla wszystkich parametrów θ_i (równocześnie).

Krótko mówiąc, algorytm ten redukuje współczynnik uczenia, ale dokonuje tego szybciej dla wymiarów o stromym przebiegu funkcji niż dla wymiarów o bardziej łagodnym przebiegu. Jest to tak zwany **adaptacyjny współczynnik uczenia** (ang. *adaptive learning rate*). Pomaga on nakierować uzyskiwane aktualizacje w kierunku globalnego optimum (rysunek 11.7). Dodatkową zaletą jest fakt, że wymaga on znacznie mniej strojenia od klasycznego hiperparametru η .

Algorytm AdaGrad często sprawdza się dobrze w przypadku prostych problemów kwadratowych, ale potrafi zatrzymać się zbyt wcześnie podczas uczenia sieci neuronowych. Współczynnik uczenia może zostać zredukowany w takim stopniu, że algorytm skończy pracę jeszcze przed osiągnięciem minimum globalnego. Zatem mimo obecności klasy Adagrad w interfejsie Keras nie należy używać jej do trenowania głębokich sieci neuronowych (jednak może być ona wydajna wobec prostszych zadań, np. regresji liniowej). Warto jednak poznać algorytm AdaGrad po to, aby lepiej zrozumieć inne optymalizatory wykorzystujące adaptacyjny współczynnik uczenia.

¹⁵ John Duchi i in., *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, „Journal of Machine Learning Research” 12 (2011), s. 2121 – 2159.



Rysunek 11.7. Porównanie algorytmu AdaGrad z klasycznym gradientem prostym: w tym pierwszym mogą być wcześniej korygowane kierunki do punktu optymalnego

RMSProp

Jak już wiesz, algorytm AdaGrad może zwalniać nieco zbyt szybko i kończyć działanie, nigdy nie osiągając zbieżności z minimum globalnym, ale problem ten zostaje rozwiązany przez algorytm **RMSProp**¹⁶ poprzez gromadzenie wyłącznie gradientów z najbardziej aktualnych przebiegów (a nie wszystkich gradientów od początku procesu nauki). Dokonuje tego poprzez przeprowadzenie rozkładu wykładniczego na pierwszym etapie (równanie 11.7).

Równanie 11.7. Algorytm RMSProp

1. $s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

Współczynnik rozkładu β zazwyczaj ma wartość 0,9. Tak, mamy do czynienia z kolejnym hiperparametrem, ale ta domyślna wartość często okazuje się wystarczająca, dlatego może nie będzie trzeba w ogóle go dostrajać.

Jak można było się spodziewać, w interfejsie znajdziemy klasę RMSprop:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Zwrót uwagę, że argument rho odpowiada współczynnikowi β z równania 11.7. Nie licząc bardzo prostych problemów, optymalizator ten niemal zawsze osiąga lepsze wyniki od algorytmu AdaGrad.

¹⁶Algorytm ten został stworzony przez Tijmena Tielemana i Geoffreya Hintona w 2012 roku, a następnie zaprezentowany przez drugiego autora na wykładzie Coursera dotyczącym sieci neuronowych (prezentacja: <https://homl.info/57>; video: <https://homl.info/58>). Co zabawne, autorzy nie opisali tego algorytmu w żadnej publikacji, dlatego badacze, odnosząc się do źródła, używają zapisu np. „slajd 29. w wykładzie 6.”.

W istocie, aż do momentu pojawienia się optymalizatora Adam, był to preferowany rodzaj algorytmu optymalizacyjnego.

Optymalizatory Adam i Nadam

Algorytm **Adam**¹⁷ (ang. *adaptive moment estimation* — szacowanie adaptacyjnego momentu) łączy koncepcję optymalizacji momentum i optymalizatora RMSProp: z tego pierwszego elementu bierze śledzenie rozkładu wykładniczego średniej wcześniejszych gradientów, a z tego drugiego — śledzenie rozkładu wykładniczego średniej wcześniejszych kwadratów gradientów (równanie 11.8)¹⁸:

Równanie 11.8. Algorytm 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. $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}}} + \epsilon$

W tym równaniu t wyznacza numer przebiegu (począwszy od 1).

Jeśli przyjrzymy się wyłącznie wzorom 1., 2. i 5., to zauważymy scisłe podobieństwo algorytmu Adam zarówno do optymalizacji momentum, jak i optymalizatora RMSProp. Jedyna różnica polega na tym, że w etapie 1. wyliczany jest rozkład wykładniczy średniej, a nie sumy, ale w rzeczywistości te operacje są równoważne, nie licząc współczynnika ważenia (średnia zanikająca to jedynie $1 - \beta_1$ pomnożone przez zanikającą sumę). Etapy 3. i 4. dotyczą niejako szczegółów technicznych: ponieważ wektory \mathbf{m} i \mathbf{s} są inicjalizowane z wartością 0, to na początku treningu będą obciążone w kierunku zera, zatem te dwa etapy służą wzmocnieniu tych wektorów na samym początku.

Hiperparametr rozkładu momentu β_1 jest zazwyczaj inicjalizowany z wartością 0,9, natomiast hiperparametr rozkładu skalowania β_2 często otrzymuje wartość 0,999. Podobnie jak we wcześniejszych przypadkach człon wygładzający ma niewielką wartość, np. 10^{-7} . Są to domyślne wartości klasy Adam (mówiąc dokładniej, wartość domyślna parametru epsilon to None, co oznacza dla interfejsu Keras, że ma użyć funkcji keras.backend.epsilon() dającej wartość 10^{-7} ; możemy zmienić tę wartość za pomocą funkcji keras.backend.set_epsilon()). Oto sposób utworzenia optymalizatora Adam za pomocą interfejsu Keras:

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

¹⁷ Diederik P. Kingma i Jimmy Ba, *Adam: A Method for Stochastic Optimization*, arXiv preprint arXiv: 1412.6980 (2014), <https://arxiv.org/abs/1412.6980>.

¹⁸ Są to oszacowania średniej i wariancji (niewyśrodkowanej) gradientów. Średnia jest często nazywana **pierwszym momentem** (ang. *first moment*), natomiast wariancja — **drugim momentem** (ang. *second moment*), stąd nazwa algorytmu.

W rzeczywistości skoro optymalizator Adam jest algorytmem wykorzystującym adaptacyjny współczynnik uczenia (podobnie jak AdaGrad i RMSProp), nie musimy bardzo męczyć się ze strojeniem hiperparametru η . Możesz zawsze skorzystać z domyślnej wartości $\eta = 0,001$, dzięki czemu optymalizator Adam jest jeszcze łatwiejszy w użyciu od metody gradientu prostego.



Nie martw się, jeżeli ta mnogość technik i konieczność wyboru którejś z nich trochę Cię przytłacza — na końcu rozdziału umieściłem pewne wytyczne.

Na koniec warto wspomnieć o dwóch odmianach optymalizatora Adam:

AdaMax

Spójrz na etap 2. w równaniu 11.8: algorytm Adam gromadzi kwadraty gradientów w wektorze s (bieżące gradienty mają większą wagę). Jeżeli na etapie 5. zignorujemy człon ϵ oraz etapy 3. i 4. (które i tak prezentują wyłącznie szczegóły techniczne), optymalizator przeskaluje aktualizacje parametrów przez pierwiastek kwadratowy z s . Krótko mówiąc, Adam przeskaluje aktualizacje parametrów za pomocą normy ℓ_2 malejących w czasie gradientów (jak pamiętasz, norma ℓ_2 jest pierwiastkiem kwadratowym z sumy kwadratów). Algorytm AdaMax (opisany w tym samym artykule co algorytm Adam) zastępuje normę ℓ_2 normą ℓ_∞ (jest to wyszukany sposób zapisania wartości maksymalnej). Mówiąc dokładniej, zastępujemy tu etap 2. z równania 11.8 wzorem $s \leftarrow \max(\beta_2 s, \nabla_{\theta} J(\theta))$, pomijamy etap 4., a w etapie 5. skalujemy aktualizacje gradientów przez czynnik s , czyli przez wartość maksymalną malejących gradientów. W praktyce rozwiązywanie to zwiększa stabilność optymalizatora AdaMax, ale jest to uzależnione od zestawu danych i zasadniczo algorytm Adam sprawuje się lepiej. Zatem jest to kolejny optymalizator, który warto wypróbować, jeżeli algorytm Adam będzie miał problemy z jakiś zadaniem.

Nadam

Optymalizator Nadam łączy w sobie algorytm Adam ze sztuczką Nesterova, zatem zawsze osiąga zbieżność nieco szybciej w stosunku do klasycznej wersji. Timothy Dozat w raporcie poświęconym tej metodzie (http://cs229.stanford.edu/proj2015/054_report.pdf)¹⁹ porównuje wiele różnych optymalizatorów w rozmaitych zadaniach i zauważa, że zasadniczo Nadam osiąga lepsze rezultaty niż Adam, ale czasami okazuje się gorszy od optymalizatora RMSProp.



Adaptacyjne metody optymalizacji (takie jak RMSProp, Adam i Nadam) często sprawdzają się znakomicie i szybko osiągają zbieżność z dobrym rozwiązaniem. Jednak Ashia C. Wilson i in. udowodnili w publikacji z 2017 roku (<https://arxiv.org/abs/1705.08292>)²⁰, że mogą prowadzić one do rozwiązań niezbyt radzących sobie z pewnymi zbiorami danych. Jeśli więc będziesz zawiedzioną/zawiedziony wydajnością modelu, spróbuj skorzystać ze zwykłego algorytmu Nesterova — Twój zestaw danych może po prostu mieć konflikt z adaptacyjną techniką gradientów. Obserwuj również współczesne osiągnięcia, gdyż dziedzina ta rozwija się bardzo szybko.

¹⁹ Timothy Dozat, *Incorporating Nesterov Momentum into Adam* (2016).

²⁰ Ashia C. Wilson i in., *The Marginal Value of Adaptive Gradient Methods in Machine Learning*, „Advances in Neural Information Processing Systems” 30 (2017), s. 4148 – 4158.

Wszystkie omówione do tej pory metody optymalizacji polegają wyłącznie na **pochodnych częstekowych pierwzego rzędu (jakobianach)**. W literaturze poświęconej zagadnieniom optymalizacji znajdziemy znakomite algorytmy bazujące na **pochodnych częstekowych drugiego rzędu (hesjanach)**. Niestety algorytmy te bardzo trudno zaimplementować w głębokich sieciach neuronowych, gdyż na każde wyjście przypada n^2 hesjanów (gdzie n stanowi liczbę parametrów), w przeciwieństwie do n jakobianów. Przeciętna głęboka sieć neuronowa ma dziesiątki tysięcy parametrów, dlatego algorytmy optymalizacyjne drugiego rzędu często nie mieścią się nawet w pamięci, a te, które się mieścią, są po prostu zbyt wolne.

Uczenie modeli rzadkich

Wszystkie zaprezentowane algorytmy optymalizacyjne generują modele gęste, co oznacza, że większość parametrów przyjmuje niezerowe wartości. Jeżeli potrzebujesz zdumiewająco szybkiego modelu lub masz za mało pamięci, lepszym rozwiązaniem byłoby uzyskanie modelu rzadkiego.

Jednym z najprostszych rozwiązań jest normalne wyuczenie modelu, a następnie usunięcie bardzo małych wag (wyznaczenie im wartości 0). Zwrót uwagę, że zazwyczaj nie prowadzi ono do bardzo rzadkiego modelu i może spowodować spadek wydajności.

Inną możliwością jest wprowadzenie silnej regularyzacji ℓ_1 na etapie uczenia (wykorzystamy ten sposób w dalszej części rozdziału), gdyż dąży ona do wyznaczania zer jak największej liczby wag (została ona omówiona w rozdziale 4. podczas analizy regresji typu Lasso).

Jeżeli techniki te okażą się niewystarczające, wypróbuj pakiet TensorFlow Model Optimization Toolkit (TF-MOT; https://www.tensorflow.org/model_optimization/) zawierający interfejs API umożliwiający obcinanie lub usuwanie połączeń w trakcie uczenia na podstawie ich wartości.

W tabeli 11.2 znajdziesz porównanie wszystkich omówionych do tej pory optymalizatorów (* to zły wynik, *** — dobry).

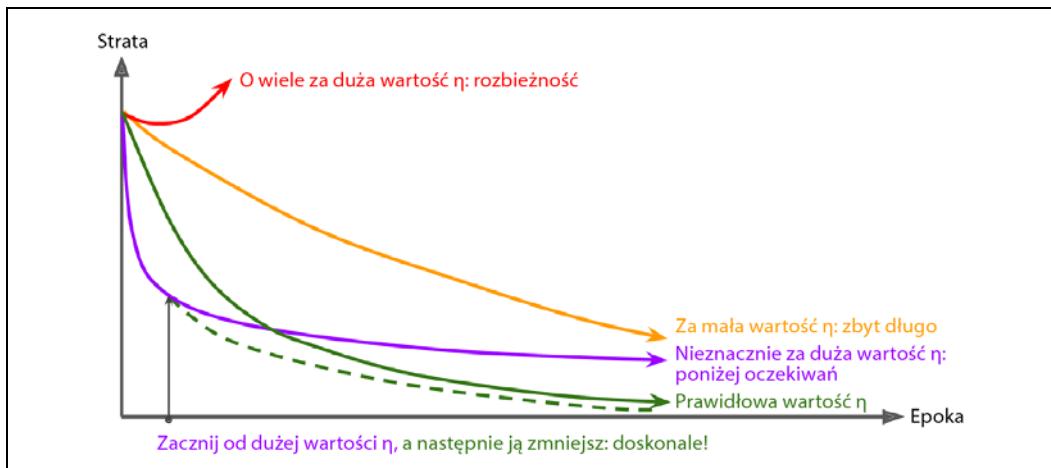
Tabela 11.2. Porównane optymalizatorów

Klasa	Szybkość uzyskiwania zbieżności	Stopień uzyskanej zbieżności
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (zatrzymuje się zbyt wcześnie)
RMSprop	***	** lub ***
Adam	***	** lub ***
Nadam	***	** lub ***
AdaMax	***	** lub ***

Harmonogramowanie współczynnika uczenia

Określenie odpowiedniej wartości współczynnika uczenia jest bardzo istotne. Jeżeli wyznaczymy ją za dużą, algorytm stanie się rozbieżny (wyjaśniłem to w rozdziale 4., omawiając metody gradientu prostego). Z kolei przy zbyt małej wartości algorytm uzyska w końcu zbieżność z optimum funkcji, ale

będzie potrzebował do tego mnóstwo czasu. Przy niewielkim zawyżeniu wartości współczynnik uczenia na początku będzie sobie radził bardzo dobrze, ale ostatecznie będzie ciągle przeskakiwał optimum funkcji i nigdy nie znajdzie jej minimum. Przy ograniczonych zasobach obliczeniowych trzeba czasami przerwać naukę przed osiągnięciem zbieżności, co spowoduje uzyskanie rezultatu poniżej oczekiwanych (rysunek 11.8).



Rysunek 11.8. Krzywe uczenia dla różnych wartości współczynnika uczenia η

Jak już wiesz z rozdziału 10., możesz znaleźć dobrą wartość współczynnika uczenia, jeżeli będziesz trenować model przez kilkaset iteracji — w każdym przebiegu wartość współczynnika jest zwiększana wykładniczo, od bardzo małych do bardzo dużych wartości, po czym przyglądamy się krzywej uczenia i wybieramy współczynnik uczenia w punkcie znajdującym się nieco przed miejscem, w którym krzywa uczenia zaczyna ponownie rosnąć. Możesz teraz ponownie zainicjalizować model i wyuczyć go z tak wyznaczonym współczynnikiem uczenia.

Istnieją jednak lepsze rozwiązania od stałej wartości współczynnika uczenia: jeżeli wyznaczysz na początku jego wysoką wartość, a następnie zmniejszysz, gdy zmalaє szybkość uczenia, możesz osiągnąć lepsze rezultaty niż pozostawiając stałą, optymalną wartość. Istnieje wiele strategii redukowania współczynnika uczenia podczas trenowania modelu. Nieraz również skuteczne okazuje się rozpoczęcie od małej wartości współczynnika uczenia, zwiększenie jej, a następnie ponowne zmniejszenie. Są to tzw. **harmonogramy uczenia** (ang. *learning schedules*; wspomniałem już o nich w rozdziale 4.). Najpopularniejsze z nich to:

Harmonogramowanie potęgowe (ang. *power scheduling*)

Wyznaczamy wartość współczynnika uczenia na podstawie funkcji liczby iteracji t : $\eta(t) = \eta_0 / (1 + t/s)^c$. Początkowy współczynnik uczenia η_0 , potęga c (zazwyczaj ma wyznaczoną wartość 1) i liczba skoków s są hiperparametrami. Wartość współczynnika uczenia maleje z każdym krokiem. Po s skoków jej wartość wynosi $\eta_0/2$. Po s kolejnych kroków osiąga wartość $\eta_0/3$, następnie $\eta_0/4$, $\eta_0/5$ itd. Jak widać, w tym harmonogramie współczynnik uczenia maleje najpierw szybko, a później zwalnia coraz bardziej. Oczywiście wymagane jest tutaj strojenie hiperparametrów η_0 i s (a czasami także c).

Harmonogramowanie wykładnicze (ang. *exponential scheduling*)

Wyznaczamy współczynnik uczenia: $\eta(t) = \eta_0(0,1)^{t/s}$. Jego wartość będzie stopniowo maleć dziesięciokrotnie co s skoków. W porównaniu do harmonogramowania potęgowego, gdzie współczynnik uczenia maleje szybko, ale stopniowo zwalnia ten proces, w harmonogramowaniu wykładniczym wartość ta stale maleje o rząd wielkości co s skoków.

Harmonogramowanie stałoprzedziałowe (ang. *piecewise constant scheduling*)

Wyznacz stały współczynnik uczenia na określoną liczbę epok (np. $\eta_0 = 0,1$ na 5 epok), następnie zmniejsz go na kolejne epoki (np. $\eta_1 = 0,001$ na 50 epok) itd. Takie rozwiązanie sprawdza się bardzo dobrze, trzeba jednak poświęcić czas na określenie właściwej sekwencji współczynników uczenia i czasu ich stosowania.

Harmonogramowanie wydajnościowe (ang. *performance scheduling*)

Mierzmy błąd walidacji co N przebiegów (zupełnie jak w metodzie wczesnego zatrzymywania) i w momencie, gdy wartość błędu przestaje maleć, redukujemy współczynnik uczenia o wartość λ .

Harmonogramowanie 1cycle (ang. *1cycle scheduling*)

W przeciwieństwie do pozostałych technik, w opisany w artykule Lesliego Smitha z 2018 roku (<https://arxiv.org/abs/1803.09820>)²¹ harmonogramowaniu 1cycle rozpoczynamy od zwiększania początkowego współczynnika uczenia η_0 i pozwalamy mu rosnąć liniowo do η_1 w pierwszej połowie uczenia. Następnie przez drugą połowę uczenia wartość ta maleje z powrotem do η_0 , a w kilku ostatnich epokach gwałtownie redukujemy ją o kilka rzędów wielkości (ale ciągle liniowo). Maksymalna wartość współczynnika uczenia η_1 zostaje dobrana tak samo jak w przypadku wyszukiwania optymalnego współczynnika uczenia, natomiast początkowa wartość uczenia η_0 powinna być mniej więcej dziesięciokrotnie mniejsza. Jeśli korzystasz z momentu, jego wartość na początku powinna być duża (np. 0,95), po czym powinna liniowo maleć w pierwszej połowie treningu (mniej więcej do wartości 0,85), a w drugiej połowie powinna wzrastać do wartości pierwotnej i utrzymać się przy niej przez kilka ostatnich epok. Smith przeprowadził wiele eksperymentów i udowodnił, że rozwiązanie to znaczco przyspiesza proces uczenia i pozwala osiągać lepszą wydajność. Przykładowo na popularnym zestawie obrazów CIFAR10 twórca był w stanie osiągać dokładność walidacji rzędu 91,9% w zaledwie 100 epok, podczas gdy przy użyciu standardowych technik udawało się uzyskiwać dokładność 90,3% po 800 epokach (w tej samej architekturze sieci).

Andrew Senior i in. w publikacji z 2013 roku (<https://static.googleusercontent.com/media/research.google.com/pl//pubs/archive/40808.pdf>)²² porównali wydajność kilku najpopularniejszych harmonogramów uczenia podczas trenowania głębokich sieci neuronowych, przeznaczonych do rozpoznawania mowy, za pomocą optymalizacji momentum. W podsumowaniu autorzy stwierdzili, że w takim środowisku zarówno harmonogramowanie wydajnościowe, jak i wykładnicze spisywały się dobrze, ale bardziej polecali to drugie z powodu łatwości implementacji, strojenia i nieco większej szybkości

²¹ Leslie N. Smith, *A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 — Learning Rate, Batch Size, Momentum, and Weight Decay*, arXiv preprint arXiv:1803.09820 (2018).

²² Andrew Senior i in., *An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition*, „Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing” (2013), s. 6724 – 6728.

w uzyskaniu zbieżności (wspomnieli także, że harmonogramowanie wykładnicze jest nieco łatwiejsze do zaimplementowania od harmonogramowania wydajnościowego, chociaż obecnie w interfejsie Keras wprowadzenie obydwu rozwiązań nie stanowi problemu). Okazuje się natomiast, że harmonogramowanie 1cycle sprawdza się jeszcze lepiej.

Implementacja harmonogramowania potęgowego w interfejsie Keras jest najłatwiejsza — wystarczy wyznaczyć parametr decay podczas tworzenia optymalizatora:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

Parametr decay jest odwrotnością s (liczby skoków wymaganych do podzielenia współczynnika uczenia przez jeszcze jedną jednostkę). Ponadto Keras zakłada, że wartość c jest równa 1.

Harmonogramowanie wykładnicze i stałopredziałowe są również całkiem proste do zaimplementowania. Najpierw musimy zdefiniować funkcję przyjmującą numer bieżącej epoki i zwracającą współczynnik uczenia. Zaimplementujmy na przykład harmonogramowanie wykładnicze:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

Jeżeli nie chcesz definiować na stałe wartości parametrów η_0 i s , możesz utworzyć funkcję zwracającą skonfigurowane parametry:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Następnie utworzymy wywołanie zwrotne LearningRateScheduler, podamy mu funkcję harmonogramowania i przekażemy je metodzie fit():

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

Na początku każdej epoki wywołanie zwrotne LearningRateScheduler zaktualizuje atrybut learning_rate optymalizatora. Zazwyczaj wystarczy taka częstotliwość aktualizacji, ale jeżeli chcesz, aby były one przeprowadzane częściej, np. co skok, możesz przygotować własne wywołanie zwrotne (zob. sekcję „Harmonogramowanie wykładnicze” w notatniku Jupyter). Aktualizacja co skok ma sens wtedy, gdy każda epoka składa się z wielu skoków. Ewentualnie możesz skorzystać z techniki keras.optimizers.schedules, do której powrócimy niebawem.

Dodatkowo funkcja harmonogramowania może przyjąć bieżącą wartość współczynnika uczenia jako drugi argument. Na przykład poniższa funkcja mnoży wcześniejszą wartość współczynnika uczenia przez $(0,1)^{1/20}$, co powoduje taki sam zanik wykładniczy jak wcześniej (różnica polega na tym, że zanik rozpoczyna się już od epoki 0., a nie 1.):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

Implementacja ta jest uzależniona od początkowej wartości współczynnika uczenia (w przeciwieństwie do poprzedniej implementacji), dlatego należy ją odpowiednio wyznaczyć.

Podczas zapisywania modelu zostają wraz z nim zapisane również jego optymalizator i współczynnik uczenia. Oznacza to, że w tej nowej funkcji harmonogramowania wystarczy wczytać wytrenowany model i kontynuować uczenie w przerwanym miejscu. Niestety przestaje to być takie proste w przypadku korzystania z argumentu epoch i jego wartość zawsze zostaje wyzerowana w momencie wywołania metody fit(). Gdyby kontynuować uczenie modelu od miejsca przerwania, doprowadziłoby to do bardzo dużych wartości współczynnika uczenia i prawdopodobnie zaszkodziłoby wagom modelu. Jednym z rozwiązań jest własnoręczne wyznaczenie argumentu initial_epoch w metodzie fit(), dzięki czemu parametr epoch będzie miał właściwą wartość.

Jeżeli chcesz zaimplementować harmonogramowanie stałoprzedziałowe, możesz skorzystać z takiej funkcji jak pokazana poniżej (podobnie jak wcześniej możesz zdefiniować także funkcję ogólniejszą; przykład znajdziesz w sekcji „Harmonogramowanie stałoprzedziałowe” w notatniku Jupyter), następnie przygotować wywołanie zwrotne LearningRateScheduler z tą funkcją i przekazać je metodzie fit() tak samo, jak robiliśmy to z harmonogramowaniem wykładowniczym:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

W harmonogramowaniu wydajnościowym wykorzystujemy wywołanie zwrotne ReduceLROnPlateau. Na przykład jeżeli przekażesz poniższe wywołanie zwrotne metodzie fit(), współczynnik uczenia zostanie pomnożony przez 0,5 wtedy, gdy najlepsza funkcja straty walidacji nie zostanie poprawiona przez pięć kolejnych epok (dostępne są także inne ustawienia; więcej szczegółów znajdziesz w dokumentacji):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Należy również wspomnieć o alternatywnej formie implementacji harmonogramowania współczynnika uczenia: możesz definiować współczynnik uczenia za pomocą jednego z harmonogramów dostępnych w pakiecie keras.optimizers.schedules, a następnie przekazać otrzymaną wartość do wolnemu optymalizatorowi. W tym przypadku współczynnik uczenia jest aktualizowany co skok, a nie co epokę. Przykładowo poniżej prezentuję taki sam harmonogram wykładowiczny jak uzyskany za pomocą zdefiniowanej wcześniej funkcji exponential_decay():

```
s = 20 * len(X_train) // 32 # liczba skoków w 20 epokach (rozmiar grupy = 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

Wszystko gra i buczy, a do tego podczas zapisywania modelu zapisywane są również współczynnik uczenia i harmonogram (włącznie z jego stanem). Technika ta jednak nie stanowi części interfejsu Keras — jest ona charakterystyczna dla pakietu tf.keras.

Jeśli chodzi o harmonogramowanie 1cycle, jego implementacja nie powinna sprawiać większych problemów — wystarczy utworzyć niestandardowe wywołanie zwrotne modyfikujące współczynnik uczenia w każdym przebiegu (możesz aktualizować wartość współczynnika uczenia za pomocą zmiany self.model.optimizer.lr). Dobry przykład znajdziesz w sekcji „Harmonogramowanie 1cycle” w notatniku Jupyter.

Podsumowując, harmonogramowanie wykładowicze, wydajnościowe i 1cycle mogą znacząco przyspieszyć osiąganie zbieżności przez model, dlatego warto je wypróbować!

Regularyzacja jako sposób zapobiegania przetrenowaniu

Mając cztery parametry, mogę dopasować słonia do danych, a mając pięć, mogę sprawić, żeby zaczął machać trąbą.

— John von Neumann, zacytowany przez Enrico Fermiego w periodyku „Nature” nr 427

Mając miliony parametrów, możesz dopasować całe zoo do danych. Głębokie sieci neuronowe zazwyczaj zawierają dziesiątki tysięcy, a nawet miliony parametrów. Przy takiej ich liczbie sieć ma nieprawidłowo dużą swobodę i może być trenowana wobec najróżniejszych rodzajów skomplikowanych zbiorów danych. Taka elastyczność oznacza jednak podatność na przetrenowanie. Potrzebujemy mechanizmu regularyzacji.

W rozdziale 10. omówiłem już jedną z najlepszych technik regularyzacji: wczesne zatrzymywane. Mimo tego, że technika normalizacji wsadowej została zaprojektowana z myślą o wyeliminowaniu problemu niestabilnych gradientów, działa również jako świetny regularyzator. W tym podrozdziale zaprezentuję niektóre z najpopularniejszych technik regularyzacji stosowanych w sieciach neuronowych: regularyzację ℓ_1 i ℓ_2 , porzucanie oraz regularyzację typu max-norm.

Regularyzacja ℓ_1 i ℓ_2

Tak jak w prostych modelach liniowych z rozdziału 4., regularyzacja ℓ_2 służy do ograniczania wag sieci neuronowych, natomiast regularyzacja ℓ_1 przydaje się do tworzenia modeli rzadkich (w których wiele wag ma wartość równą 0). Oto sposób zastosowania regularyzacji ℓ_2 o współczynniku 0,01 wobec wag połączeń w danej warstwie:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Funkcja `l2()` zwraca regularyzator, który będzie wywoływany w każdym skoku podczas uczenia i będzie obliczał funkcję straty regularyzacji. Wartość ta następnie będzie dodawana do końcowej funkcji straty. Jak łatwo się domyślić, możesz wykorzystać regularyzator ℓ_1 , jeżeli użyjesz metody `keras.regularizers.l1()`; jeżeli zależy Ci na wykorzystaniu obydwu typów regularyzatorów naraz, posłuż do tego metoda `keras.regularizers.l1_l2()` (pamiętaj, żeby określić współczynniki regularyzacji).

Zazwyczaj chcemy stosować ten sam typ regularyzatora we wszystkich warstwach sieci, podobnie jak w przypadku korzystania z tej samej funkcji aktywacji i strategii inicjalizowania w warstwach ukrytych, dlatego prawdopodobnie te same argumenty będą wielokrotnie powtarzane. Sprawia to, że kod staje się nieelegancki i podatny na błędy. Możemy tego uniknąć, jeśli wprowadzimy pętle. Innym rozwiązaniem okazuje się użycie funkcji `functools.partial()`, dzięki której jesteśmy w stanie utworzyć funkcję opakowującą (z pewnymi domyślnymi wartościami argumentów) każdy wywoływalny element:

```

from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])

```

Porzucanie

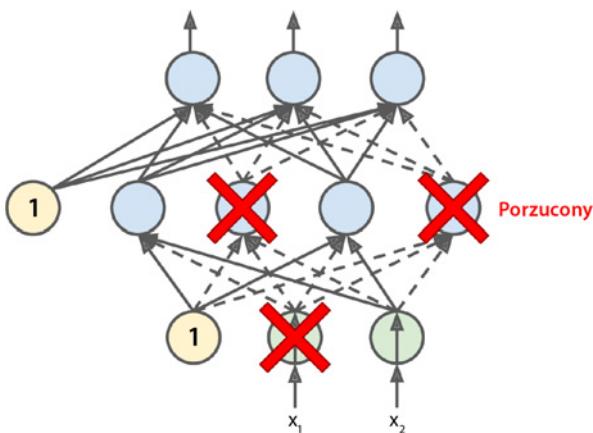
Prawdopodobnie najpopularniejszą techniką regularyzacji w głębszych sieciach neuronowych jest **porzucanie** (ang. *dropout*). Została ona zaproponowana (<https://arxiv.org/abs/1207.0580>)²³ w 2012 roku przez G. E. Hintona i rozwinięta w 2014 roku (<http://jmlr.org/papers/v15/srivastava14a.html>)²⁴ przez Nitisha Srivastavę i in. Udowodniono jej dużą skuteczność: po jej wprowadzeniu nawet do najnowocześniejszych sieci GSN ich dokładność wzrasta o 1 – 2%. Być może nie wydaje się to wiele, ale gdy pierwotny model osiąga dokładność rzędu 95%, dodatkowe 2% oznaczają zmniejszenie współczynnika błędu o niemal 40% (z błędu na poziomie 5% do 3%).

Algorytm ten nie jest skomplikowany: w poszczególnych przebiegach uczenia każdy neuron (oprócz warstwy wyjściowej) ma pewne prawdopodobieństwo p , że może zostać tymczasowo „porzucony”, tzn. całkowicie pominięty w procesie uczenia, ale w każdym przebiegu może znów być aktywny (rysunek 11.9). Hiperparametr p nosi nazwę **współczynnika porzucenia** (ang. *dropout rate*) i zazwyczaj ma przyporządkowaną wartość w zakresie od 10% do 50%; wartości rzędu 20 – 30% wykorzystywane są w sieciach rekurencyjnych (zob. rozdział 15.), natomiast zbliżone do 40 – 50% stosuje się w sieciach splotowych (zob. rozdział 14.). Po zakończeniu nauki neurony przestają być porzucane. I to wszystko (nie licząc szczegółów technicznych, którymi zajmiemy się za moment).

Na początku może wydawać się dość zaskakujące, że taka brutalna technika może okazać się skuteczna. Czy jakaś firma byłaby bardziej wydajna, gdyby kazano jej pracownikom każdego dnia rzucać monetą, aby określić, czy mają przyjść do pracy? Kto wie, może tak by było! Firma musiałaby oczywiście dostosować się do nowego rodzaju organizacji pracy; nie można byłoby polegać na żadnym pracowniku, że uzupełni automat do kawy albo zajmie się innymi krytycznymi zadaniami, dlatego obowiązki te powinny zostać rozzielone pomiędzy kilka osób. Pracownicy musieliby nauczyć się współpracować z wieloma osobami, a nie tylko kilkoma. Sama firma musiałaby stać się znacznie prężniejsza. Odejście jednej osoby nie mogłoby mieć na nią dużego wpływu. Nie jest jasne, jak taka koncepcja sprawdzałaby się w firmach, ale udowodniono, że znakomicie spisuje się w sieciach neuronowych. Neurony uczone

²³ Geoffrey E. Hinton i in., *Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors*, arXiv preprint arXiv:1207.0580 (2012).

²⁴ Nitish Srivastava i in., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, „Journal of Machine Learning Research” 15 (2014), s. 1929 – 1958.



Rysunek 11.9. Dzięki regularyzacji przez porzucanie w każdym przebiegu uczenia losowy podzbior neuronów w przynajmniej jednej warstwie (oprócz warstwy wyjściowej) jest „porzucony”; w danej iteracji neurony te wysyłają sygnał 0 (oznaczony przerywanymi strzałkami na rysunku)

poprzez porzucanie nie mogą uzależniać się od sąsiadów; muszą być we własnym zakresie jak najbardziej przydatne. Nie mogą również bezgranicznie polegać na kilku neuronach wejściowych; muszą zwracać baczną uwagę na swoje wejścia. W konsekwencji stają się mniej wrażliwe na drobne zmiany na wejściach. W ostatecznym rozrachunku uzyskujesz bardziej wszechstronną sieć, radzącą sobie lepiej z generalizowaniem wyników.

Mogimy również przeanalizować cudowne właściwości porzucania z innej perspektywy: zwróć uwagę, że w każdym przebiegu uczącym zostaje wygenerowana niepowtarzalna sieć neuronowa. Każdy neuron może stanowić jej część lub być nieobecny, dlatego istnieje 2^N możliwych kombinacji sieci (gdzie N stanowi całkowitą liczbę neuronów, które mogą zostać porzucone). To jest tak olbrzymia liczba, że w niektórych sieciach niemożliwe jest dwukrotne pojawienie się tej samej konfiguracji neuronów. Po 10 000 przebiegów okazuje się, że wytrenowaliśmy 10 000 różnych sieci neuronowych (za pomocą tylko jednej próbki uczącej). Oczywiście sieci te nie są od siebie całkowicie niezależne, ponieważ współdzielą wiele takich samych wag, mimo to każda z nich jest inna. Otrzymaną w ten sposób sieć możemy uznać za uśredniony zespół tych pomniejszych sieci neuronowych.



W praktyce możemy zazwyczaj wprowadzać metodę porzucania jedynie do neuronów znajdujących się maksymalnie w trzech szczytowych warstwach (nie licząc warstwy wyjściowej).

Przyjrzyjmy się pewnemu niewielkiemu, ale bardzo istotnemu szczegółowi technicznemu. Założymy, że $p = 50\%$, co oznacza, że w trakcie testowania dany neuron będzie podłączony do dwukrotnie większej liczby neuronów (średnio) niż podczas nauki. Aby zównoważyć to zjawisko, musimy przemnożyć wagi połączeń wejściowych każdego neuronu przez wartość 0,5 po zakończeniu treningu. Jeśli tego nie zrobimy, każdy neuron będzie otrzymywał w przybliżeniu dwukrotnie większą sumę sygnałów podczas testowania w porównaniu do nauki, co nie wróżyłoby zbyt dobrze jego wydajności. Mówiąc bardziej ogólnie, musimy pomnożyć każdą wagę połączenia wejściowego przez prawdopo-

dobieństwo utrzymywania (ang. *keep probability*; $1-p$) po zakończeniu treningu. Ewentualnie możemy podzielić wartość wyjścia każdego neuronu przez prawdopodobieństwo utrzymywania podczas nauki (obydwa te rozwiązania nie są równoważne, ale sprawdzają się równie dobrze).

Aby zaimplementować metodę porzucania za pomocą interfejsu Keras, wystarczy wprowadzić warstwę keras.layers.Dropout. W czasie uczenia funkcja ta losowo porzuca niektóre elementy (wyznacza im wartość 0), a pozostałe dzieli przez prawdopodobieństwo utrzymywania. Po zakończeniu trenowania staje się ona niepotrzebna i przepuszcza sygnały do następnej warstwy. Dzięki poniższemu listingu wprowadzamy warstwę porzucania przed każdą warstwą gęstą i wyznaczamy współczynnik porzucania równy 0,2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```



Porzucanie jest aktywne wyłącznie w fazie uczenia, dlatego porównywanie funkcji straty uczenia i walidacji może nie mieć sensu. Model może być przetrenowany na zestawie uczącym, a mimo to uzyskiwać podobne wartości obydwu funkcji straty. Z tego powodu nie zapomnij ocenić funkcji straty uczenia bez porzucania (tzn. po zakończeniu uczenia).

Jeśli zauważysz, że model ulega przetrenowaniu, możesz zwiększyć współczynnik porzucania. I odwrotnie, należy zmniejszyć jego wartość, jeśli model wykazuje oznaki niedotrenowania wobec zbioru uczącego. Pomocne okazuje się również zwiększenie tego współczynnika dla rozbudowanych warstw i zmniejszenie w niewielkich warstwach. Ponadto w wielu nowoczesnych strukturach sieci warstwa porzucania jest wprowadzana wyłącznie po ostatniej warstwie ukrytej, dlatego warto wypróbować to rozwiązanie, jeżeli pełna technika porzucania okaże się zbyt drastyczna.

Metoda porzucania zazwyczaj znacznie spowalnia proces konwergencji algorytmu, ale jednocześnie po właściwym dostrojeniu hiperparametrów pozwala uzyskać znacznie lepszy model. Zatem warto poświęcić na nią dodatkowy czas i wysiłek.



Jeżeli chcesz regularyzować sieć samonormalizującą opartą na funkcji aktywacji SELU (została omówiona wcześniej), skorzystaj z **porzucania alfa** (ang. *alpha dropout*): w tej odmianie porzucania zostają zachowane średnia i odchylenie standardowe danych wejściowych (rozwiązanie to zostało opisane w tym samym artykule co funkcja SELU, gdyż klasyczna metoda porzucania wpływa bardzo niekorzystnie na proces samonormalizacji).

Regularizacja typu Monte Carlo (MC)

W publikacji z 2016 roku (<https://homl.info/mcdropout>)²⁵ Yarin Gal i Zoubin Ghahramani podali kolejne powody, dla których warto stosować porzucanie:

- Po pierwsze, udowodniono wyraźny związek pomiędzy sieciami porzucania (tzn. sieciami neuronowymi zawierającymi warstwę Dropout przed każdą warstwą ukrytą) a aproksymowanym wnioskowaniem bayesowskim²⁶, dzięki czemu technika porzucania uzyskuje solidne uzasadnienie matematyczne.
- Po drugie, autorzy zaproponowali potężną technikę **porzucania MC**, która jest w stanie zwiększyć wydajność każdego wytrenowanego modelu porzucania bez konieczności jego ponownego uczenia, a nawet modyfikowania, zapewnia znacznie lepszy wskaźnik niepewności modelu, a także jest niezwykle prosta do zaimplementowania.

Jeżeli brzmi to zbyt pięknie, aby mogło być prawdziwe, to spójrz na poniższy przykład. Jest to pełna implementacja **porzucania MC**, zwiększająca skuteczność naszego wcześniejszego modelu porzucania bez konieczności jego ponownego uczenia:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Obliczamy jedynie 100 prognoz na zestawie testowym i wyznaczamy `training=True` po to, aby warstwa Dropout była aktywna, i gromadzimy uzyskane predykcje. Warstwa porzucania jest aktywna, dlatego każda prognoza będzie inna. Jak pamiętamy, funkcja `predict()` zwraca macierz, w której każdy rząd symbolizuje przykład, a każda kolumna klasę. Zestaw testowy składa się z 10 000 przykładów podzielonych na 10 klas, uzyskujemy w ten sposób macierz o rozmiarze [10000, 10]. Jeżeli zbierzymy 100 takich macierzy, `y_probas` stanie się tablicą o rozmiarze [100, 10000, 10]. Po uśrednieniu wyników w pierwszym wymiarze (`axis=0`) otrzymujemy `y_proba`, tablicę o rozmiarze [10000, 10], czyli taki sam rozmiar jak w przypadku pojedynczej prognozy. To wszystko! Uśrednianie wielu prognoz za pomocą porzucania stanowi oszacowanie Monte Carlo, które zazwyczaj daje pewniejszy wynik w porównaniu do rezultatu pojedynczej prognozy bez użycia porzucania. Spójrzmy przykładowo na wynik dla pierwszego obrazu z zestawu testowego Fashion MNIST (bez porzucania):

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99]], 
      dtype=float32)
```

Model wydaje się niemal pewny, że obraz należy do klasy 9. (trzewik). Czy należy mu ufać? Czy naprawdę występuje tu tak mały margines niepewności? Porównajmy ten rezultat z wynikami uzyskanyimi za pomocą techniki porzucania:

²⁵ Yarin Gal i Zoubin Ghahramani, *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*, „Proceedings of the 33rd International Conference on Machine Learning” (2016), s. 1050 – 1059.

²⁶ Mówiąc dokładniej, udowodnili oni, że uczenie sieci z porzucaniem jest matematycznie równoważne aproksymowaniu wnioskowaniu bayesowskiemu w szczególnym rodzaju modelu probabilistycznego, znany jako **głęboki proces gaussowski** (ang. *deep gaussian process*).

```
>>> np.round(y_probas[:, :1], 2)
array([[0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68],
       [[0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64]],
       [[0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]],
       [...]]
```

Tutaj sytuacja jest zupełnie inna. Po włączeniu porzucania model przestaje być już taki pewny rezultatów. Nadal największe prawdopodobieństwo ma klasa 9., ale brane pod uwagę są również klasa 5. (sandal) i 7. (tenisówka), co jest logiczne, gdyż wszystkie trzy klasy symbolizują obuwie. Jeżeli dokonamy uśrednienia w pierwszym wymiarze, uzyskamy następujące prognozy porzucania MC:

```
>>> np.round(y_proba[:, 1], 2)
array([[0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]],
      dtype=float32)
```

Model nadal uważa, że obraz przynależy do klasy 9., ale ma co do tego jedynie 62% pewności, co jest znacznie rozsądniejszym prawdopodobieństwem niż 99%. Przydaje się również wiedza, które inne klasy są brane pod uwagę. Możesz także przyjrzeć się odchyleniu standardowemu oszacowań prawdopodobieństwa (<https://xkcd.com/2110/>):

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:, 1], 2)
array([[0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]],
      dtype=float32)
```

Najwidoczniej występuje całkiem spora wariancja w oszacowaniach prawdopodobieństw: gdybyśmy tworzyli system uwzględniający ryzyko (np. medyczny lub finansowy), prawdopodobnie traktowalibyśmy taką niepewną prognozę z największą ostrożnością. Zdecydowanie nie należy traktować takiej predykcji jako pewnej na 99%. Co więcej, dokładność modelu wzrosła bardzo nieznacznie, z 86,8 do 86,9:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)
>>> accuracy
0.8694
```



Liczba stosowanych przykładów Monte Carlo (w naszym przypadku 100) jest hiperparametrem, który możemy stroić. Im jest ona większa, tym dokładniejsze są prognozy i oszacowania niepewności. Jednak po jej podwojeniu czas wnioskowania również wydłuży się dwukrotnie. Ponadto przekroczywszy pewną liczbę przykładów, wydajność modelu przestanie znacząco wzrastać. Twoim zadaniem jest więc znalezienie odpowiedniego kompromisu pomiędzy opóźnieniem a dokładnością, uzależnionego od zastosowania.

Jeżeli Twój model zawiera inne warstwy działające w specyficzny sposób podczas uczenia (np. warstwy BatchNormalization), to nie należy wymuszać fazy uczenia, możesz natomiast zastąpić warstwy Dropout następującą klasą MCDropout²⁷:

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

²⁷ Klasa MCDropout współdziała ze wszystkimi interfejsami Keras, także z interfejsem sekwencyjnym. Jeżeli zależy Ci wyłącznie na interfejsie funkcyjnym lub podklassowym, to nie musisz tworzyć klasy MCDropout — możesz utworzyć standardową warstwę Dropout i wyołać ją za pomocą `training=True`.

Umieszczamy tu warstwę Dropout w podklasie i tak przesłaniamy metodę `call()`, aby jej argument `training` był równy `True` (zob. rozdział 12.). Możemy w podobny sposób zdefiniować klasę `MCAalphaDropout` poprzez umieszczenie warstwy `AlphaDropout` w podklasie. Jeżeli budujesz model od podstaw, wystarczy zastąpić warstwę Dropout warstwą `MCDropout`. Jeżeli jednak dysponujesz modelem wytrenowanym za pomocą warstwy Dropout, musisz utworzyć nowy model, w którym jedyną różnicą jest użycie warstw `MCDropout` zamiast warstw `Dropout`, a następnie skopiować wagi ze starego do nowego modelu.

Krótko mówiąc, porzucanie MC jest doskonałą techniką udoskonalającą modele porzucania i gwarantującą lepsze oszacowania niepewności, a do tego działa jak zwykła technika porzucania w trakcie uczenia, dlatego spisuje się świetnie jako regularyzator.

Regularyzacja typu max-norm

Kolejną całkiem popularną techniką regularyzacji sieci neuronowych jest **regularyzacja typu max-norm** (ang. *max-norm regularization*): dla każdego neuronu wagi połączeń wejściowych w są ograniczane tak, że $\|w\|_2 \leq r$, gdzie r stanowi hiperparametr metody max-norm, a $\|\cdot\|_2$ jest członem regularyzacji ℓ_2 . W regularyzacji typu max-norm człon funkcji straty regularyzacji nie jest dodawany do całkowitej funkcji straty. Zazwyczaj implementujemy ją, wyliczając $\|w\|_2$ po każdym przebiegu uczenia i w razie potrzeby przeskalowując wartość w (

$$w \leftarrow w \frac{r}{\|w\|_2}$$
)

Zmniejszenie wartości r zwiększa stopień regularyzacji i pomaga zredukować poziom przetrenowania. Regularyzacja typu max-norm pomaga również zminimalizować problem zanikających/eksplodujących gradientów (jeśli nie korzystasz z normalizacji wsadowej).

Aby zaimplementować regularyzację max-norm w interfejsie Keras, wyznacz w każdej warstwie ukrytej argument `kernel_constraint` z ograniczeniem `max_norm()` i odpowiednią wartością maksymalną, na przykład:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                    kernel_constraint=keras.constraints.max_norm(1.))
```

Po każdej iteracji uczenia metoda `fit()` modelu wywoła obiekt zwracany przez funkcję `max_norm()` i przekaże mu wartości wag danej warstwy, a on obliczy z nich wartości przeskalowane, którymi następnie zostaną zastąpione wartości pierwotne. Jak się przekonasz w rozdziale 12., możesz zdefiniować niestandardową funkcję ograniczającą i wykorzystać ją jako `kernel_constraint`. Masz również możliwość ograniczenia członów obciążenia za pomocą argumentu `bias_constraint`.

Funkcja `max_norm()` zawiera argument `axis`, którego wartość domyślna to 0. Warstwa gęsta sieci zazwyczaj ma wagi o wymiarach [liczba wejść, liczba neuronów], zatem `axis=0` oznacza, że ograniczenie max-norm będzie stosowane niezależnie do wektora wag każdego neuronu. Jeżeli chcesz korzystać z techniki max-norm w warstwach splotowych (zob. rozdział 14.), należy we właściwy sposób wyznać argument `axis` (przeważnie `axis=[0, 1, 2]`).

Podsumowanie i praktyczne wskazówki

W tym rozdziale omówiliśmy szereg różnorodnych technik, dlatego możesz zastanawiać się, których z nich należy używać. Wszystko zależy od realizowanego zadania i nie ma jeszcze ustalonej, jednogłośnej zgodności, ale zauważylem, że konfiguracja podana w tabeli 11.3 w większości przypadków powinna sprawdzać się bardzo dobrze, bez konieczności znacznego strojenia hiperparametrów. Pamiętaj jednak, że wartości domyślne nie są nieodzowne!

Tabela 11.3. Domyślna konfiguracja głębokiej sieci neuronowej

Hiperparametr	Wartość domyślna
Inicjalizator jądra	Inicjalizacja He
Funkcja aktywacji	ELU
Normalizacja	Brak w sieci płytkiej; normalizacja wsadowa w sieci głębokiej
Regularyzacja	Wczesne zatrzymywanie (i w razie potrzeby regularyzacja ℓ_2)
Optymalizator	Optymalizacja momentum (ewentualnie RMSProp lub Nadam)
Harmonogram współczynnika uczenia	1cycle

Jeżeli dana sieć składa się z prostej struktury nałożonych na siebie warstw gęstych, może być ona poddawana samonormalizacji dzięki konfiguracji zaprezentowanej w tabeli 11.4.

Tabela 11.4. Konfiguracja samonormalizującej głębokiej sieci neuronowej

Hiperparametr	Wartość domyślna
Inicjalizator jądra	Inicjalizacja LeCuna
Funkcja aktywacji	SELU
Normalizacja	Brak (samonormalizacja)
Regularyzacja	W razie potrzeby porzucanie alfa
Optymalizator	Optymalizacja momentum (ewentualnie RMSProp lub Nadam)
Harmonogram współczynnika uczenia	1cycle

Nie zapomnij o normalizowaniu danych wejściowych! Weź także pod uwagę możliwość wykorzystania fragmentów uprzednio wytrenowanej sieci, przeznaczonej do rozwiązywania podobnego problemu lub skorzystania z nienadzorowanego uczenia wstępnego w przypadku dysponowania znaczną ilością danych nieoznakowanych, ewentualnie zastosowania uczenia wstępnego za pomocą zadania dodatkowego, jeśli masz dużą ilość danych oznakowanych przeznaczonych do podobnego zadania.

Powyższe wytyczne powinny wystarczyć w większości przypadków, nie należy jednak zapominać o pewnych wyjątkach:

- Jeżeli korzystasz z modelu rzadkiego, możesz zastosować regularyzację ℓ_1 (i ewentualnie wyzerować wagi o małych wartościach po zakończeniu uczenia). Jeżeli potrzebujesz modelu jeszcze rzadszego, użyj pakietu TensorFlow Model Optimization Toolkit. Oznacza to rezygnację z samonormalizacji, dlatego należy w takiej sytuacji skorzystać z konfiguracji domyślnej.

- Jeżeli potrzebujesz modelu o małym opóźnieniu (błyskawicznie uzyskującego prognozy), może być wymagana mniejsza liczba warstw, umieszczenie warstw normalizacji wsadowej w warstwach wcześniejszych i, być może, użycie szybszej funkcji aktywacji, takiej jak standardowa lub przeciekająca funkcja ReLU. Pomaga również wprowadzenie modelu rzadkiego. Ostatecznie możesz także zmniejszyć precyzyję danych zmiennoprzecinkowych z 32 na 16, a nawet 8 bitów (zob. podrozdział „Wdrażanie modelu na urządzeniu mobilnym lub wbudowanym” w rozdziale 19.). Również w tym przypadku warto skorzystać z pakietu TF-MOT.
- Jeżeli tworzysz aplikację uwzględniającą ryzyko lub opóźnienie wnioskowania nie stanowi priorytetu, możesz poprawić wydajność i uzyskać rzetelniejsze oszacowania prawdopodobieństwa i niepewności za pomocą porzucania MC.

Masz już wiedzę wystarczającą do rozpoczęcia uczenia bardzo głębokich sieci neuronowych! Mam nadzieję, że udało mi się Cię przekonać do przydatności interfejsu Keras. Czasami może być potrzebna jednak jeszcze większa kontrola: być może będziesz musiał napisać niestandardową funkcję straty lub zmodyfikować algorytm uczenia. W takich sytuacjach musisz skorzystać z ogólnego API modułu TensorFlow, któremu został poświęcony następny rozdział.

Ćwiczenia

1. Czy dobrym rozwiązaniem jest inicjalizowanie wszystkich wag z taką samą wartością, jeśli jest ona losowo wyznaczana za pomocą inicjalizacji He?
2. Czy inicjalizowanie członów obciążen z wartością 0 jest poprawne?
3. Wymień trzy zalety funkcji aktywacji SELU w porównaniu do funkcji ReLU.
4. W jakich sytuacjach należy stosować następujące funkcje aktywacji: SELU, przeciekająca ReLU (i jej odmiany), ReLU, tangensa hiperbolicznego, logistyczna i softmax?
5. Co się może stać, jeśli wyznaczysz wartość hiperparametru momentum zbyt bliską 1 (np. 0,99999) podczas korzystania z optymalizatora SGD?
6. Wymień trzy sposoby uzyskiwania modelu rzadkiego.
7. Czy metoda porzucania spowalnia proces uczenia? Czy spowalnia ona proces wnioskowania (tj. wyliczania prognoz dla nowych przykładów)? Jak to wygląda w przypadku porzucania MC?
8. Potrenuj uczenie głębokiej sieci neuronowej na zestawie obrazów CIFAR10:
 - a. Utwórz sieć głęboką składającą się z 20 warstw ukrytych zawierających po 100 neuronów (jest ich za dużo, ale taki jest morał tego ćwiczenia). Skorzystaj z inicjalizacji He i funkcji aktywacji ELU.
 - b. Wprowadź optymalizację Nadam i wczesne zatrzymywanie, po czym wyucz sieć na zestawie danych CIFAR10. Możesz go wczytać za pomocą funkcji keras.datasets.cifar10.load_data(). Zestaw ten składa się z 60 000 kolorowych obrazów (50 000 przykładów uczących i 10 000 testowych) o rozmiarach 32×32 , tworzących 10 klas, co oznacza, że musisz w warstwie wyjściowej wprowadzić funkcję softmax i 10 neuronów. Pamiętaj, aby po każdej zmianie struktury lub hiperparametrów modelu poszukać właściwej wartości współczynnika uczenia.

- c. Spróbuj teraz dodać normalizację wsadową i porównać współczynniki uczenia. Czy model uzyskuje teraz szybciej zbieżność? Czy otrzymany model jest lepszy? Co się dzieje z szybkością uczenia?
- d. Zastęp teraz normalizację wsadową funkcją SELU i wprowadź wszelkie modyfikacje wymagane do spełnienia wymogu samonormalizacji (tzn. standaryzuj cechy wejściowe, skorzystaj z klasycznej inicjalizacji LeCuna, upewnij się, że sieć głęboka będzie zawierała wyłącznie sekwencję warstw gęstych, itd.).
- e. Spróbuj przeprowadzić regularyzację modelu za pomocą porzucania alfa. Następnie sprawdź, czy możesz uzyskać lepszą wydajność za pomocą porzucania MC bez trenowania modelu.
- f. Wytrenuj ponownie model za pomocą harmonogramowania 1cycle i sprawdź, jaki ma ono wpływ na szybkość uczenia i dokładność modelu.

Rozwiązania tych ćwiczeń znajdziesz w dodatku A.

Modele niestandardowe i uczenie za pomocą modułu TensorFlow

Do tej pory korzystaliśmy wyłącznie z wyspecjalizowanego API tf.keras, stanowiącego część modułu TensorFlow, ale mimo to osiągnęliśmy dzięki niemu już bardzo dużo: stworzyliśmy zróżnicowane struktury sieci neuronowych, w tym takie jak sieci regresyjne i klasyfikacyjne, sieci Wide & Deep i sieci samonormalizujące. Korzystaliśmy przy tym z przeróżnych technik, np. normalizacji wsadowej, porzucania i harmonogramów współczynnika uczenia. Istotnie, w 95% przypadków, z jakimi będziesz mieć do czynienia, powiniens w zupełności wystarczyć interfejs tf.keras (a czasami także tf.data; zob. rozdział 13.). Nadszedł jednak czas zagłębić się dokładniej w strukturę modułu TensorFlow i przyjrzeć się jego ogólnemu API Pythona (https://www.tensorflow.org/versions/r2.0/api_docs/python/tf). Przydaje się on w sytuacji, gdy trzeba pisać niestandardowe funkcje straty, wskaźniki, warstwy, modele, inicjalizatory, regularizatory, ograniczenia wag itd. Być może będziesz musiał/musiał nawet w pełni kontrolować samą pętlę uczenia po to, aby stosować specjalne przekształcenia lub ograniczenia gradientów (wykraczające poza ich obcinanie) albo wykorzystywać wiele optymalizatorów w odrębnych częściach sieci. W tym rozdziale zajmiemy się wszystkimi wspomianymi aspektami, a także omówię, w jaki sposób można usprawniać modele niestandardowe i algorytmy uczenia za pomocą funkcji automatycznego generowania grafów w TensorFlow. Najpierw jednak przedstawię ogólną strukturę modułu TensorFlow.



Moduł TensorFlow 2.0 (w wersji beta) został opublikowany w czerwcu 2019 roku, dzięki czemu stał się znacznie przystępniejszy w użytkowaniu. W pierwszym wydaniu książki korzystaliśmy z modułu TF1, obecnie zaś przechodzimy na wersję TF2.

Krótkie omówienie modułu TensorFlow

Jak już wiesz, TensorFlow to potężna biblioteka przeznaczona do obliczeń numerycznych, dostosowana zwłaszcza do wielkoskalowego uczenia maszynowego (może być jednak używana również w innych zastosowaniach wiążących się z zaawansowanymi obliczeniami). Została zaprojektowana przez zespół Google Brain i obecnie stanowi trzon wielu rozbudowanych usług firmy Google, takich jak Google Cloud Speech, Zdjęcia Google czy Google Search. W listopadzie 2015 roku udostępniono publicznie jej kod źródłowy i od tamtej pory stała się najpopularniejszą biblioteką uczenia głębokiego (pod względem nawiązań do niej w publikacjach naukowych, wykorzystania

w firmach, liczby gwiazdek w repozytorium GitHub itd.). Jest ona stosowana w niezliczonych projektach uczenia maszynowego, takich jak klasyfikowanie obrazów, przetwarzanie języka naturalnego, systemy rekomendacji produktów czy prognozowanie szeregów czasowych.

Cóż takiego więc oferuje moduł TensorFlow? Oto małe podsumowanie:

- Jego rdzeń jest bardzo podobny do biblioteki NumPy, zawiera jednak dodatkowo obsługę procesorów graficznych.
- Obsługuje obliczenia rozproszone (wykorzystujące wiele urządzeń i serwerów).
- Zawiera kompilator JIT (ang. *just-in-time*) umożliwiający optymalizowanie obliczeń pod kątem szybkości i zużycia pamięci. Jego mechanizm działania polega na wydobyciu grafu obliczeniowego z funkcji Pythona, jego zoptymalizowaniu (np. poprzez wycięcie nieużywanych węzłów) i wydajnej realizacji (np. poprzez automatyczne zrównoleglenie wykonywania niezależnych operacji).
- Grafy obliczeniowe mogą być zapisywane w formacie przenośnym, dzięki czemu model TensorFlow może być uczony w jednym środowisku (np. w środowisku Python na Linuksie), a uruchamiany w innym (np. za pomocą środowiska Java na urządzeniu wyposażonym w system Android).
- Implementuje mechanizm różniczkowania automatycznego (zob. rozdział 10. i dodatek D), a także pewne znakomite optymalizatory, np. RMSProp czy Nadam (zob. rozdział 11.), dzięki czemu możesz minimalizować wszystkie typy funkcji straty.

TensorFlow zawiera wiele dodatkowych funkcji, a najważniejszą z nich jest oczywiście interfejs `tf.keras`¹, dostępne są jednak również operacje wczytywania i przetwarzania wstępnego danych (`tf.data`, `tf.io` itd.), przetwarzania obrazów (`tf.image`), przetwarzania sygnałów (`tf.signal`) i inne (znajdziesz je na rysunku 12.1).

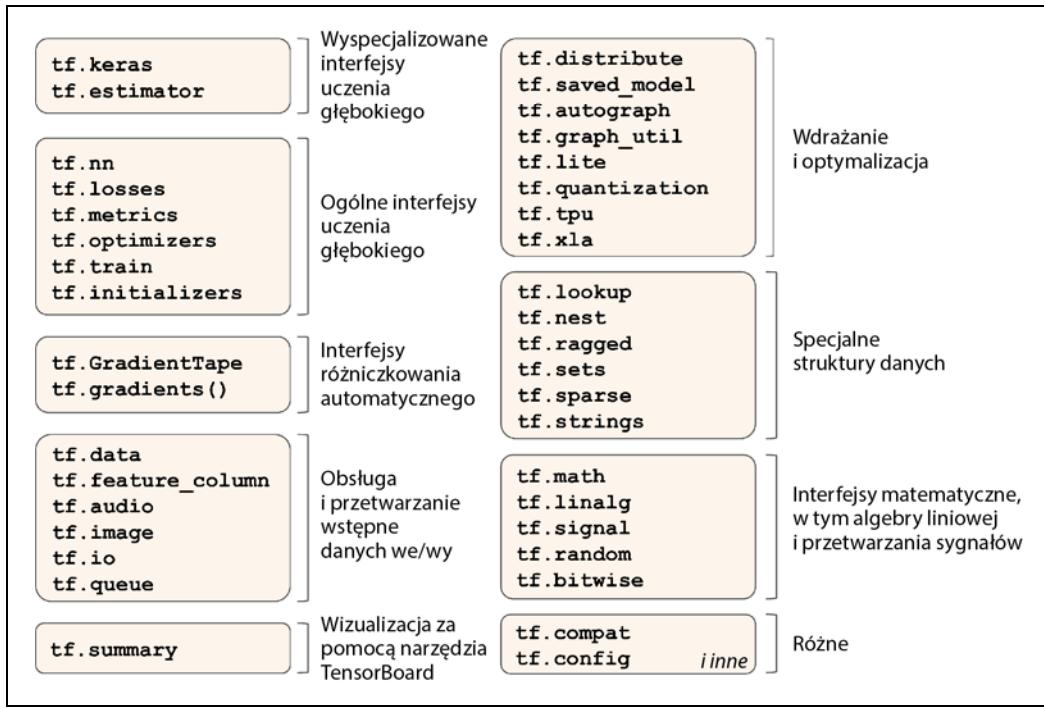


Zajmiemy się wieloma pakietami i funkcjami modułu TensorFlow, niemożliwe jest jednak opisanie ich wszystkich, dlatego poświęć czas na samodzielne zapoznanie się z dostępnymi API, a przekonasz się, że TensorFlow jest rozbudowany i zawiera bogatą dokumentację.

Na najbardziej podstawowym poziomie każda operacja TensorFlow (w piśmiennictwie anglojęzycznym stosowany jest skrót **op**) jest implementowana za pomocą wydajnego kodu C++². Wiele implementacji istnieje w różnych odmianach zwanych **jądrami** (ang. *kernels*): każde jądro jest wyspecjalizowane pod kątem określonego typu urządzenia, np. procesora głównego (CPU), procesora graficznego (GPU) czy nawet **procesora tensorowego** (ang. *Tensor Processing Unit* — TPU). Być może wiesz już, że procesory graficzne mogą znaczco przyspieszyć obliczenia poprzez rozdzielenie ich na wiele mniejszych podoperacji i ich równoległe przetwarzanie w licznych wątkach GPU.

¹ Znajdziemy tu także inny interfejs uczenia głębokiego, o nazwie **Estimators API**, ale sam zespół twórców TensorFlow zaleca korzystanie z `tf.keras`.

² W przypadku konieczności stworzenia własnej operacji (co raczej nie zdarza się nigdy) możesz to zrobić za pomocą interfejsu C++.



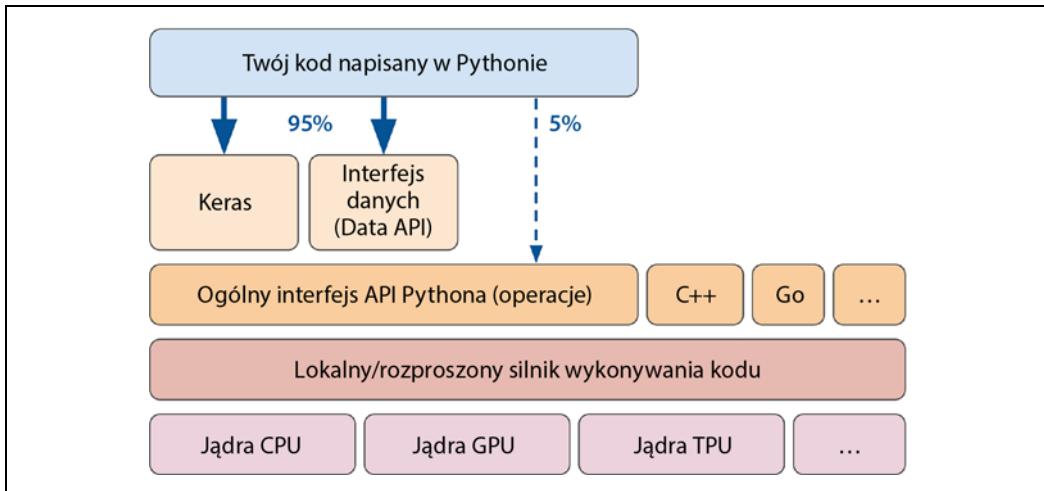
Rysunek 12.1. API modułu TensorFlow

Procesory tensorowe są jeszcze szyszki: mamy tu do czynienia z niestandardowymi układami ASIC stworzonymi z myślą o operacjach uczenia głębokiego³ (rozdział 19. został poświęcony omówieniu działania modułu TensorFlow na procesorach GPU i TPU).

Architektura modułu TensorFlow została zaprezentowana na rysunku 12.2. Przez większość czasu kod będzie wykorzystywał wyspecjalizowane API (zwłaszcza `tf.keras` i `tf.data`), jeśli jednak będziesz potrzebować większej swobody, możesz przejść do ogólnego API i bezpośrednio przetwarzać tensory. Zwróć uwagę, że dostępne są również interfejsy przeznaczone dla innych języków. W każdym razie silnik TensorFlow zajmie się wydajnym realizowaniem operacji, nawet pomiędzy wieloma urządzeniami, jeżeli taka będzie Twoja wola.

TensorFlow działa nie tylko na najważniejszych systemach operacyjnych (Windows, Linux, macOS), lecz także na urządzeniach mobilnych (dzięki interfejsowi **TensorFlow Lite**), zarówno wyposażonych w system iOS, jak i z systemem Android (zob. rozdział 19.). Jeżeli nie chcesz używać interfejsu Pythona, masz do dyspozycji także API C++, Javy, Go i Swifta. Dostępna jest nawet implementacja JavaScriptu, **TensorFlow.js**, dzięki której jesteś w stanie uruchamiać modele bezpośrednio z poziomu przeglądarki.

³ Więcej informacji na temat procesorów tensorowych i mechanizmu ich działania znajdziesz pod adresem <https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning>.



Rysunek 12.2. Architektura modułu TensorFlow

TensorFlow to jednak nie tylko biblioteka. Moduł ten stanowi centrum rozbudowanego środowiska bibliotek. Do wizualizowania modeli i ich danych statystycznych służy narzędzie TensorBoard (zob. rozdział 10.). Dostępny jest również pakiet TensorFlow Extended (TFX; <https://www.tensorflow.org/tfx>), czyli zbiór bibliotek stworzonych przez firmę Google z myślą o łatwiejszym wdrażaniu modeli TensorFlow do etapu produkcyjnego — znajdziesz tu narzędzia walidacji danych, przetwarzania wstępnego, analizy modeli i eksploatacji (za pomocą biblioteki TF Serving; zob. rozdział 19.). Biblioteka **TensorFlow Hub** firmy Google pozwala na łatwe pobieranie i wielokrotne użytkowanie gotowych sieci neuronowych. Wiele struktur sieci neuronowych, z których część została wytrenowana, znajdziesz w ogrodzie modeli TensorFlow (<https://github.com/tensorflow/models/>). Więcej projektów opartych na module TensorFlow jest dostępnych w serwisie TensorFlow Resources (<https://www.tensorflow.org/resources/models-datasets>) i pod adresem <https://github.com/jtoy/awesome-tensorflow>. Repozytorium GitHub przechowuje setki projektów TensorFlow, dlatego dość łatwo można znaleźć model zaprojektowany z myślą o podobnym zadaniu, jakie Ty masz wyznaczone.



Codziennie pojawiają się nowe publikacje poświęcone uczeniu maszynowemu (wraz z implementacjami, a czasami nawet wyuczonymi modelami). Wyszukasz je z łatwością na stronie <https://paperswithcode.com/>.

Pamiętaj także, że z modułem TensorFlow wiąże się wyspecjalizowany zespół zapaleńców i pomocnych programistów, jak również duża społeczność zajmująca się usprawnianiem tej biblioteki. Aby zadać pytanie natury technicznej, odwiedź serwis <https://stackoverflow.com/> i oznacz pytanie słowami kluczowymi *tensorflow* i *python*. Możesz informować o błędach i proponować nowe funkcje poprzez repozytorium GitHub (<https://github.com/tensorflow/tensorflow>). Jeżeli interesują Cię dyskusje ogólne, dołącz do grupy Google (<https://groups.google.com/a/tensorflow.org/forum/#!overview>).

No dobrze, przejdźmy do pisania kodu!

Korzystanie z modułu TensorFlow jak z biblioteki NumPy

Interfejs TensorFlow bazuje na koncepcji **tensorów** przesyłanych pomiędzy poszczególnymi opercjami (stąd nazwa **TensorFlow**). Struktura tensora bardzo przypomina obiekt ndarray zdefiniowany w bibliotece NumPy: zazwyczaj przyjmuje postać wielowymiarowej tablicy, może jednak przechowywać również wartość skalarną (zwykłą liczbę, taką jak 42). Tensory będą pełnić ważną funkcję podczas tworzenia niestandardowych funkcji kosztu, wskaźników, warstw itd., dlatego nauczmy się najpierw je tworzyć i modyfikować.

Tensorы i operacje

Tensory tworzymy za pomocą funkcji `tf.constant()`. Przykładowo poniżej przedstawiam reprezentację tensorową macierzy składającej się z dwóch rzędów i trzech kolumn wypełnionych wartościami zmiennoprzecinkowymi:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # Macierz
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42) # Skalar
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

Podobnie jak obiekt ndarray, `tf.Tensor` cechuje się wymiarami i typem danych (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Indeksowanie wygląda podobnie jak w bibliotece NumPy:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Najważniejsze jednak, że dostępne są wszystkie rodzaje operacji tensorowych:

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Zwróć uwagę, że zapis `t + 10` jest równoważny wywołaniu funkcji `tf.add(t, 10)` (rzeczywiście, Python wywołuje metodę magiczną `t.__add__(10)`, która wywołuje po prostu `tf.add(t, 10)`). Obsługiwane są również inne operatory, takie jak – czy *. Operator @ został dodany w środowisku Python 3.5 i oznacza iloczyn macierzowy; jest on równoważny wywołaniu funkcji `tf.matmul()`.

Dostępne są tu wszystkie podstawowe operacje matematyczne (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()` itd.), a także większość operacji występujących w bibliotece NumPy (np. `tf.reshape()`, `tf.squeeze()`, `tf.tile()`). Niektóre funkcje mają nieco zmienione nazwy w stosunku do ich odpowiedników NumPy; przykładowo funkcje `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()` i `tf.math.log()` stanowią odpowiedniki funkcji `np.mean()`, `np.sum()`, `np.max()` i `np.log()`. Często za zmianą nazwy kryje się dobry powód. Na przykład w module TensorFlow musisz zapisać `tf.transpose(t)` — w przeciwnieństwie do składni NumPy nie możesz napisać po prostu `t.T`. Wynika to z faktu, że działanie funkcji `tf.transpose()` jest nieco odmienne od atrybutu `T` z modułu NumPy: w przypadku biblioteki TensorFlow tworzony jest nowy tensor wraz z kopią transponowanych danych, natomiast w przypadku NumPy atrybut `t.T` stanowi jedynie transponowany widok tych samych danych. Podobnie sytuacja wygląda z operacją `tf.reduce_sum()`: zmiana nazwy spowodowana jest tym, że jądro GPU (tzn. implementacja przeznaczona dla kart graficznych) wykorzystuje algorytm redukujący, który nie gwarantuje utrzymania kolejności dodawania elementów; 32-bitowe wartości zmiennoprzecinkowe mają ograniczoną precyzję, dlatego za każdym razem możemy otrzymywać nieznacznie odmienny rezultat. To samo dotyczy funkcji `tf.reduce_mean()` (ale oczywiście operacja `tf.reduce_max()` jest deterministyczna).



Wiele funkcji i klas zawiera zdefiniowane aliasy. Na przykład pod nazwami `tf.add()` i `tf.math.add()` kryje się ta sama funkcja. Dzięki temu w module TensorFlow mogą występować zwięzłe nazwy najczęściej stosowanych operacji⁴ przy zachowaniu uporządkowanej struktury pakietów.

Interfejs ogólny Keras

Keras zawiera własny, ogólny interfejs API, umieszczony w module `keras.backend`. Zawiera on takie funkcje jak `square()`, `exp()` i `sqrt()`. W interfejsie `tf.keras` funkcje te zazwyczaj wywołują jedynie odpowiednie operacje TensorFlow. Jeżeli chcesz napisać kod, który można przenieść na inne implementacje Keras, korzystaj z tych funkcji. Odzwierciedlają one jednak tylko niewielki podzbior wszystkich funkcji dostępnych w module TensorFlow, dlatego w tej książce będziemy korzystać bezpośrednio z operacji TensorFlow. Oto prosty przykład użycia modułu `keras.backend`, który często w skrócie jest nazywany po prostu `K`:

```
>>> from tensorflow import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

⁴ Wartym wspomnienia wyjątkiem jest funkcja `tf.math.log()`, która jest jedną z częściej używanych, ale nie istnieje jej alias `tf.log()` (ponieważ byłaby prawdopodobnie myloną z funkcją tworzenia dzienników zdarzeń).

Tensory a biblioteka NumPy

Tensory dobrze współpracują z biblioteką NumPy: możesz stworzyć tensor z tablicy NumPy i odwrotnie. Możesz nawet stosować operacje TensorFlow na tablicach NumPy i operacje NumPy na tensorach:

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
>>> t.numpy() # Lub np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```



Zwróc uwagę, że biblioteka NumPy wykorzystuje domyślnie 64-bitową precyzję, natomiast w TensorFlow jest to precyzja 32-bitowa. Wynika to stąd, że precyzja 32-bitowa zazwyczaj zupełnie wystarczy w sieciach neuronowych, a do tego umożliwia szybsze działanie modeli i zajmuje mniej miejsca w pamięci operacyjnej. Z tego powodu, jeśli tworzysz tensor z tablicy NumPy, nie zapomnij wyznaczyć `dtype=tf.float32`.

Konwersje typów

Konwersje typów mogą znaczco pogorszyć wydajność, a jeżeli są przeprowadzane automatycznie, bardzo łatwo je przeoczyć. Aby uniknąć tego problemu, moduł TensorFlow nie przeprowadza automatycznie żadnej konwersji typu — w przypadku próby zrealizowania jakiejś operacji na tensorach o nieprawidłowych typach zostaje wyświetlony komunikat o wyjątku. Na przykład nie możesz zsumować tensora zmiennoprzecinkowego ze stałoprzecinkowym, a nawet tensora zmiennoprzecinkowego o precyzyji 32-bitowej z tensorem zmiennoprzecinkowym o precyzyji 64-bitowej:

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...]InvalidArgumentError[...]expected to be a float[...]
>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...]InvalidArgumentError[...]expected to be a double[...]
```

Początkowo może to być nieco irytujące, pamiętaj jednak, że rozwiązanie to zostało zaprojektowane z myślą o Tobie! Możesz oczywiście użyć funkcji `td.cast()`, jeżeli koniecznie musisz dokonać konwersji typów:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

Zmienne

Dotychczas omawiane wartości `tf.Tensor` są niezmienne — nie możemy ich modyfikować. Oznacza to, że za pomocą standardowych tensorów nie jesteśmy w stanie implementować wag sieci neuronowej, ponieważ muszą być one dostrajane przez algorytm propagacji wstecznej. Ponadto

inne parametry również mogą być zmieniane wraz z upływem czasu (np. optymalizator momentum śledzi wcześniejsze gradienty). Potrzebujemy więc wartości `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])  
>>> v  
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>
```

Wartość `tf.Variable` ma wiele elementów wspólnych z obiektem `tf.Tensor`: możesz przeprowadzać na niej te same operacje, dobrze współdziała z biblioteką NumPy i jest równie wybredna w kwestii typów danych. Możemy ją jednak modyfikować za pomocą metody `assign()` (ewentualnie za pomocą metod `assign_add()` lub `assign_sub()`, które, odpowiednio, zwiększą i zmniejszą zmiennej o wyznaczoną wartość). Możesz także modyfikować poszczególne komórki (fragmenty) przy użyciu metody `assign()` komórki (fragmentu) (nie zadziała bezpośrednio przypisywanie elementu) albo po zastosowaniu metod `scatter_update()` lub `scatter_nd_update()`:

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]  
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]  
v[:, 2].assign([0., 1.])  # => [[2., 42., 0.], [8., 10., 1.]]  
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])  
                         # => [[100., 42., 0.], [8., 10., 200.]]
```



W praktyce rzadko kiedy istnieje potrzeba ręcznego tworzenia zmiennych, ponieważ, jak się przekonasz, Keras zawiera metodę `add_weight()`, która wykoną tę operację za nas. Co więcej, parametry modelu będą zazwyczaj aktualizowane bezpośrednio przez optymalizatory, dlatego raczej nie będzie potrzeby ręcznego aktualizowania zmiennych.

Inne struktury danych

TensorFlow obsługuje kilka innych struktur danych, w tym następujące (szczegóły znajdziesz w sekcji „Tensory i operacje” w notatniku Jupyter lub w dodatku F):

Tensory rzadkie (`tf.SparseTensor`)

Skutecznie reprezentują tensory składające się głównie z zer. Pakiet `tf.sparse` zawiera operacje na tensorach rzadkich.

Tablice tensorów (`tf.TensorArray`)

Są to listy tensorów. Domyślnie mają stały rozmiar, ale można je przekształcić w postać dynamiczną. Wszystkie zawarte w nich tensory muszą mieć takie same wymiary i typ danych.

Tensory nierówne (`tf.RaggedTensor`)

Reprezentują statyczną listę list tensorów, w której każdy tensor ma takie same wymiary i typ danych. Operacje na tensorach nierównych mieszą się w pakiecie `tf.ragged`.

Tensory znakowe

Są to standardowe tensory o typie `tf.string`. Reprezentują one bajtowe łańcuchy znaków, nie znaki Unicode, dlatego jeśli stworzysz tensor znakowy za pomocą łańcucha znaków Unicode (np. standardowy łańcuch znaków Pythona 3, taki jak "café"), to zostanie automatycznie za-

kodowany w formacie UTF-8 (b"caf\xc3\xa9"). Możesz ewentualnie reprezentować łańcuchy znaków Unicode za pomocą tensorów typu `tf.int32`, gdzie każdy element symbolizuje punkt kodowy Unicode (np. [99, 97, 102, 233]). Pakiet `tf.strings` (z literą „s” na końcu) zawiera operacje na znakach bajtowych i łańcuchach Unicode (a także narzędzia do ich wzajemnej konwersji). Pamiętaj, że typ `tf.string` jest atomowy, co oznacza, że jego długość nie jest pokazywana w wymiarach tensora. Po jego przekształceniu w tensor Unicode (tzn. tensor typu `tf.int32` przechowujący punkty kodowe Unicode) długość typu zostaje wyświetlona w wymiarach tensora.

Zbiory

Są reprezentowane jako tensoły standardowe (lub rzadkie). Na przykład `tf.constant ([[1, 2], [3, 4]])` reprezentuje dwa zbiory: {1, 2} i {3, 4}. Mówiąc ogólnie, każdy zbiór jest reprezentowany przez wektor umieszczony w ostatniej osi tensora. Możesz przeprowadzać działania na zbiorach za pomocą operacji zawartych w pakiecie `tf.sets`.

Kolejki

Przechowuje tensoły na różnych etapach przetwarzania modelu. W module TensorFlow występują różne rodzaje kolejek: proste kolejki typu „pierwszy na wejściu, pierwszy na wyjściu” (ang. *First In, First Out* — FIFO; `FIFOQueue`), priorytetyzujące pewne elementy (`PriorityQueue`), taszące zawartość (`RandomShuffleQueue`), a także tworzące grupy elementów o różnych rozmiarach poprzez uzupełnianie (`PaddingFIFOQueue`). Wszystkie te klasy znajdziesz w pakiecie `tf.queue`.

Gdy masz do dyspozycji tensoły, operacje, zmienne i różne struktury danych, możesz zacząć dostosowywać swoje modele i algorytmy uczenia!

Dostosowywanie modeli i algorytmów uczenia

Zacznijmy od stworzenia niestandardowej funkcji straty, gdyż jest to prosta i często spotykana sytuacja.

Niestandardowe funkcje straty

Założymy, że chcesz wytrenować model regresji, ale zestaw danych uczących jest nieco zbyt zaszmiony. Oczywiście najpierw starasz się oczyścić zestaw danych poprzez usunięcie lub poprawienie elementów odstających, ale okazuje się, że to za mało, i dane pozostają zaszumione. Której funkcji straty należy użyć? Błąd średniokwadratowy może zbyt mocno karać duże wartości błędów, przez co model nie będzie precyzyjny. Średni błąd bezwzględny nie karałby elementów odstających tak mocno, ale uzyskiwanie zbieżności mogłoby potrwać długo, a wyuczony model nie byłby zbyt precyzyjny. To jest dobry moment na użycie omówionej w rozdziale 10. funkcji straty Hubera zamiast starego dobrego błędu średniokwadratowego. Funkcja straty Hubera nie należy obecnie do oficjalnego interfejsu Keras, ale jest dostępna w interfejsie `tf.keras` (wystarczy użyć wystąpienia klasy `keras.losses.Huber`). Założymy jednak, że funkcja ta nie jest dostępna — jej zaimplementowanie to kaszka z mlekiem! Wystarczy stworzyć funkcję, która przyjmuje etykiety i prognozy jako argumenty i wykorzystuje operacje TensorFlow do obliczenia funkcji straty dla każdego przykładu:

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
```

```
is_small_error = tf.abs(error) < 1
squared_loss = tf.square(error) / 2
linear_loss = tf.abs(error) - 0.5
return tf.where(is_small_error, squared_loss, linear_loss)
```



Aby uzyskać lepszą wydajność, należy, tak jak w powyższym przykładzie, wykorzystać implementację zwektoryzowaną. Ponadto jeżeli chcesz skorzystać z możliwości grafo-wych TensorFlow, ogranicz się wyłącznie do operacji TensorFlow.

Lepiej jest również zwracać tensor zawierający po jednej funkcji straty na każdy przykład zamiast zwracający uśrednioną funkcję straty. W ten sposób Keras będzie mógł stosować wagi klasy lub wagi przykładów w razie potrzeby (zob. rozdział 10.).

Możesz teraz wykorzystać tę funkcję straty podczas komplikowania modelu Keras, a następnie wytrenować model:

```
model.compile(loss=huber_fn, optimizer="adam")
model.fit(X_train, y_train, [...])
```

I to wszystko! Podczas uczenia Keras będzie dla każdej grupy przykładów wywoływać funkcję huber_fn(), dzięki której zostanie obliczona funkcja straty używana w fazie gradientów prostych. Do tego funkcja ta będzie śledzić całkowitą wartość funkcji straty od początku epoki i wyświetlała jej wartość średnią.

Co się jednak dzieje z tą niestandardową funkcją straty podczas zapisywania modelu?

Zapisywanie i wczytywanie modeli zawierających elementy niestandardowe

Zapisywanie modelu zawierającego niestandardową funkcję straty nie sprawia problemu, ponieważ Keras zapisuje nazwę funkcji. Podczas jego wczytywania musisz wprowadzić słownik, który odwzorowuje nazwę funkcji na właściwą funkcję. Mówiąc ogólniej, jeżeli wczytujesz model zawierający obiekty niestandardowe, musisz odwzorowywać nazwy na te obiekty:

```
model = keras.models.load_model("moj_model_z_niestandardowa_funkcja_straty.h5",
                                custom_objects={"huber_fn": huber_fn})
```

W bieżącej implementacji każda wartość błędu w zakresie od -1 do 1 jest uznawana za „małą”. Czy możemy zmienić te wartości progowe? Jednym z rozwiązań jest stworzenie funkcji definiującej skonfigurowaną funkcję straty:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn
```

```
model.compile(loss=create_huber(2.0), optimizer="adam")
```

Niestety w trakcie zapisywania modelu wartość threshold zostaje pominięta. Oznacza to, że musisz ją wyznaczyć w momencie wczytywania modelu (zwróć uwagę, że należy użyć nazwy huber_fn, czyli nazwy funkcji nadanej w interfejsie Keras, a nie nazwy tworzącej ją funkcji):

```
model = keras.models.load_model("moj_model_z_niestandardowym_progiem_f_straty_2.h5",
                                custom_objects={"huber_fn": create_huber(2.0)})
```

Możemy rozwiązać ten problem poprzez utworzenie podklasy klasy `keras.losses.Loss` i zaimplementowanie jej metody `get_config()`:

```
class HuberLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss = self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```



Keras obecnie pozwala jedynie określić sposób wykorzystania podklas do definowania warstw, modeli, wywołań zwrotnych i regularyzatorów. Jeżeli tworzysz jakieś inne składniki (np. funkcje strat, wskaźniki, inicjalizatory czy ograniczenia) za pomocą podklas, być może nie będziesz w stanie przenosić ich na inne implementacje interfejsu Keras. Prawdopodobnie API zostanie zaktualizowany z myślą o dostosowaniu podklas również do tych elementów.

Przeanalizujmy powyższy listing:

- Konstruktor przyjmuje zmienne `**kwargs` i przekazuje je do konstruktora nadzawanego, który obsługuje standardowe hiperparametry: nazwę funkcji straty (`name`) i algorytm redukujący (`reduction`), który służy do gromadzenia wartości funkcji straty z poszczególnych przykładów. Domyslnie algorytm ten to `"sum_over_batch_size"`, co oznacza, że funkcja straty będzie sumą funkcji straty w instancji ważoną względem wag przykładów (jeśli występują) i podzieloną przez rozmiar grupy (nie przez sumę wag, zatem *nie* mamy tu do czynienia ze średnią ważoną)⁵. Inne dostępne wartości to `"sum"` i `"none"`.
- Metoda `call()` przyjmuje etykiety i prognozy, oblicza funkcje straty dla wszystkich przykładów i je zwraca.
- Metoda `get_config()` zwraca słownik odwzorowujący nazwę każdego hiperparametru do jego wartości. Wywołuje najpierw metodę `get_config()` klasy nadzawanej, a następnie dodaje nowe hiperparametry do tego słownika (wygodna składnia `{x**}` została dodana w Pythonie 3.5).

Możesz teraz wykorzystać dowolny przykład z tej klasy podczas kompilowania modelu:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

Podczas zapisywania modelu wartość progowa zostanie zapisana wraz z nim, natomiast w trakcie jego wczytywania wystarczy odwzorować nazwę klasy na odpowiednią klasę:

⁵ Użycie średniej ważonej byłoby tu złym pomysłem: gdybyś z niej skorzystał, to dwa przykłady o takich samych wagach, ale znajdujące się w różnych grupach miałyby odmienny wpływ na wynik uczenia, w zależności od całkowitej wagi każdej grupy.

```
model = keras.models.load_model("moj_model_z_niestandardowa_klasa_funkcji_straty.h5",
                                custom_objects={"HuberLoss": HuberLoss})
```

Gdy zapisujesz model, Keras wywołuje metodę `get_config()` wystąpienia funkcji straty i zapisuje konfigurację jako format JSON w pliku HDF5. W momencie wczytania modelu zostaje wywołana metoda `from_config()` wobec klasy `HuberLoss` — metoda ta jest implementowana przez klasę bazową (`Loss`) i tworzy wystąpienie klasy poprzez przekazanie `**config` do konstruktora.

To tyle w kwestii funkcji straty! Nie było zbyt trudno, prawda? Równie łatwo jest w przypadku niestandardowych funkcji aktywacji, inicjalizatorów, regularyzatorów i ograniczeń, które stanowią nasz kolejny obiekt zainteresowania.

Niestandardowe funkcje aktywacji, inicjalizatory, regularyzatory i ograniczenia

W taki sam sposób możemy dostosowywać w interfejsie Keras większość składników, takich jak funkcje straty, regularyzatory, ograniczenia, inicjalizatory, wskaźniki, funkcje aktywacji, warstwy, a nawet całe modele. W większości przypadków wystarczy napisać prostą funkcję ze zdefiniowanymi odpowiednimi wejściami i wyjściami. Poniżej prezentuję przykłady niestandardowej funkcji aktywacji (równoważnej funkcji `keras.activations.softplus()` lub `tf.nn.softplus()`), inicjalizatora Glorota (odpowiednika funkcji `keras.initializers.glorot_normal()`), regularyzatora ℓ_1 (ekwiwalentu funkcji `keras.regularizers.l1(0.01)`), a także ograniczenia gwarantującego wartość dodatnią wszystkich wag (równoznacznego z funkcją `keras.constraints.nonneg()` lub `tf.nn.relu()`):

```
def my_softplus(z): # Zwracaną wartością jest po prostu tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)

def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)

def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))

def my_positive_weights(weights): # Zwracaną wartością jest po prostu tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

Jak widać, argumenty zależą od typu niestandardowej funkcji. Możemy teraz korzystać normalnie z tych funkcji, na przykład:

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

Na wyjściu tej warstwy `Dense` zostanie zastosowana funkcja aktywacji, a jej wynik będzie przekazany do następnej warstwy. Wagi warstwy zostaną zainicjalizowane za pomocą wartości zwróconej przez inicjalizator. W każdym przebiegu uczenia wagi będą przekazywane do funkcji regularyzacji w celu obliczenia funkcji straty regularyzacji, która zostanie z kolei dodana do całkowitej funkcji straty używanej podczas trenowania modelu. Zostaje nam jeszcze funkcja ograniczająca, która jest wywoływana po każdym przebiegu uczącym, a wagi warstwy będą zastępowane ograniczonymi wartościami wag.

Jeżeli funkcja zawiera hiperparametry, które należy zapisać wraz z modelem, to chcesz wydzielić podklasę z odpowiedniej klasy, takiej jak keras.regularizers.Regularizer, keras.constraints.Constraint, keras.initializers.Initializer czy keras.layers.Layer (dla dowolnej warstwy, włącznie z funkcjami aktywacji). Podobnie jak to było w przypadku niestandardowej funkcji straty, prezentuję poniżej prostą klasę definiującą regularyzację ℓ_1 , która pozwala zapisywać hiperparametr factor (tym razem nie musimy wywoływać konstruktora nadzawanego czy metody get_config(), ponieważ nie są one zdefiniowane przez klasę nadzawodną):

```
class MyL1Regularizer(keras.regularizers.Regularizer):  
    def __init__(self, factor):  
        self.factor = factor  
    def __call__(self, weights):  
        return tf.reduce_sum(tf.abs(self.factor * weights))  
    def get_config(self):  
        return {"factor": self.factor}
```

Zwróć uwagę, że musisz zaimplementować metodę call() dla funkcji straty, warstw (w tym funkcji aktywacji) i modeli albo metodę __call__() dla regularyzatorów, inicjalizatorów i ograniczeń. Jak się za chwilę przekonasz, sprawia ma się nieco inaczej w przypadku wskaźników.

Niestandardowe wskaźniki

W ujęciu pojęciowym funkcje straty i wskaźniki nie są tożsame: funkcje straty (np. entropia krzyżowa) wykorzystywane są w technice gradientu prostego do **uczenia** modelu, muszą być więc różniczkowalne (przynajmniej w wykorzystywanym zakresie), a ich gradiensty nie mogą być wszędzie równe 0. Ponadto nie muszą być łatwo interpretowane przez człowieka. Z kolei wskaźniki (np. dokładność) są używane do **oceny** modelu: muszą być łatwo interpretowane, a do tego mogą być nieróżniczkowalne lub ich gradiensty mogą być wszędzie równe 0.

Mimo to w większości przypadków definiowanie niestandardowej funkcji metrycznej wygląda tak samo jak definiowane niestandardowej funkcji straty. Istotnie, gdybyśmy chcieli, moglibyśmy wykorzystać nawet utworzoną wcześniej funkcję straty Hubera jako wskaźniki⁶ — spełniałaby swoje zadanie (podobnie jak możliwość zapisu; w tym przypadku wystarczyłoby zapisać nazwę funkcji: "huber_fn"):

```
model.compile(loss="mse", optimizer="nadam", metrics=[create_huber(2.0)])
```

Interfejs Keras w fazie uczenia będzie obliczał dla każdej grupy wskaźnik i śledził jego średnią od początku każdej epoki. Przez większość czasu zależy nam właśnie na tym, ale nie zawsze! Weźmy na przykład pod uwagę precyzyję klasyfikatora binarnego. Jak wiesz z rozdziału 3., precyzyję definiujemy jako stosunek wyników prawdziwie pozytywnych i liczby pozytywnych prognoz (sumy prawdziwie pozytywnych i fałszywie pozytywnych). Założmy, że model dla pierwszej grupy uzyskał pięć prognoz, z których cztery okazały się trafne — jest to precyzyja rzędu 80%. Teraz powiedzmy, że dla drugiej grupy model uzyskał trzy prognozy, z których wszystkie są niewłaściwe — czyli w tym przypadku precyzyja wynosi 0%. Jeżeli obliczysz teraz średnią z tych dwóch wyników, otrzymasz wartość 40%. Ale chwila, przecież to **nie** jest precyzyja modelu dla dwóch grup danych! Rzeczywiście, łącznie otrzymaliśmy cztery prawdziwie pozytywne prognozy (4+0) z ośmiu pozytywnych predykcji

⁶ Funkcja straty Hubera jest jednak rzadko używana jako wskaźnik (lepiej skorzystać z funkcji MAE lub MSE).

(5+3), zatem całkowita precyzaja wynosi nie 40%, lecz 50%. Potrzebujemy obiektu, który będzie śledził liczbę wartości prawdziwie pozytywnych oraz fałszywie pozytywnych i obliczał wspomniany stosunek w razie potrzeby. Dokładnie tego typu zadaniem zajmuje się klasa keras.metrics.Precision:

```
>>> precision = keras.metrics.Precision()  
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])  
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>  
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])  
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

W przykładzie tym utworzyliśmy obiekt Precision, użyliśmy go jak funkcji poprzez przekazanie mu etykiet i prognoz z pierwszej grupy, a następnie przekazaliśmy predykcje z drugiej grupy (moglibyśmy również przekazać wagi przykładów). Wprowadziłmy taką samą liczbę prognoz prawdziwie pozytywnych i fałszywie pozytywnych jak w omówionym przed chwilą przykładzie. Po pierwszej grupie obiekt ten zwraca wartość precyzji na poziomie 80%, a po drugiej otrzymujemy precyzję rzedu 50% (jest to ogólna wartość precyzji, a nie wyliczona dla grupy drugiej). Jest to tak zwany **wskaźnik strumieniowy** (ang. *streaming metric*) lub **wskaźnik stanowy** (ang. *stateful metric*), ponieważ jest on stopniowo aktualizowany grupa po grupie.

Mozemy w dowolnej chwili wywołać metodę result(), aby uzyskać bieżącą wartość wskaźnika. Możemy również sprawdzić jej zmienne (śledzące liczby prognoz prawdziwie pozytywnych i fałszywie pozytywnych) za pomocą atrybutu variables, jeżeli zaś chcemy wyzerować te zmienne, zrobimy to za pomocą metody reset_states():

```
>>> precision.result()  
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>  
>>> precision.variables  
[<tf.Variable 'true_positives:0' [...] numpy=array([4.], dtype=float32)>,  
 <tf.Variable 'false_positives:0' [...] numpy=array([4.], dtype=float32)>]  
>>> precision.reset_states() # Obydwie zmienne zostaną wyzerowane
```

Jeśli musisz utworzyć taki wskaźnik strumieniowy, wyznacz podklasę klasy keras.metrics.Metric. Poniżej prezentuję prosty przykład, który śledzi wartość całkowitej funkcji straty Hubera i liczbę dotychczas wykorzystanych przykładów. Jako rezultat zwracany jest współczynnik, będący zasadniczo średnią funkcją straty Hubera:

```
class HuberMetric(keras.metrics.Metric):  
    def __init__(self, threshold=1.0, **kwargs):  
        super().__init__(**kwargs) # Obsługuje argumenty bazowe (np. dtype)  
        self.threshold = threshold  
        self.huber_fn = create_huber(threshold)  
        self.total = self.add_weight("total", initializer="zeros")  
        self.count = self.add_weight("count", initializer="zeros")  
    def update_state(self, y_true, y_pred, sample_weight=None):  
        metric = self.huber_fn(y_true, y_pred)  
        self.total.assign_add(tf.reduce_sum(metric))  
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))  
    def result(self):  
        return self.total / self.count  
    def get_config(self):  
        base_config = super().get_config()  
        return {**base_config, "threshold": self.threshold}
```

Przeanalizujmy powyższy listing⁷:

- Konstruktor wykorzystuje metodę `add_weight()` do utworzenia zmiennych śledzących stan wskaźnika po przetworzeniu wielu grup — w tym przypadku stany te są definiowane przez sumę wszystkich wartości funkcji straty Hubera (`total`) i liczbę wykorzystanych dotychczas przykładów (`count`). Jeśli wolisz, możesz samodzielnie utworzyć te zmienne. Keras śledzi każdy obiekt `tf.Variable`, który jest wyznaczony jako atrybut (a ogólnie każdy „obserwowałny” obiekt, np. warstwy czy modele).
- Metoda `update_state()` jest wywoływana wtedy, gdy wykorzystujesz wystąpienie tej klasy jako funkcję (tak jest w przypadku obiektu `Precision`). Aktualizuje ona zmienne na podstawie etykiet i prognoz uzyskanych z jednej grupy (i wag przykładów, ale w tym przypadku ignorujemy je).
- Metoda `result()` oblicza i zwraca rezultat końcowy, czyli w naszym przykładzie wskaźnik Hubera uśredniony dla wszystkich przykładów. Kiedy wykorzystujesz wskaźnik jak funkcję, najpierw zostaje wywołana metoda `update_state()`, po niej metoda `result()`, po czym otrzymujemy wynik.
- Implementujemy także metodę `get_config()`, dzięki czemu parametr `threshold` zostaje również zapisany wraz z resztą modelu.
- Domyślona implementacja metody `reset_states()` zeruje wszystkie zmienne (ale możesz zmodyfikować to działanie).



Keras automatycznie zajmie się zapisywaniem zmiennych, a z Twojej strony nie jest wymagane żadne działanie.

Podczas definiowania wskaźnika za pomocą prostej funkcji Keras będzie automatycznie wywoływał go dla każdej grupy i śledził jego średnią w każdej epoce tak samo, jak robiliśmy to własnoręcznie. Zatem jedyną zaletą naszej klasy `HuberMetric` jest możliwość zapisania parametru `threshold`. Jest jednak oczywiste, że niektórych wskaźników, takich jak precyzyj, nie da się uśrednić na podstawie wszystkich grup — w takich przypadkach jedynym rozwiązaniem pozostaje implementacja wskaźnika strumieniowego.

Skoro potrafisz już utworzyć wskaźnik strumieniowy, zbudowanie niestandardowej warstwy to będzie bułka z masłem!

Niestandardowe warstwy

Od czasu do czasu może pojawić się konieczność utworzenia architektury zawierającej jakąś egzotyczną warstwę, której domyślona implementacja nie istnieje w module TensorFlow. W takim przypadku należy utworzyć niestandardową warstwę. Ewentualnie możesz po prostu zbudować bardzo powtarzalną strukturę zawierającą wielokrotnie powtarzane bloki warstw sieci, gdzie najwygodniej byłoby traktować każdy taki blok jak pojedynczą warstwę. Jeżeli na przykład model stanowi sekwencję warstw A, B, C, A, B, C, A, B, C, to możesz chcieć zdefiniować niestandardową warstwę D utworzoną z warstw

⁷ Klasę tę należy traktować wyłącznie w kategoriach dydaktycznych. Prostszą i lepszą implementacją jest utworzenie podklasy klasy `keras.metrics.Mean` — przykład jej implementacji znajdziesz w sekcji „Wskaźniki strumieniowe” w notatniku Jupyter.

A, B, C, dzięki czemu model zostanie uproszczony do struktury D, D, D. Zobaczmy, jak możemy tworzyć niestandardowe warstwy.

Zacznijmy od tego, że nie wszystkie warstwy nie zawierają wag, na przykład keras.layers.Flatten czy keras.layers.ReLU. Jeżeli chcesz zdefiniować taką niestandardową warstwę pozabawioną wag, najprościej jest napisać funkcję i umieścić ją w warstwie keras.layers.Lambda. Na przykład na wejściach poniższej warstwy będzie stosowana funkcja wykładnicza:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

Takiej niestandardowej warstwy możemy następnie używać tak samo jak każdej innej, za pomocą interfejsu sekwencyjnego, funkcyjnego i podkласowego. Możesz również wykorzystać ją jako funkcję aktywacji (ewentualnie możesz zaimplementować activation=tf.exp, activation=keras.activations. \rightarrow exponential lub po prostu activation="exponential"). Warstwa wykładnicza jest czasami stosowana jako warstwa wyjściowa w modelu regresji, gdy prognozowane wartości mogą znacznie różnić się skalą (np. 0,001, 10, 1000).

Jak zapewne się domyślasz, w celu utworzenia niestandardowej warstwy stanowej (np. zawierającej wagi) musisz utworzyć podklasę klasy keras.layers.Layer. Przykładowo w tym listingu implementujemy uproszczoną wersję warstwy Dense:

```
class MyDense(keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="jadro", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="obcielenie", shape=[self.units], initializer="zeros")
        super().build(batch_input_shape) # Musi znajdować się na końcu

    def call(self, x):
        return self.activation(x @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1] + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

Przeanalizujmy ten listing:

- Konstruktor przyjmuje wszystkie hiperparametry jako argumenty (w tym przykładzie units i activation) i, co ważne, również argument **kwargs. Wywołuje on konstruktor nadzędny i przekazuje mu kwargs — w ten sposób rozwiązujemy problem standardowych argumentów, takich jak input_shape, trainable czy name. Następnie konstruktor zapisuje hiperparametry jako atrybuty i przekształca argument activation w odpowiednią funkcję aktywacji za pomocą

funkcji `keras.activations.get()` (przyjmuje ona funkcje, łańcuchy znaków, takie jak "relu" lub "selu", albo wartość `None`)⁸.

- Zadaniem metody `build()` jest utworzenie zmiennych warstwy poprzez wywołanie metody `add_weight()` dla każdej wagi. Metoda `build()` pierwszy raz zostaje wywołana w chwili pierwszego użycia warstwy. Keras będzie wtedy znał wymiary wejść tej warstwy i przekaże te informacje metodzie `build()`⁹, co nieraz jest składnikiem wymaganym do utworzenia niektórych wag. Na przykład aby utworzyć macierz wag połączeń (czyli "jadro"), musimy znać liczbę neuronów tworzących wcześniejszą warstwę — odpowiada to rozmiarowi ostatniego wymiaru wejść. Na końcu metody `build()` (nie może znajdować się nigdzie indziej) musimy wywołać nadzczną metodę `build()` — w ten sposób Keras zostaje poinformowany o ukończeniu warstwy (zostaje po prostu wyznaczony argument `self.built=True`).
- Metoda `call()` realizuje wyznaczone operacje. W tym przypadku obliczamy iloczyn macierzy danych wejściowych `X` i jądra warstwy, dodajemy wektor obciążen, a wynik przetwarzamy za pomocą funkcji aktywacji, dzięki czemu otrzymujemy sygnał na wyjściu warstwy.
- Metoda `compute_output_shape()` zwraca wymiary wyjść w tej warstwie. W tym przypadku są one takie same jak wymiary wejść, a jedyna różnica polega na tym, że ostatni wymiar zostaje liczbą neuronów tworzących warstwę. Zwróć uwagę, że w interfejsie `tf.keras` wymiary są wystąpieniami klasy `tf.TensorShape`, które możemy przekształcić w listy Pythona za pomocą funkcji `as_list()`.
- Metoda `get_config()` niczym nie różni się od poprzednich klas niestandardowych. Zwróć uwagę, że zapisujemy pełną konfigurację funkcji aktywacji za pomocą wywołania metody `keras.activations.serialize()`.

Możesz teraz korzystać z warstwy `MyDense` tak jak z innych warstw!



Zasadniczo możesz pomijać metodę `compute_output_shape()`, ponieważ interfejs `tf.keras` automatycznie określa wymiary wyjściowe, chyba że dana warstwa jest dynamiczna (o czym przekonasz się niebawem). W innych implementacjach interfejsu Keras metoda ta może być wymagana lub jej domyślna konfiguracja zakłada, że wymiary wyjść są takie same jak wymiary wejść.

Aby utworzyć warstwę z wieloma wejściami (np. `Concatenate`), argumentem przekazywanym funkcji `call()` powinna być krotka definiująca wszystkie wejścia; podobnie sytuacja wygląda w przypadku argumentu przekazywanego metodzie `compute_output_shape()`, który powinien być krotką zawierającą wymiary grupy każdego wejścia. Jeśli chcesz przygotować wiele wyjść, metoda `call()` powinna zwracać ich listę, natomiast metoda `compute_output_shape()` powinna zwracać listę wymiarów grup wyjściowych (po jednym na każde wyjście). W tej przykładowej warstwie wyznaczamy dwa wejścia i trzy wyjścia:

```
class MyMultiLayer(keras.layers.Layer):  
    def call(self, x):
```

⁸ Funkcja ta jest specyficzna dla interfejsu `tf.keras`. Należy zastąpić ją funkcją `keras.layers.Activation`.

⁹ W interfejsie Keras argument ten nosi nazwę `input_shape`, ale skoro zawiera również wymiary grupy, wolę nazywać go `batch_input_shape`. To samo dotyczy funkcji `compute_output_shape()`.

```

X1, X2 = X
return [X1 + X2, X1 * X2, X1 / X2]

def compute_output_shape(self, batch_input_shape):
    b1, b2 = batch_input_shape
    return [b1, b1, b1] # Prawdopodobnie powinny być tu obsługiwane reguły rozgłaszania

```

Warstwa ta może być teraz używana jak każda inna, pod warunkiem że ograniczysz się do interfejsu funkcyjnego lub podklasowego (gdyż interfejs sekwencyjny akceptuje wyłącznie warstwy zawierające po jednym wejściu i wyjściu).

Jeżeli dana warstwa ma działać inaczej podczas uczenia, a inaczej w czasie testowania (np. zawiera warstwy Dropout lub BatchNormalization), to musisz do metody `call()` dodać argument `training` i za jego pomocą decydować o mechanizmie działania warstwy. Utwórzmy na przykład warstwę dołączającą szum gaussowski w fazie uczenia (w celu regularyzacji), która przestaje być aktywna podczas testowania (Keras zawiera domyślną implementację tej warstwy: `keras.layers.GaussianNoise`):

```

class MyGaussianNoise(keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape

```

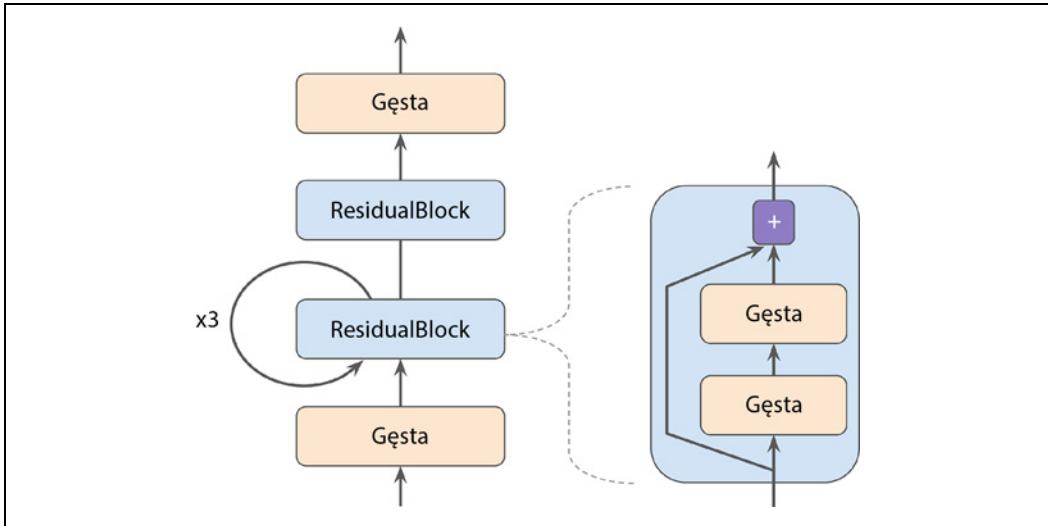
Jesteś już w stanie tworzyć dowolne warstwy niestandardowe! Przejdzmy teraz do tworzenia niestandardowych modeli.

Niestandardowe modele

Mieliszmy już do czynienia z tworzeniem niestandardowych klas modeli w rozdziale 10., gdy omawiam interfejs podklasowy (subclassing API)¹⁰. Proces ten jest bardzo prosty: utwórz podklaśkę klasy `keras.Model`, następnie warstwy i zmienne w konstruktorze i zaimplementuj metodę `call()` w sposób wyznaczający mechanizm działania modelu. Założymy, że chcemy zbudować model zaprezentowany na rysunku 12.3.

Sygnal wejściowy przechodzi najpierw przez warstwę gęstą, następnie trzykrotnie przez ten sam **blok rezydualny** (ang. *residual block*), składający się z dwóch warstw gęstych i operacji sumowania (przekonasz się w rozdziale 14., że blok rezydualny dodaje sygnał wejściowy do wyjściowego), trafia stamtąd do drugiego bloku rezydualnego, a z niego do gęstej warstwy wyjściowej. Zwróć uwagę, że model ten sam w sobie nie ma większego sensu — stanowi on jedynie przykład ilustrujący fakt, jak łatwo jest budować dowolny, własny model, nawet zawierający pętle i połączenia pomijające.

¹⁰ Nazwa „subclassing API” zazwyczaj odnosi się wyłącznie do tworzenia niestandardowych modeli poprzez tworzenie podklas, ale zdążyliśmy się już przekonać, że w ten sposób można tworzyć także wiele innych składników.



Rysunek 12.3. Przykład niestandardowego modelu: dowolny model zawierający niestandardową warstwę ResidualBlock, w której występuje połączenie pomijające

Aby zaimplementować ten model, najlepiej najpierw utworzyć warstwę ResidualBlock, gdyż wykorzystujemy tutaj dwa identyczne bloki (i być może będziemy chcieli je wykorzystać w innym modelu):

```
class ResidualBlock(keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         kernel_initializer="he_normal")
                      for _ in range(n_layers)]

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

Warstwa ta jest nieco specyficzna, gdyż zawiera w sobie inne warstwy. Keras nie ma z tym problemu: automatycznie odkrywa, że atrybut `hidden` przechowuje obiekty obserwowlane (w tym przypadku warstwy), dlatego ich zmienne są automatycznie dodawane do listy zmiennych tej warstwy. Pozostała część klasy nie powinna wymagać dalszych wyjaśnień. Użyjmy teraz interfejsu podklasowego do zdefiniowania właściwego modelu:

```
class ResidualRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                         kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)
```

```
def call(self, inputs):
    Z = self.hidden1(inputs)
    for _ in range(1 + 3):
        Z = self.block1(Z)
    Z = self.block2(Z)
    return self.out(Z)
```

W konstruktorze tworzymy warstwy i wykorzystujemy je w metodzie `call()`. Następnie można używać tego modelu tak jak innych (skompilować go, wyuczyć, ocenić i wyznaczyć do uzyskiwania prognoz). Jeżeli chcesz zapisywać go za pomocą metody `save()` i wczytywać przy użyciu funkcji `keras.models.load_model()`, musisz zaimplementować (tak jak wcześniej) metodę `get_config()`, zarówno w klasie `ResidualBlock`, jak i `ResidualRegressor`. Ewentualnie możesz zapisać i wczytać wagę za pomocą metod, odpowiednio, `save_weights()` i `load_weights()`.

Klasa `Model` stanowi podkласę klasy `Layer`, zatem modele mogą być definiowane i wykorzystywane tak samo jak warstwy. Model ma jednak kilka dodatkowych możliwości, w tym oczywiście metody `compile()`, `fit()`, `evaluate()` i `predict()` (wraz z odmianami), a także metodę `get_layers()` (zwraca ona dowolną warstwę modelu na podstawie nazwy lub indeksu) oraz metodę `save()` (i obsługuje metodę `keras.models.load_model()` i `keras.models.clone_model()`).



Jeżeli modele zapewniają więcej możliwości niż warstwy, to dlaczego nie definiujemy każdej warstwy jako modelu? Technicznie rzecz biorąc, moglibyśmy to robić, ale przeważnie schładziej jest oddzielać wewnętrzne składniki modelu (np. warstwy czy wielokrotnie używane bloki warstw) od samego modelu (tzn. trenowanego obiektu).

Te pierwsze powinny stanowić podkласę klasy `Layer`, natomiast w drugim przypadku należy tworzyć podklasę klasy `Model`.

Dzięki temu możemy zwięźle i dość naturalnie budować niemal każdy zaprojektowany model za pomocą interfejsów sekwencyjnego, funkcjonalnego oraz podklasowego, a nawet ich kombinacji. „Niemal” każdy model? Tak, mimo wszystko ciągle musimy brać pod uwagę pewne aspekty, a mianowicie sposób definiowania funkcji straty i wskaźników na podstawie wewnętrznych elementów modelu oraz mechanizm budowania niestandardowej pętli uczenia.

Funkcje straty i wskaźniki oparte na elementach wewnętrznych modelu

Definiowane przez nas wcześniej niestandardowe funkcje straty i wskaźniki były oparte na etykietach i predykcjach (ewentualnie także na wagach przykładów). W pewnych sytuacjach konieczne jest definiowanie funkcji straty na podstawie innych elementów modelu, np. wag lub funkcji aktywacji warstw ukrytych. Rozwiązania takie przydają się do regularyzacji lub do monitorowania jakiegoś aspektu modelu.

Aby zdefiniować niestandardową funkcję straty opartą na elementach wewnętrznych modelu, oblicz ją dla wymaganego elementu modelu, a następnie przekaż wynik do metody `add_loss()`. Zbudujmy na przykład niestandardowy regresyjny perceptron wielowarstwowy składający się z sekwencji pięciu warstw ukrytych i warstwy wyjściowej. Model będzie miał również wyjście dodatkowe z górnej warstwy ukrytej. Funkcję straty powiązaną z tym dodatkowym wyjściem nazwiemy **funkcją straty rekonstrukcji** (ang. *reconstruction loss*) (zob. rozdział 17.); jest to różnica średniokwadratowa między rekonstrukcją a sygnałem wejściowym. Jeżeli dodamy funkcję straty rekonstruk-

cji do głównej funkcji straty, wymusimy zachowanie jak największej ilości informacji w warstwach ukrytych (w tym również informacji, które nie są bezpośrednio potrzebne w zadaniu regresji). W praktyce ta funkcja straty czasami poprawia zdolność uogólniania (jest to funkcja straty regulatoryzacji). Oto listing definiujący taki model z niestandardową funkcją straty rekonstrukcji:

```
class ReconstructingRegressor(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                         kernel_initializer="lecun_normal")]
        for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

Przeanalizujmy ten kod:

- Konstruktor tworzy głęboką sieć neuronową zawierającą pięć gęstych warstw ukrytych i jedną gęstą warstwę wyjściową.
- Metoda `build()` generuje dodatkową warstwę gęstą, której zadaniem będzie rekonstruowanie sygnałów wejściowych modelu. Musimy ją utworzyć tutaj, ponieważ liczba tworzących ją jednostek musi się zgadzać z liczbą wejść, która jest nieznana przed wywołaniem tej metody.
- Metoda `call()` przetwarza sygnały wejściowe przez wszystkie pięć warstw ukrytych, po czym przekazuje wyniki przez warstwę rekonstrukcji, co pozwala uzyskać rekonstrukcję sygnału.
- Następnie metoda `call()` oblicza funkcję straty rekonstrukcji (różnicę średniokwadratową między rekonstrukcją a sygnałami wejściowymi) i za pomocą metody `add_loss()` dodaje tę funkcję do listy funkcji straty¹¹. Zwróć uwagę, że skalujemy funkcję straty przez współczynnik 0,05 (możesz stroić ten hiperparametr). Mamy w ten sposób pewność, że funkcja straty rekonstrukcji nie będzie dominującą składową w ogólnej funkcji straty.
- Na koniec metoda `call()` przekazuje wynik działania warstw ukrytych do warstwy wyjściowej i zwraca rezultat końcowy.

W podobny sposób możesz dodać niestandardowy wskaźnik oparty na elementach wewnętrznych modelu poprzez obliczenie go w dowolny sposób, pod warunkiem że uzyskany rezultat będzie wynikiem obiektu wskaźnika. Na przykład możesz utworzyć w konstruktorze obiekt `keras.metrics`.

¹¹ Możesz również wywołać metodę `add_loss()` w dowolnej warstwie modelu, gdyż model rekurencyjnie gromadzi wartości funkcji straty z każdej warstwy.

Mean, następnie wywołać go w metodzie `call()` (przekazawszy jej wynik `recon_loss`), a na końcu dodać go modelu poprzez wywołanie metody `add_metric()` modelu. W ten sposób podczas uczenia modelu Keras będzie wyświetlał w każdej epoce zarówno średnią funkcję straty (całkowitą funkcją straty jest tu suma głównej funkcji straty i pięciu procent funkcji straty rekonstrukcji), jak i średnią funkcję straty rekonstrukcji. Obydwie wartości będą maleć w czasie uczenia:

```
Epoch 1/5
11610/11610 [=====] [...] loss: 4.3092 - reconstruction_error: 1.7360
Epoch 2/5
11610/11610 [=====] [...] loss: 1.1232 - reconstruction_error: 0.8964
[...]
```

W ponad 99% przypadków przekazane do tej pory informacje powinny wystarczyć do zaimplementowania dowolnego interesującego Cię modelu, nawet takiego, który zawiera skomplikowane struktury, funkcje straty i wskaźniki. Jednak w pewnych rzadkich sytuacjach może zaistnieć potrzeba modyfikacji samej pętli uczenia. Zanim się tym zajmiemy, musimy dowiedzieć się, w jaki sposób gradienty mogą być automatycznie obliczane w module TensorFlow.

Obliczanie gradientów za pomocą różniczkowania automatycznego

Aby zobaczyć, w jaki sposób automatycznie obliczać gradienty za pomocą różniczkowania automatycznego (zob. rozdział 10. i dodatek D), zdefiniujmy prostą funkcję:

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

Jeżeli znasz się na analizie matematycznej, to możesz analitycznie dowieść, że pochodną cząstkową tej funkcji w odniesieniu do w_1 jest $6 \cdot w_1 + 2 \cdot w_2$. Z kolei pochodna cząstkowa w odniesieniu do w_2 to $2 \cdot w_1$. Na przykład w punkcie $(w_1, w_2) = (5, 3)$ wartości tych pochodnych są równe, odpowiednio, 36 i 10, zatem wektor gradientów w tym punkcie wynosi $(36, 10)$. Gdybyśmy mieli tu jednak do czynienia z siecią neuronową, funkcja byłaby o wiele bardziej skomplikowana (zawierałaby zazwyczaj dziesiątki tysięcy parametrów) i własnoręczne wyznaczenie jej pochodnych cząstkowych w sposób analityczny granicyłoby z cudem. Jednym z rozwiązań okazuje się obliczenie przybliżenia każdej pochodnej cząstkowej poprzez pomiar stopnia zmian wyników funkcji w reakcji na modyfikację określonego parametru:

```
>>> w1, w2 = 5, 3
>>> eps = 1e-6
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps
36.000003007075065
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps
10.00000003174137
```

Wygląda na to, że wszystko się zgadza! Rozwiążanie to sprawdza się całkiem nieźle i jest proste do zaimplementowania, ale nie zapominajmy, że to tylko przybliżenie, a do tego musimy wywoływać `f()` przynajmniej raz na każdy parametr (nie dwa razy, ponieważ możemy obliczyć `f(w1, w2)` tylko raz). Właśnie ten drugi problem sprawia, że technika przestaje być użyteczna w dużych sieciach neuronowych. Z tego powodu powinniśmy wykorzystać różniczkowanie automatyczne. TensorFlow bardzo ułatwia nam to zadanie:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
```

```
z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

Najpierw definiujemy dwie pierwsze zmienne `w1` i `w2`, następnie tworzymy tzw. taśmę gradientów `tf.GradientTape`, która będzie automatycznie rejestrowała każdą operację wykorzystującą zmienną, a na koniec chcemy, aby kontekst ten obliczał gradienty wyniku z w odniesieniu do obydwu zmiennych `[w1, w2]`. Sprawdźmy, jakie gradienty zostały obliczone w TensorFlow:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Doskonale! Wynik nie tylko jest dokładny (jedynym ograniczeniem precyzji są błędy zmiennoprzecinkowe), ale także metoda `gradient()` wykonuje tylko jeden przebieg przez zarejestrowane obliczenia (w odwrotnej kolejności) bez względu na liczbę występujących zmiennych, zatem rozwiązanie to jest niebyvale skuteczne. Przypomina ono wprost magię!



Aby zaoszczędzić pamięć, umieszczaj tylko niezbędne minimum w bloku `tf.GradientTape()`. Ewentualnie możesz wstrzymać rejestrowanie za pomocą bloku `tape.stop_recording()`, umieszczonego w bloku `tf.GradientTape()`.

Taśma została automatycznie i natychmiastowo usunięta po wywołaniu metody `gradient()`, dlatego zostanie wyświetlony komunikat o wyjątku, jeżeli spróbujesz wywołać `gradient()` dwukrotnie:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) #=> Tensor 36.0
dz_dw2 = tape.gradient(z, w2) #RuntimeError!
```

Jeżeli musisz wywoływać metodę `gradient()` częściej niż raz, trzeba sprawić, żeby taśma stała się trwała, i kasować ją za każdym razem, gdy przestaje być potrzebna, w ten sposób bowiem zwolnisz część zasobów¹²:

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1) #=> Tensor 36.0
dz_dw2 = tape.gradient(z, w2) #=> Tensor 10.0, teraz działa właściwie!
del tape
```

Domyślnie taśma będzie śledziła tylko operacje wykorzystujące zmienne, dlatego jeżeli spróbujesz obliczyć gradient z w odniesieniu do czegoś innego niż zmienna, to w rezultacie będzie zwracana wartość `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2]) # Zwraca [None, None]
```

¹² Jeżeli taśma wykracza poza zakres (np. gdy powraca funkcja, która raz już ją wykorzystała), mechanizm odśmieciania pamięci usunie ją automatycznie.

Możesz jednak zmusić taśmę do obserwowania dowolnego tensora po to, aby rejestrowała każdą operację związaną z tym tensorem. Możesz następnie obliczać gradienty w odniesieniu do tych tensorów tak, jakby tensory te były zmiennymi:

```
with tf.GradientTape() as tape:  
    tape.watch(c1)  
    tape.watch(c2)  
    z = f(c1, c2)  
  
gradients = tape.gradient(z, [c1, c2]) # Zwraca [tensor 36., tensor 10.]
```

Jest to przydatne w pewnych sytuacjach, na przykład wtedy, gdy chcesz zaimplementować funkcję straty regularizacji nakładającą kary na funkcje aktywacji o dużym zróżnicowaniu, podczas gdy dane wejściowe nie są mocno zróżnicowane — funkcja straty będzie bazować na gradiencie funkcji aktywacji w odniesieniu do sygnałów wejściowych. Dane wejściowe nie są zmiennymi, dlatego należałoby śledzić je za pomocą taśmy.

Przez większość czasu taśma gradientowa jest wykorzystywana do obliczania gradientów pojedynczej wartości (najczęściej funkcji straty) w odniesieniu do zbioru wartości (zazwyczaj parametrów modelu). To właśnie w takiej sytuacji odwrotne różniczkowanie automatyczne sprawdza się najlepiej, ponieważ wystarczy jeden przebieg w przód i jeden wstecz, aby uzyskać wszystkie gradienty naraz. Jeżeli spróbujesz obliczyć gradient wektora (np. zawierającego wartości wielu funkcji straty), to TensorFlow obliczy gradienty sumy tego wektora. Jeśli więc będziesz kiedyś potrzebować pojedynczych gradientów (tzn. gradientów każdej funkcji straty w odniesieniu do parametrów modelu), musisz wywołać metodę `jacobian()` taśmy — przeprowadzi ona odwrotne różniczkowanie automatyczne dla każdej funkcji straty w wektorze (domyślnie operacja ta jest realizowana równolegle). Możliwe jest nawet obliczanie pochodnych cząstkowych drugiego rzędu (hesjanów, czyli pochodnych cząstkowych z pochodnych cząstkowych), ale w praktyce rzadko jest to potrzebne (przykład znajdziesz w sekcji „Obliczanie gradientów za pomocą różniczkowania automatycznego” w notatniku Jupyter).

Czasami trzeba w pewnych obszarach sieci zatrzymać proces propagacji wstecznej gradientów. Aby to zrobić, musisz użyć funkcji `tf.stop_gradient()`. Funkcja ta zwraca swoje dane wejściowe podczas przebiegu w przód (podobnie jak funkcja `tf.identity()`), ale nie przepuszcza gradientów w czasie fazy propagacji wstecznej (zachowuje się jak stała):

```
def f(w1, w2):  
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)  
  
with tf.GradientTape() as tape:  
    z = f(w1, w2) # Taki sam wynik jak uzyskany bez użycia stop_gradient()  
  
gradients = tape.gradient(z, [w1, w2]) # => Zwraca [tensor 30., None]
```

Możesz także natrafiać czasami na problemy numeryczne podczas obliczania gradientów. Na przykład jeśli obliczysz gradienty funkcji `my_softplus()` dla dużych wartości wejściowych, otrzymasz wynik `NaN`:

```
>>> x = tf.Variable([100.])  
>>> with tf.GradientTape() as tape:  
...     z = my_softplus(x)  
...  
>>> tape.gradient(z, [x])  
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

Wynika to stąd, że obliczanie gradientów tej funkcji za pomocą różniczkowania automatycznego prowadzi do pewnych trudności numerycznych: z powodu błędów precyzyji w wartościach zmiennoprzecinkowych różniczkowanie automatyczne może próbować dzielić nieskończoność przez nieskończoność (co prowadzi do wyniku *Nan*). Na szczęście możemy dowieść analitycznie, że pochodna funkcji softplus to po prostu $1/(1+exp(x))$, która jest numerycznie stabilna. Możemy sprawić za pomocą dekoratora `@tf.custom_gradient`, że TensorFlow będzie korzystał z tej pochodnej podczas obliczania gradientów funkcji `my_softplus()` — dekorator ten będzie zwracał zarówno normalny wynik, jak i funkcję obliczającą pochodne (zauważ, że na wejściu będzie otrzymywał dotychczas uzyskane gradienty w wyniku propagacji wstecznej, aż do poziomu funkcji softplus; zgodnie z regułą lańcuchową należy pomnożyć je przez gradienty tej funkcji):

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

Teraz podczas obliczania gradientów funkcji `my_better_softplus()` otrzymujemy właściwy wynik, nawet w przypadku dużych wartości wejściowych (mimo to główny wynik „eksploduje” z powodu wykładnika; jednym z rozwiązań jest użycie metody `tf.where()` tak, aby zwracane były dane wejściowe, gdy mają dużą wartość).

Gratulacje! Umiesz już obliczać gradienty dowolnej funkcji (pod warunkiem, że jest ona różniczkowalna w obliczanym punkcie), blokować propagację wstecną w razie potrzeby, a także pisać własne funkcje gradientowe! Prawdopodobnie możliwości te nie będą Ci nigdy potrzebne, nawet podczas tworzenia opisanych poniżej niestandardowych pętli uczenia.

Niestandardowe pętle uczenia

W pewnych rzadkich sytuacjach metoda `fit()` może nie być wystarczająco elastyczna. Na przykład w omówionej w rozdziale 10. publikacji Wide & Deep (<https://arxiv.org/abs/1606.07792>) autorzy stosują dwa różne optymalizatory: jeden dla ścieżki szerokiej, a drugi dla głębokiej. Metoda `fit()` wykorzystuje tylko jeden optymalizator (wyznaczany w fazie kompilowania modelu), dlatego do zaimplementowania modelu omówionego w tej publikacji wymagane jest napisanie własnej pętli uczenia.

Możesz również tworzyć niestandardowe pętle uczenia po to, aby mieć pewność, że będą precyzyjnie realizować powierzone zadanie (być może nie masz pewności co do jakichś szczegółów metody `fit()`). Czasami bezpieczniej jest deklarować wszystko jawnie. Pamiętaj jednak, że taka własna pętla uczenia zwiększa objętość kodu i jego podatność na błędy oraz utrudnia jego utrzymywanie.



Jeżeli dodatkowa swoboda, zapewniana przez niestandardowe pętle uczenia, nie jest Ci niezbędna, korzystaj z metody `fit()`, zwłaszcza podczas pracy w zespole.

Zbudujmy najpierw prosty model. Nie musimy go kompilować, gdyż będziemy ręcznie zajmować się pętlą uczenia:

```

l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu", kernel_initializer="he_normal",
                      kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])

```

Utworzmy teraz malutką funkcję, która losuje grupę przykładów z zestawu danych uczących (w rozdziale 13. omówię API Data, który stanowi znacznie lepsze rozwiązanie):

```

def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]

```

Zdefiniujmy także funkcję wyświetlającą postępy uczenia, w tym bieżący przebieg, całkowitą liczbę przebiegów, średnią funkcję straty liczoną od początku epoki (w tym celu użyjemy wskaźnika Mean), a także inne wskaźniki:

```

def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}"].format(m.name, m.result())
                          for m in [loss] + (metrics or [])))
    end = "" if iteration < total else "\n"
    print("\r{} / {} - ".format(iteration, total) + metrics,
          end=end)

```

Jeżeli nie znasz formatowaniałańcuchów znaków w Pythonie, to powyższy listing wymaga krótkiego omówienia: zapis `{:.4f}` formatuje wartość zmiennoprzecinkową z dokładnością do czterech miejsc po przecinku, natomiast kombinacja `\r` (powrotu karetki) i `end=""` spowoduje, że pasek stanu będzie zawsze wyświetlany w tym samym wierszu. W utworzonym na potrzeby tego rozdziału notatniku Jupyter pasek postępu otrzymujemy za pomocą funkcji `print_status_bar()`, ale można ją zastąpić przydatną biblioteką `tqdm`.

Możesz w końcu przejść do sedna! Musimy najpierw zdefiniować hiperparametry, a także wybrać optymalizator, funkcję straty i wskaźniki (w tym przykładzie wystarczy błąd MAE):

```

n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]

```

Wreszcie możemy zbudować niestandardową pętlę!

```

for epoch in range(1, n_epochs + 1):
    print("Epoka numer {} / {}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
            gradients = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))
            mean_loss(loss)
        for metric in metrics:

```

```

        metric(y_batch, y_pred)
    print_status_bar(step * batch_size, len(y_train), mean_loss, metrics)
    print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
    for metric in [mean_loss] + metrics:
        metric.reset_states()

```

Dzieje się tu całkiem sporo, dlatego przeanalizujmy ten kod:

- Tworzymy dwie pętle zagnieżdżone: jedną dla epok, a drugą dla grup danych w każdej epoce.
- Następnie pobieramy losowo grupę przykładów z zestawu danych uczących.
- W bloku `tf.GradientTape()` uzyskujemy prognozę dla jednej grupy (używamy modelu jak funkcji) i obliczamy funkcję straty — jest ona równa sumie głównej funkcji straty i pozostałych funkcji straty (w omawianym modelu w każdej warstwie występuje jedna funkcja straty regularizacji). Funkcja `mean_squared_error()` zwraca po każdej funkcji straty na każdy przykład, obliczamy średnią dla całej grupy za pomocą funkcji `tf.reduce_mean()` (gdybyśmy chcieli wyznaczać inne wagę w każdym przykładzie, zrobilibyśmy to właśnie tutaj). Każda funkcja straty regularizacji zostaje zredukowana do pojedynczej wartości skalarnej, dlatego wystarczy je zsumować (za pomocą metody `tf.add_n()`, która sumuje wiele tensorów o tych samych wymiarach i typie danych).
- Chcemy następnie, aby taśma obliczyła gradient funkcji straty w odniesieniu do każdej modyfikowej zmiennej (*nie* do wszystkich zmiennych!). Gradient ten zostaje przekazany optymalizatorowi w celu zrealizowania fazy gradientu prostego.
- Teraz aktualizujemy uśrednioną funkcję straty i wskaźniki (dla bieżącej epoki) oraz wyświetlamy pasek stanu.
- Na końcu każdej epoki wyświetlamy ponownie pasek stanu po to, aby wyglądał na kompletny¹³, a także aby wyświetlić nowy wiersz, po czym zerujemy stany średniej funkcji straty i wskaźników.

Jeżeli wyznaczysz hiperparametr `clipnorm` lub `clipvalue` optymalizatora, to operacje obcinania gradientów będą wykonywane automatycznie. Jeżeli chcesz wprowadzić inne przekształcenia gradientów, zrób to przed wywołaniem metody `apply_gradients()`.

Jeżeli dodasz ograniczenia wag do modelu (np. poprzez wyznaczenie `kernel_constraint` lub `bias_constraint` podczas tworzenia warstwy), zaktualizuj pętlę uczenia do uwzględniania tych ograniczeń tuż za metodą `apply_gradients()`:

```

for variable in model.variables:
    if variable.constraint is not None:
        variable.assign(variable.constraint(variable))

```

Najważniejszy fakt jest taki, że ta pętla uczenia nie obsługuje warstw działających odmiennie w czasie uczenia i testowania (tzn. warstw BatchNormalization i Dropout). Aby to zmienić, musisz wywołać model z parametrem `training=True` i rozprowadzić go do każdej wymagającej tego warstwy.

¹³ W rzeczywistości nie przetworzyliśmy wszystkich przykładów z zestawu danych uczących, ponieważ próbkovaliśmy je losowo: niektóre były przetwarzane kilka razy, a inne zostały zupełnie pominięte. Ponadto jeżeli rozmiar zestawu danych nie stanowi wielokrotności rozmiaru grupy, to kilka przykładów zostanie pominiętych. W praktyce nie stanowi to problemu.

Jak widać, należy się zatroszczyć o wiele rzeczy i bardzo łatwo popełnić błąd. Z drugiej jednak strony uzyskujesz pełną kontrolę nad modelem, więc wybór należy do Ciebie.

Skoro potrafisz już dostosowywać każdy aspekt modelu¹⁴ i algorytmu uczącego, zajmijmy się możliwością automatycznego generowania grafów — może ona znacznie przyspieszyć działanie niestandardowego kodu, a także pozwala przenosić go na każdą platformę obsługującą moduł TensorFlow (zob. rozdział 19.).

Funkcje i grafy modułu TensorFlow

W pierwszej wersji modułu TensorFlow nie dało się unikać grafów (ani związań z nimi zawiłości), ponieważ stanowiły one główny element API. Występują one również w module TensorFlow 2, nie stanowią jednak jego kluczowego elementu i są znacznie (znacznie!) łatwiejsze w użytkowaniu. Aby to udowodnić, zdefiniujemy trywialną funkcję obliczającą szescian z danych wejściowych:

```
def cube(x):
    return x ** 3
```

Możemy oczywiście wywołać tę funkcję z wartością języka Python, taką jak *int* czy *float*, albo z tensorem:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

Wykorzystajmy teraz metodę `tf.function()` do przekształcenia funkcji Pythona w **funkcję TensorFlow**:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

Funkcji tej można używać tak samo jak standardowej funkcji Pythona i będzie zwracać identyczne wyniki (ale w postaci tensorowej):

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

Metoda `tf.function()` przeanalizowała poza naszym wzrokiem obliczenia przeprowadzone przez funkcję `cube()` i wygenerowała na tej podstawie graf obliczeniowy! Jak widać, proces ten był raczej bezproblemowy (niebawem przyjrzymy mu się uważniej). Ewentualnie możemy wykorzystać `tf.function` jako dekorator — rozwiązanie to jest dość powszechnie spotykane:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

¹⁴ Oprócz optymalizatorów, gdyż bardzo rzadko istnieje potrzeba ich modyfikowania; przykład znajdziesz w sekcji „Niestandardowe optymalizatory” w notatniku Jupyter.

Pierwotna funkcja Pythona jest ciągle dostępna w atrybucie `python_function` na wypadek, gdybyśmy kiedykolwiek jej potrzebowali:

```
>>> tf_cube.python_function(2)  
8
```

TensorFlow optymalizuje grafy obliczeniowe poprzez usuwanie nieużywanych węzłów, upraszczanie wyrażeń (np. zapis $1 + 2$ zostanie zastąpiony wartością 3) itd. Po przygotowaniu zoptymalizowanego grafu funkcja TF wydajnie przetwarza zdefiniowane w nim operacje, w odpowiedniej kolejności (i, w razie możliwości, równolegle). W konsekwencji funkcja TF działa zazwyczaj znacznie szybszej od jej pierwotnego odpowiednika, zwłaszcza w przypadku skomplikowanych obliczeń¹⁵. Przez większość czasu ta wiedza powinna w zupełności Ci wystarczyć: jeżeli chcesz przyspieszyć funkcję Pythona, wystarczy przekształcić ją w funkcję TF. I to wszystko!

Ponadto podczas pisania niestandardowej funkcji straty, wskaźnika, warstwy lub dowolnej innej funkcji niestandardowej używanej w modelu Keras (tak jak robimy to w tym rozdziale) będzie ona automatycznie przekształcana w funkcję TF, bez potrzeby stosowania metody `tf.function()`. Zatem przez większość czasu magia ta jest zupełnie niedostrzegalna.



Możesz wyłączyć możliwość przekształcania funkcji Pythona w funkcje TF przez Keras, jeżeli wyznaczyś parametr `dynamic=True` w trakcie tworzenia niestandardowej warstwy lub niestandardowego modelu. Ewentualnie możesz wstawić `run_eagerly=True` na etapie wywoływania metody `compile()` modelu.

Funkcja TF generuje domyślnie nowy graf dla każdego niepowtarzanego zbioru wejściowych wymiarów i typów danych oraz przechowuje go na wypadek kolejnych wywołań. Na przykład jeżeli wywołasz `tf_cube(tf.constant(10))`, zostanie wygenerowany graf dla tensorów typu `int32` o wymiarach `[]`. Jeżeli następnie wywołasz `tf_cube(tf.constant(20))`, zostanie ponownie wykorzystany ten sam graf. Jeśli jednak wywołasz teraz `tf_cube(tf.constant([10, 20]))`, zostanie wygenerowany nowy graf dla tensorów `int32` o wymiarach `[2]`. Właśnie w taki sposób funkcje TF obsługują wielopostaciowość (tzn. różnice typów i wymiarów argumentów). Jest to jednak prawda wyłącznie dla argumentów tensorowych: jeżeli przekażesz numeryczne wartości języka Python do funkcji TF, zostanie wygenerowany osobny graf dla poszczególnych wartości, na przykład wywołanie `tf_cube(10)` i `tf_cube(20)` wygeneruje dwa grafy.

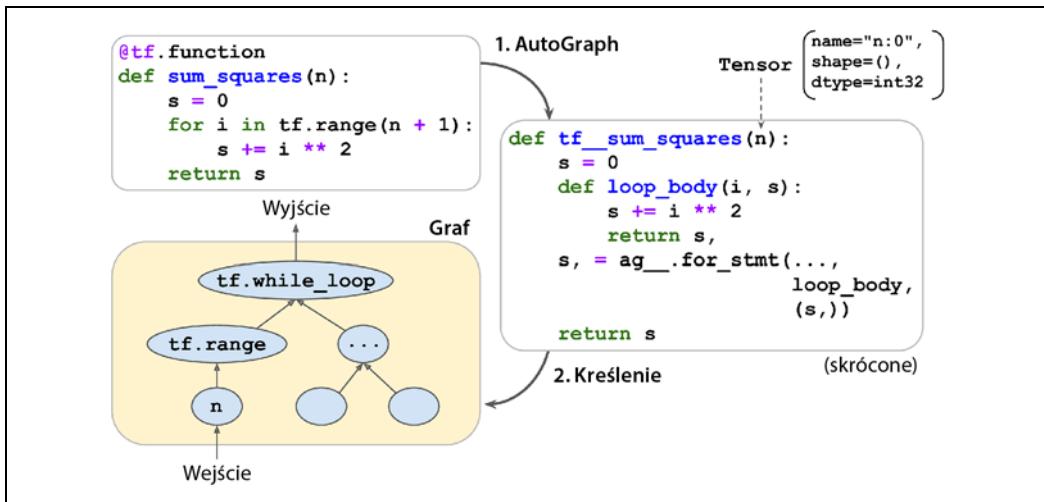


Jeżeli będziesz wielokrotnie wywoływać funkcję TF z różnymi wartościami numerycznymi Pythona, to zostanie wygenerowanych wiele grafów, co spowolni działanie programu i zajmie znaczną część pamięci operacyjnej (aby ją zwolnić, trzeba usunąć funkcję TF). Wartości języka Python powinny być zarezerwowane dla argumentów mających niewiele niepowtarzalnych wartości, np. takich hiperparametrów jak liczba neuronów tworzących daną warstwę. Dzięki temu TensorFlow jest w stanie skuteczniej optymalizować każdy wariant modelu.

¹⁵ Omawiany przykład jest jednak tak prosty, że nie ma w nim niczego, co można byłoby zoptymalizować, dlatego w istocie funkcja `tf_cube()` działa znacznie wolniej od funkcji `cube()`.

AutoGraph i kreślenie

W jaki więc sposób TensorFlow generuje grafy? Najpierw analizuje kod źródłowy funkcji Pythona w celu wychwycenia wszystkich instrukcji przebiegu sterowania, takich jak pętle `for` i `while` czy instrukcje `if`, a także `break`, `continue` oraz `return`. Pierwszy etap nosi nazwę **AutoGraph**. TensorFlow musi analizować kod źródłowy, ponieważ w Pythonie nie istnieje żaden inny sposób wyszukiwania instrukcji przebiegu sterowania — dostępne są metody magiczne, takie jak `__add__()` i `__mul__()`, pozwalające wychwytywać operatory, odpowiednio `+ i *`, ale nie znajdziemy metod typu `__while__()` czy `__if__()`. Po przeanalizowaniu kodu funkcji AutoGraph tworzy zaktualizowaną jego wersję, w której instrukcje przebiegu sterowania zostają zastąpione odpowiednimi operacjami TensorFlow, na przykład `tf.while_loop()` w przypadku pętli i `tf.cond()` zastępujących instrukcję `if`. Na rysunku 12.4 AutoGraph analizuje kod źródłowy funkcji `sum_squares()` i generuje z niego funkcję `tf_sum_squares()`. W funkcji tej pętla `for` zostaje zastąpiona definicją funkcji `loop_body()` (zawierającą ciało pierwotnej pętli `for`), po której umieszczono wywołanie funkcji `for_stmt()`. Zadaniem tego wywołania jest utworzenie właściwej operacji `tf.while_loop()` w grafie obliczeniowym.



Rysunek 12.4. Mechanizm generowania grafów TensorFlow za pomocą rozwiązania AutoGraph i kreślenia

Następnie TensorFlow wywołuje taką „udoskonaloną” funkcję, ale nie przekazuje jej argumentu, lecz **tensor symboliczny** (ang. *symbolic tensor*), czyli tensor niezawierający faktycznej wartości, lecz jedynie nazwę, typ danych i wymiary. Na przykład jeżeli wywołasz funkcję `sum_squares(tf.constant(10))`, to zostanie wywołana funkcja `tf_sum_squares()` z tensorem symbolicznym typu `int32` i o wymiarach `[]`. Funkcja będzie działała w **trybie grafowym** (ang. *graph mode*), co oznacza, że każda operacja TensorFlow będzie dodawała symbolizujący ją węzeł, a także jej tensor/-y wyjściowy/-e do grafu (w przeciwieństwie do trybu standardowego, znanego jako **tryb pośpieszny** (ang. *eager mode*) lub **realizacja pośpieszna** (ang. *eager execution*)). W trybie grafowym operacje TF nie przeprowadzają żadnych obliczeń. Taka sama sytuacja występowała w module TensorFlow 1, gdyż tryb grafowy był domyślny. Na rysunku 12.4 widzimy funkcję `tf_sum_squares()` wywoływaną z tensorem symbolicznym jako argumentem (w tym przypadku jest on typu `int32` i ma wymiary `[]`), a graf wynikowy

zostaje utworzony w fazie kreślenia. Węzły wyznaczają operacje, a strzałki reprezentują tensory (zarówno wygenerowana funkcja, jak i graf są uproszczone).



Jeżeli chcesz zobaczyć kod źródłowy wygenerowanej funkcji, możesz wywołać `tf.autograph.to_code(sum_squares.python_function)`. Kod ten nie będzie wyglądał zbyt schludnie, ale czasami zerknięcie do niego pozwala poradzić sobie ze związanymi z nim problemami.

Reguły związane z funkcją TF

W większości przypadków przekształcanie funkcji języka Python realizującej operacje TensorFlow w funkcję TF jest banalne: wystarczy użyć dekoratora `@tf.function` lub pozwolić, aby interfejs Keras wykonał to automatycznie. Należy jednak przestrzegać kilku reguł:

- Jeżeli wywołujesz jakąkolwiek bibliotekę zewnętrzną, w tym NumPy, a nawet bibliotekę standarową, wywołanie to zostanie zrealizowane wyłącznie w fazie kreślenia, nie będzie stanowiło części grafu. Istotnie, graf TensorFlow może zawierać wyłącznie konstrukty TensorFlow (tensorы, operacje, zmienne, zestawy danych itd.). Pamiętaj więc, aby zastępować funkcję `np.sum()` funkcją `tf.reduce_sum()`, zamiast funkcji `sorted()` używać funkcji `tf.sort()` itd. (chyba że zależy Ci na tym, aby kod był realizowany wyłącznie podczas kreślenia). Wynikają z tego dodatkowe konsekwencje:
 - Jeżeli definiujesz funkcję TF `f(x)` zwracającą jedynie `np.random.rand()`, wartość losowa zostanie wygenerowana wyłącznie podczas kreślenia, zatem funkcje `f(tf.constant(2.))` i `f(tf.constant(3.))` zwrócią tę samą wartość, ale już `f(tf.constant([2., 3.]))` zwróci inną liczbę. Jeżeli zastąpisz funkcję `np.random.rand()` funkcją `tf.random.uniform([])`, to przy każdym wywołaniu będzie generowana nowa liczba losowa, ponieważ operacja ta będzie teraz stanowiła część grafu.
 - Jeżeli kod niezawierający wyłącznie konstruktów TensorFlow wykonuje dodatkowe czynności (np. zapisuje rekordy w dzienniku zdarzeń lub aktualizuje licznik środowiska Python), to nie będą one realizowane przy każdym wywołaniu funkcji TF, lecz tylko w fazie kreślenia.
 - Możesz umieścić dowolny kod Pythona w operacji `tf.py_function()`, ale rozwiązywanie to obniży wydajność, ponieważ TensorFlow nie będzie w stanie zoptymalizować grafu. Zostanie zredukowana również przenośność grafu, gdyż będzie mógł on być realizowany wyłącznie na platformach zawierających środowisko Python (oraz tam, gdzie są zainstalowane odpowiednie biblioteki).
- Możesz wywoływać inne funkcje Pythona lub funkcje TF, ale muszą one przestrzegać tych samych zasad, gdyż TensorFlow będzie umieszczał ich operacje w grafie obliczeniowym. Te inne funkcje nie muszą zawierać dekoratora `@tf.function`.
- Jeżeli funkcja tworzy zmienną TensorFlow (lub dowolny inny obiekt stanowy TensorFlow, np. zestaw danych lub kolejkę), to musi to zrobić tylko i wyłącznie w pierwszym wywołaniu, ponieważ w przeciwnym razie zostanie wyświetlony komunikat o wyjątku. Zazwyczaj najlepiej jest tworzyć zmienne poza funkcją TF (tzn. w metodzie `build()` niestandardowej warstwy). Jeżeli chcesz przypisać nową wartość do zmiennej, pamiętaj, aby wywołać jej metodę `assign()` zamiast korzystać z operatora `=`.

- Kod źródłowy funkcji Pythona powinien być dostępny dla modułu TensorFlow. Jeżeli kod ten nie jest dostępny (gdyż na przykład zdefiniujesz funkcję w powłoce Pythona, przez co nie masz dostępu do kodu źródłowego, lub wdrażasz wyłącznie skompilowane pliki *.pyc do środowiska produkcyjnego), to proces generowania grafu zakończy się niepowodzeniem lub sam graf będzie miał ograniczoną funkcjonalność.
- TensorFlow wychwytuje wyłącznie pętle for przetwarzające tensor lub zestaw danych. Zastępuj więc konstrukcję `for i in range(x)` konstrukcją `for i in tf.range(x)`, gdyż w przeciwnym razie pętla nie zostanie uwzględniona w grafie. Będzie za to przetwarzana w fazie kreślenia (być może właśnie tego chcesz, jeżeli pętla for ma służyć do budowania grafu, na przykład w celu tworzenia każdej warstwy sieci neuronowej).
- Jak zwykle z powodów wydajnościowych należy zawsze wybierać implementację wektoryzowaną zamiast pętli.

Czas na podsumowanie! Rozdział rozpoczęliśmy od ogólnego opisu modułu TensorFlow, następnie przyjrzaliśmy się jego ogólnemu API, w tym poszczególnym elementom, takim jak tensory, operacje, zmienne i specjalne struktury danych. Wykorzystaliśmy te narzędzia do zmodyfikowania niemal każdego elementu w interfejsie `tf.keras`. Na koniec przekonaliśmy się, że funkcje TF mogą poprawić wydajność modelu, a także poznaliśmy mechanizm generowania grafów za pomocą narzędzi AutoGraph i kreślenia. Przyjrzaliśmy się również regułom, jakich należy przestrzegać, pisząc funkcje TF (jeżeli chcesz uzyskać więcej informacji, aby na przykład przeglądać wygenerowane grafy, to szczegóły techniczne znajdziesz w dodatku G).

W następnym rozdziale nauczysz się wydajnie wczytywać i wstępnie przetwarzać dane w module TensorFlow.

Ćwiczenia

1. Opisz moduł TensorFlow w jednym zdaniu. Jakie są główne cechy tego modułu? Wymień inne popularne biblioteki uczenia głębokiego.
2. Czy moduł TensorFlow bezpośrednio zastępuje bibliotekę NumPy? Jakią różnicę występują pomiędzy nimi?
3. Czy funkcje `tf.range(10)` i `tf.constant(np.arange(10))` dają takie same wyniki?
4. Wymień sześć innych struktur danych (oprócz standardowych tensorów) dostępnych w module TensorFlow.
5. Możesz zdefiniować niestandardową funkcję straty przez napisanie funkcji lub utworzenie podklasy klasy `keras.losses.Loss`. Kiedy należy wykorzystać każdą z tych opcji?
6. Możesz zdefiniować niestandardowy wskaźnik wewnętrz funkcji lub podklasy klasy `keras.metrics.Metric`. Kiedy należy wykorzystać każdą z tych opcji?
7. Kiedy należy tworzyć niestandardową warstwę, a kiedy niestandardowy model?
8. W jakich sytuacjach może być wymagane utworzenie niestandardowej pętli uczenia?
9. Czy niestandardowe składniki interfejsu Keras mogą zawierać dowolny kod języka Python, czy też muszą być przekształcalne w funkcje TF?

- 10.** Jakich głównych zasad należy przestrzegać, aby funkcja była przekształcalna w funkcję TF?
- 11.** Kiedy należy tworzyć dynamiczne modele Keras? Jak to zrobić? Dlaczego nie wszystkie modele są dynamiczne?
- 12.** Zaimplementuj niestandardową warstwę realizującą **normalizację warstwy** (ang. *layer normalization*; wykorzystamy tego typu warstwę w rozdziale 15.):
- Metoda `build()` powinna definiować dwie modyfikowalne wagi, α i β , obydwie o wymiarach `input_shape[-1:]` i typie danych `tf.float32`. Waga α powinna być inicjalizowana z jedynkami, a β z zerami.
 - Metoda `call()` powinna obliczać średnią μ i odchylenie standardowe σ cech każdego przykładu. Możesz w tym celu użyć konstrukcji `tf.nn.moments(inputs, axes=-1, keepdims=True)`, która zwraca średnią μ i wariancję σ^2 wszystkich przykładów (aby uzyskać odchylenie standardowe, spierwiastkuj wariancję). Następnie funkcja powinna obliczać i zwracać działanie $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$, gdzie \otimes oznacza mnożenie (*) poszczególnych elementów, natomiast ϵ to człon wygładzający (mała wartość stała, np. 0,001, zapobiegająca dzieleniu przez 0).
 - Spraw, aby każda niestandardowa warstwa dawała taki sam (lub prawie taki sam) wynik jak warstwa `keras.layers.LayerNormalization`.
- 13.** Wytrenuj model przetwarzający zestaw danych Fashion MNIST (zob. rozdział 10.) za pomocą niestandardowej pętli uczenia:
- Wyświetl epokę, iterację, średnią funkcję straty uczenia i średnią dokładność w każdej epoce (aktualizowaną w każdej iteracji), a także funkcję straty walidacji i dokładność na końcu każdej epoki.
 - Wypróbuj różne optymalizatory i różne współczynniki uczenia, zarówno w warstwach górnych, jak i dolnych.

Rozwiązania ćwiczeń znajdziesz w dodatku A.

Wczytywanie i wstępne przetwarzanie danych za pomocą modułu TensorFlow

Dotychczas korzystaliśmy wyłącznie z zestawów danych, które mieszczą się w pamięci, ale systemy uczenia głębokiego często są trenowane na olbrzymich zestawach danych, niemieszczących się w pamięci RAM. W innych bibliotekach uczenia głębokiego wprowadzanie i wstępne przetwarzanie dużych zestawów danych może być kłopotliwe, ale moduł TensorFlow nie ma z tym problemu dzięki obecności **interfejsu danych (Data API)**: wystarczy utworzyć obiekt zestawu danych, a następnie wskazać mu dane i wyznaczyć mechanizm ich przekształcania. Moduł TensorFlow bierze na siebie szczególne implementacje, takie jak wielowątkowość, kolejkowanie, tworzenie grup danych czy pobieranie wstępne. Ponadto interfejs danych działa bezproblemowo z interfejsem `tf.keras`!

Standardowo interfejs danych może odczytywać pliki tekstowe (np. pliki CSV), pliki binarne z rekordami o stałym rozmiarze, a także pliki binarne wykorzystujące specyficzny dla modułu TensorFlow format TFRecord, obsługujący rekordy o zmiennych rozmiarach. TFRecord jest elastycznym i wydajnym formatem binarnym, który zazwyczaj zawiera tzw. bufory protokołu (ang. *protocol buffers*; format binarny o otwartym kodzie). Interfejs danych obsługuje również odczyt z baz danych SQL. Co więcej, dostępnych jest wiele rozszerzeń o otwartym kodzie, które pozwalają na odczyt z różnorodnych źródeł danych, np. z usługi BigQuery firmy Google.

Skuteczny odczyt rozbudowanych zestawów danych nie stanowi jedynej trudności, dane bowiem muszą być również wstępnie przetwarzane (najczęściej normalizowane). Ponadto nie zawsze składają się one wyłącznie z wygodnych pól numerycznych — mogą występować cechy tekstowe, kategoryalne itd. Należy je zakodować, na przykład za pomocą kodowania gorącojedynkowego, kodowania „worka słów” lub **wektorów właściwościowych** (ang. *embeddings*; jak się przekonasz, są to modyfikowalne wektory gęste reprezentujące kategorię albo token). Jedną z możliwości obsługi całokształtu przetwarzania wstępnego jest utworzenie własnych, niestandardowych warstw realizujących to zadanie. Innym rozwiązaniem jest wykorzystanie gotowych warstw dostępnych w interfejsie Keras.

W niniejszym rozdziale przyjrzymy się interfejsowi danych, formatowi TFRecord, a także sposobom tworzenia niestandardowych warstw wstępnego przetwarzania oraz wykorzystania ich standardowych odpowiedników, które stanowią część interfejsu Keras. Przyjrzymy się również побieżnie kilku powiązanym projektom:

TF Transform (tf.Transform)

Umożliwia pisanie pojedynczych funkcji przetwarzania wstępnego, które można uruchamiać w trybie wsadowym na pełnym zestawie danych uczących jeszcze przed rozpoczęciem treningu (w celu jego przyspieszenia); następnie można eksportować je do funkcji TF i dodać do wyuczonego modelu, dzięki czemu po wdrożeniu do środowiska produkcyjnego nowe przykłady będą na bieżąco przetwarzane wstępnie.

TF Datasets (TFDS)

Zawiera wygodną funkcję umożliwiającą pobieranie najróżniejszych zestawów danych, w tym tak dużych jak np. ImageNet, a ponadto przechowuje przydatne obiekty zestawów danych, które można przetwarzać poprzez interfejs danych.

Przejdźmy do rzeczy!

Interfejs danych

Cała filozofia interfejsu danych oparta jest na koncepcji **zestawu danych** — jak łatwo się domyślić, symbolizuje on sekwencję elementów danych. Zazwyczaj będziesz korzystać z zestawów danych stopniowo odczytujących dane z dysku, ale dla uproszczenia utwórzmy za pomocą funkcji `tf.data.Dataset.from_tensor_slices()` zestaw danych przechowywany całkowicie w pamięci operacyjnej:

```
>>> X = tf.range(10) # Dowolny tensor danych
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

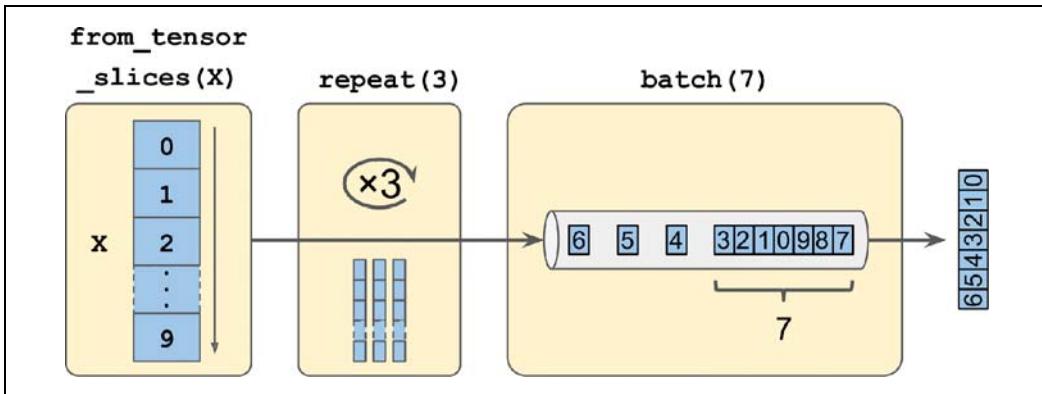
Funkcja `from_tensor_slices()` przyjmuje tensor i tworzy obiekt `tf.data.Dataset`, którego wszystkie elementy są fragmentami zestawu danych X (wzdłuż pierwszego wymiaru), zatem nasz zestaw danych zawiera 10 elementów: tensory 0, 1, 2, ..., 9. W tym przypadku uzyskalibyśmy taki sam zestaw danych, gdybyśmy użyli funkcji `tf.data.Dataset.range(10)`.

Możesz iterować po elementach zestawu danych w następujący sposób:

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

Łączenie przekształceń

Po przygotowaniu zestawu danych możesz poddać go najróżniejszym przekształceniom poprzez wywołanie jego metod transformujących. Każda taka metoda zwraca nowy zestaw danych, więc możesz łączyć przekształcenia w następujący sposób (ten łańcuch transformacji został pokazany na rysunku 13.1):



Rysunek 13.1. Łączenie przekształceń zestawów danych

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

W omawianym przykładzie wywołujemy najpierw metodę `repeat()` na pierwotnym zestawie danych, co powoduje zwrócenie nowego zestawu danych, w którym elementy zestawu pierwotnego zostają trzykrotnie powielone. Oczywiście nie oznacza to, że elementy te zostaną potrojone w pamięci operacyjnej (jeżeli wywołasz tę metodę bez argumentów, nowy zestaw danych będzie w nieskończoność powiełał elementy zestawu oryginalnego, dlatego kod iterujący po zestawie danych musiał zadecydować, kiedy przestać)! Następnie wywołujemy metodę `batch()` na nowym zestawie danych, co spowoduje utworzenie kolejnego zestawu danych. W tym przypadku elementy poprzedniego zestawu danych zostaną pogrupowane w zbiory siedmioelementowe. Na końcu iterujemy po elementach tego ostatniego zestawu danych. Jak widać, metoda `batch()` musiała utworzyć ostatnią grupę zawierającą dwa elementy, ale możesz ją pominąć za pomocą parametru `drop_remainder=True`, jeżeli chcesz, aby wszystkie grupy miały dokładnie taki sam rozmiar.



Metody zestawu danych *nie* modyfikują zestawów danych, lecz tworzą nowe, dlatego pamiętaj, aby odnosić się do tych nowych zestawów danych (np. za pomocą konstrukcji `dataset = ...`), gdyż w przeciwnym wypadku nie uzyskasz żadnych rezultatów.

Możesz również przekształcać elementy poprzez wywołanie metody `map()`. Na przykład ten kod generuje nowy zestaw danych zawierający podwojone wartości wszystkich elementów:

```
>>> dataset = dataset.map(lambda x: x * 2) # Elementy: [0,2,4,6,8,10,12]
```

Tę właśnie metodę będziesz wywoływać w celu przeprowadzania dowolnego typu wstępnego przetwarzania danych. Czasami obliczenia takie są skomplikowane, np. zmiana wymiarów lub obracanie obrazu, dlatego aby je przyspieszyć, wyznaczamy wiele wątków, co sprowadza się do wyznaczenia

argumentu `num_parallel_calls`. Zwróć uwagę, że funkcja przekazywana metodzie `map()` musi być przekształcalna do postaci funkcji TF (zob. rozdział 12.).

Metoda `map()` stosuje dane przekształcenie do każdego elementu, natomiast metoda `apply()` transformuje cały zestaw danych. Na przykład w poniższym listingu stosujemy funkcję `unbatch()` na zestawie danych (funkcja ta znajduje się obecnie w fazie eksperymentalnej, ale najprawdopodobniej zostanie przeniesiona do interfejsu głównego w którymś z przyszłych wydań modułu TensorFlow). Każdy element nowego zestawu danych będzie tensorem jednoelementowym, a nie grupą siedmiu liczb stałoprzecinkowych:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch()) # Elementy: 0,2,4,...
```

Możliwe jest także filtrowanie zestawu danych za pomocą metody `filter()`:

```
>>> dataset = dataset.filter(lambda x: x < 10) # Elementy: 0 2 4 6 8 0 2 4 6...
```

Często chcemy zerknąć na zaledwie kilka elementów zestawu danych. W takiej sytuacji należy użyć metody `take()`:

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

Tasowanie danych

Jak wiadomo, algorytm gradientu prostego sprawdza się najlepiej, gdy przykłady w zbiorze danych uczących są zmiennymi niezależnymi o tym samym rozkładzie (zob. rozdział 4.). Prostym rozwiązaniem jest przetasowanie przykładów za pomocą metody `shuffle()`. Zostanie utworzony nowy zestaw danych, którego bufor będzie zapelniany pierwszymi elementami źródłowego zestawu danych. Następnie za każdym razem, gdy trzeba będzie przekazać jakiś element z tego zestawu danych, zostanie wylosowany jakiś z bufora i zastąpiony innym elementem ze źródłowego zestawu danych aż do przetworzenia w ten sposób całego źródłowego zestawu danych. Od tego momentu elementy będą dalej losowane z bufora aż do jego opróżnienia. Musisz określić jego rozmiar, przy czym jest bardzo ważne, aby bufor był wystarczająco duży, gdyż w przeciwnym wypadku tasowanie nie będzie zbyt skuteczne¹. Nie przekraczaj tylko pojemności pamięci operacyjnej, a nawet jeżeli masz jej bardzo dużo, nie ma potrzeby wyznaczać rozmiaru większego od zestawu danych. Jeśli chcesz uzyskiwać po każdym uruchomieniu programu tę samą sekwencję losowanych elementów, możesz wyznaczyć ziarno losowości. Na przykład poniższy kod tworzy i wyświetla zestaw danych składający się z liczb od 0 do 9, powielony trzy razy, potasowany przy użyciu bufora o rozmiarze 5 i ziarna losowości równego 42, a następnie podzielony na siedmioelementowe grupy:

¹ Wyobraź sobie uporządkowaną talię kart umieszczoną po lewej stronie. Założmy, że bierzesz tylko trzy karty z wierzchu i je tasujesz, następnie wybierasz losowo jedną z nich i umieszczasz ją po prawej stronie, a dwie pozostałe zatrzymujesz w ręce. Dobierasz kolejną kartę z lewej, tasujesz trzy karty i losujesz jedną z nich, którą znów umieszczasz w stosie po prawej. Gdy powtarzysz to ze wszystkimi kartami, po prawej stronie będziesz mieć pełną talię. Myślisz, że jest ona doskonale potasowana?

```

>>> dataset = tf.data.Dataset.range(10).repeat(3) # Wartości od 0 do 9, powielone trzy razy
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)

```



Jeżeli wywołasz funkcję `repeat()` na potasowanym zestawie danych, domyślnie w każdej iteracji zostanie wygenerowana nowa kolejność elementów. Zazwyczaj jest to dobre rozwiązanie, jeżeli jednak wolisz mieć taką samą kolejność w poszczególnych iteracjach (np. podczas testowania lub usuwania błędów), to możesz wyznaczyć parametr `reshuffle_each_iteration=False`.

Jeżeli duży zestaw danych nie mieści się w pamięci, to takie proste tasowanie przy użyciu bufora może nie wystarczyć, ponieważ bufor będzie mały w stosunku do zestawu danych. Jednym z rozwiązań jest bezpośrednie przetasowanie danych źródłowych (przykładowo w Linuksie możesz tasować pliki tekstowe za pomocą polecenia `shuf`). W ten sposób zdecydowanie usprawnisz tasowanie! Nawet jeżeli dane źródłowe zostały już przetasowane, przeważnie warto je jeszcze bardziej potasować, gdyż ta sama kolejność będzie występować w następnej epoce, a model może stać się obciążony (np. w wyniku jakichś fałszywych wzorców wynikających przypadkowo z kolejnością danych źródłowych). Aby dodatkowo przetasować przykłady, często rozdzielimy dane źródłowe na wiele plików, które następnie są losowo odczytywane w fazie uczenia. Jednak przykłady mieszczące się w jednym pliku będą występować względnie blisko siebie po tasowaniu. Możemy tego uniknąć poprzez losowe dobranie wielu plików i ich jednoczesne odczytywanie, co pozwala przeplatać występujące w nich rekordy. Do tego możesz jeszcze dodać bufor tasujący za pomocą metody `shuffle()`. Jeżeli brzmi to jak mnóstwo pracy, nie musisz się obawiać, bo interfejs danych umożliwia przeprowadzenie wszystkich tych czynności poprzez zdefiniowanie zaledwie kilku wierszy kodu. Zobaczmy teraz, jak to zrobić.

Przeplatanie wierszy z różnych plików

Najpierw założmy, że wczytaliśmy zestaw danych California Housing, przetasowaliśmy go (jeżeli jeszcze nie został potasowany) i podzieliliśmy na zbiory uczący, walidacyjny i testowy. Potem dzielisz każdy z tych zbiorów na wiele plików CSV, wyglądających następująco (każdy wiersz zawiera osiem cech wejściowych i docelową medianę wartości domu):

```

MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,MedianHouseValue
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621
[...]

```

Założymy także, że zmienna `train_filepaths` przechowuje listę ścieżek do plików zbioru uczącego (podobnie w przypadku zmiennych `valid_filepaths` dla zbioru walidacyjnego i `test_filepaths` dla zbioru testowego):

```
>>> train_filepaths
['zestawy danych/housing/zbior_uczacy_00.csv', 'zestawy
danych/housing/zbior_uczacy_01.csv',...]
```

Możesz ewentualnie wykorzystać wzorce plików, na przykład `train_filepaths = "zestawy danych/housing/zbior_uczacy_*.csv"`. Utwórzmy teraz zestaw danych zawierający wyłącznie ścieżki do tych plików:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

Funkcja `list_files()` zwraca domyślnie zestaw danych z potasowanymi ścieżkami plików. Zasadniczo jest to dobre rozwiązanie, jeżeli jednak z jakiegoś powodu nie chcesz, aby były one tasowane, możesz wprowadzić parametr `shuffle=False`.

Teraz możesz wywołać metodę `interleave()`, która odczytuje pięć plików jednocześnie i przeplata zawarte w nich wiersze (za pomocą metody `skip()` pomijamy pierwszy wiersz w każdym pliku, gdyż jest to wiersz nagłówka):

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

Metoda `interleave()` utworzy zestaw danych, wydobędzie pięć ścieżek plików z `filepath_dataset` i dla każdego z nich wywoła wyznaczoną funkcję (w tym przykładzie `lambda`), generującą nowy zestaw danych (tutaj `TextLineDataset`). Dla jasności: w tej fazie będzie istnieć siedem zestawów danych, a mianowicie zestaw zawierający ścieżki plików, zestaw danych przeplotu i pięć zestawów `TextLineDataset` utworzonych przez zestaw danych przeplotu. Podczas przetwarzania zestawu danych przeplotu będzie odczytywana cyklicznie zawartość pięciu zestawów `TextLineDataset`, po jednym wierszu, aż do opróżnienia wszystkich tych zestawów danych. Potem zostanie pobranych pięć kolejnych ścieżek z zestawu `filepath_dataset` i cały proces będzie powtarzany aż do wyczerpania wszystkich ścieżek.



Aby operacja przeplatania była maksymalnie wydajna, pliki powinny być identycznej długości, gdyż w przeciwnym wypadku końcówki najdłuższych plików będą pomijane.

Metoda `interleave()` nie korzysta domyślnie ze zrównoleglania — odczytuje sekwencyjnie po jednym wierszu z każdego pliku. Jeżeli wolisz, żeby pliki były odczytywane równolegle, możesz wyznaczyć argument `num_parallel_calls` z określona przez siebie liczbą wątków (metoda `map()` również akceptuje ten argument). Możesz nawet wyznaczyć `tf.data.experimental.AUTOTUNE` i pozwolić modułowi TensorFlow, aby dynamicznie dobierał liczbę wątków na podstawie dostępnych rdzeni procesora (obecnie jest to eksperymentalna funkcja). Zobaczmy, jak teraz wygląda nasz zestaw danych:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
```

```
b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'  
b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

Są to pierwsze rzędy (bez uwzględniania nagłówków) pięciu plików CSV w kolejności losowej. Wygląda nieźle! Jak jednak widać, mamy tu do czynienia jedynie z bajtowymi łańcuchami znaków — musimy przeprowadzić ich analizę składniową i przeskalać dane.

Wstępne przetwarzanie danych

Zaimplementujmy małą funkcję przeprowadzającą wspomniane operacje:

```
X_mean, X_std = [...] # Średnia i skala każdej cechy w zbiorze uczącym  
n_inputs = 8  
  
def preprocess(line):  
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]  
    fields = tf.io.decode_csv(line, record_defaults=defs)  
    x = tf.stack(fields[:-1])  
    y = tf.stack(fields[-1:])  
    return (x - X_mean) / X_std, y
```

Przeanalizujmy ten listing:

- Najpierw program zakłada tu, że obliczyliśmy średnią i odchylenie standardowe każdej cechy w zbiorze danych uczących. `X_mean` i `X_std` to zaledwie tensory jednowymiarowe (lub tablice NumPy), zawierające osiem wartości zmiennoprzecinkowych, po jednej na każdą cechę wejściową.
- Funkcja `preprocess()` przyjmuje jeden wiersz w formacie CSV i przeprowadza jego analizę składniową. Wykorzystuje w tym celu funkcję `tf.io.decode_csv()`, która przyjmuje dwa argumenty: pierwszym jest analizowany wiersz, a drugi to tabela zawierająca wartości domyślne dla każdej kolumny w pliku CSV. Dzięki tej tabeli TensorFlow zna nie tylko domyślne wartości każdej kolumny, lecz również liczbę i typy kolumn. W tym przykładzie informujemy TensorFlow, że wszystkie kolumny cech są typu zmiennoprzecinkowego, a brakujące wartości powinny domyślnie być uzupełnione zerami, ale w ostatniej kolumnie (docelowej) wyznaczamy pustą tablicę typu `tf.float32` — moduł TensorFlow wie teraz, że kolumna ta zawiera wartości zmiennoprzecinkowe, ale nie występują w niej żadne wartości, dlatego w przypadku natrafienia na brakującą wartość zostanie wyświetlony komunikat o wyjątku.
- Funkcja `decode_csv()` zwraca listę skalarów (po jednym na kolumnę), ale potrzebujemy jednowymiarowych tablic tensorowych. Dlatego wywołujemy metodę `tf.stack()` dla wszystkich tensorów oprócz ostatniego (docelowego) — w ten sposób tensory te zostaną umieszczone w jednowymiarowej tablicy. Robimy następnie to samo dla wartości docelowej (uzyskamy jednowymiarową tablicę tensorową przechowującą jedną wartość i pozbędziemy się skalar).
- Na koniec skalujemy cechy wejściowe poprzez odjęcie od nich średnich, a potem podzielenie przez odchylenia standardowe — zostaje zwrócona krotka zawierająca przeskalonecechy i zmienią docelową.

Przetestujmy tę funkcję wstępnego przetwarzania danych:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782')  
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=
```

```
array([ 0.16579159, 1.216324 , -0.05204564, -0.39215982, -0.5277444 ,
-0.2633488 , 0.8543046 , -1.3072058 ], dtype=float32)>,
<tf.Tensor: [...], numpy=array([2.782], dtype=float32)>
```

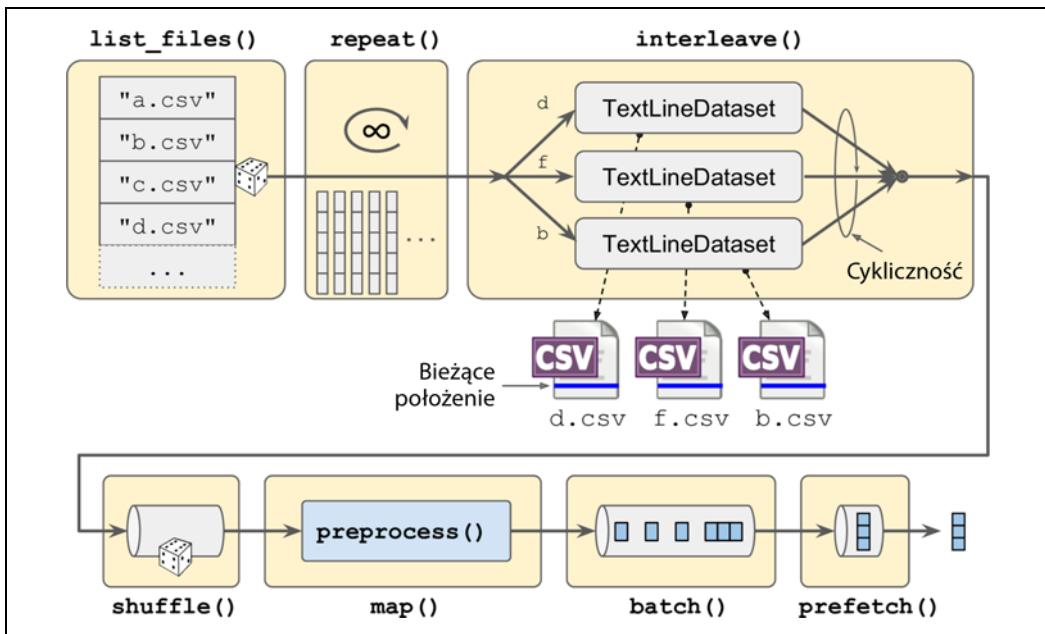
Wygląda to całkiem dobrze! Możemy teraz użyć tej funkcji na zestawie danych.

Składanie wszystkiego w całość

Jeżeli kod ma być wielokrotnie używany, musimy poskładać wszystkie omówione powyżej elementy w małą funkcję — będzie ona tworzyć i zwracać zestaw danych, do którego będą skutecznie wczytywane dane California Housing z wielu plików CSV, potem wstępnie je przetwarzać, tasować, dodatkowo zwielokrotniać i łączyć w grupy (rysunek 13.2):

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                      n_read_threads=None, shuffle_buffer_size=10000,
                      n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    return dataset.batch(batch_size).prefetch(1)
```

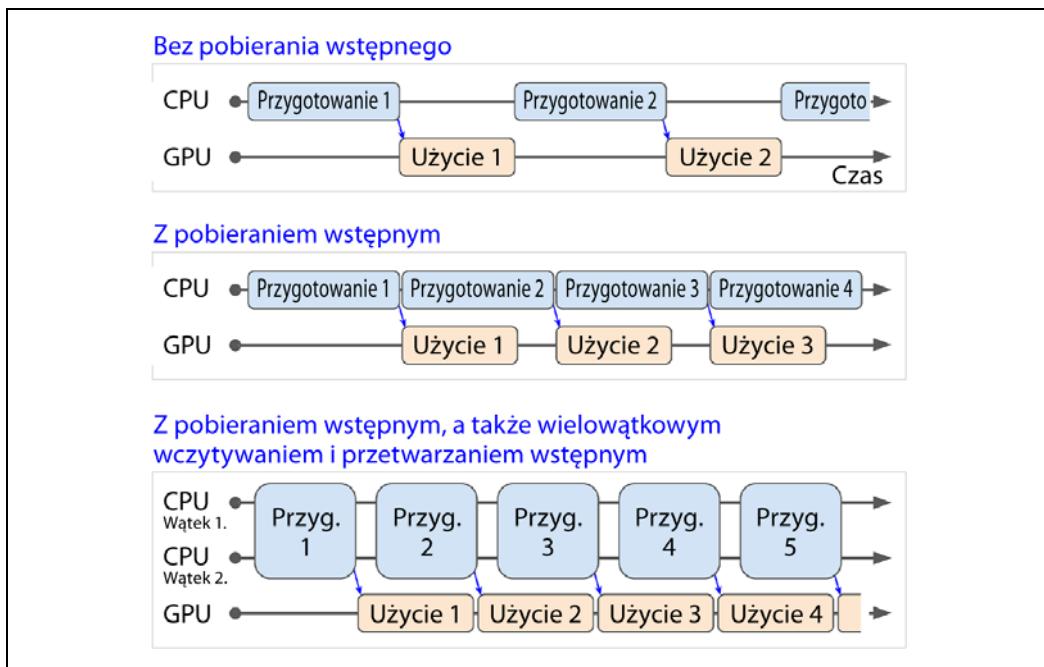
Listing ten powinien być zrozumiały, może oprócz ostatniego wiersza (prefetch(1)), który ma ważne znaczenie dla wydajności.



Rysunek 13.2. Wczytywanie i wstępne przetwarzanie danych z wielu plików CSV

Pobieranie wstępne

Wyołując funkcję `prefetch(1)` na końcu kodu, tworzymy zestaw danych, który zawsze będzie przechowywał jedną grupę z wyprzedzeniem². Inaczej mówiąc, podczas gdy algorytm uczenia przetwarza jedną grupę, zestaw danych będzie równolegle przygotowywał (tzn. odczytywał z dysku i wstępnie przetwarzał dane) kolejną grupę. Może to znacznie poprawić skuteczność modelu, co można dostrzec na rysunku 13.3. Jeżeli dodatkowo sprawimy, że wczytywanie i wstępne przetwarzanie danych będą operacjami wielowątkowymi (poprzez wyznaczenie `num_parallel_calls` w czasie wywoływanego metody `interleave()` i `map()`), możemy wykorzystywać wiele rdzeni procesora i, w idealnym przypadku, przygotowywać grupę danych szybciej, niż przebiega uczenie na karcie graficznej — w ten sposób procesor graficzny będzie wykorzystywany niemal w stu procentach (bez uwzględnienia czasu przesyłu informacji z procesora głównego do układu GPU³), a proces uczenia zostanie bardzo przyspieszony.



Rysunek 13.3. Dzięki pobieraniu wstępemu układy CPU i GPU działają równolegle: karta graficzna przetwarza jedną grupę danych, a procesor główny zajmuje się drugą grupą

² Zasadniczo wystarczy wstępne pobieranie jednej grupy, ale w niektórych sytuacjach może zaistnieć potrzeba przygotowania kilku dodatkowych grup. Ewentualnie możesz pozostawić decyzję modułowi TensorFlow poprzez przekazanie parametru `tf.data.experimental.AUTOTUNE` (obecnie znajduje się on w fazie eksperymentalnej).

³ Sprawdź jednak funkcję `tf.data.experimental.prefetch_to_device()`, która może przesyłać dane bezpośrednio do procesora graficznego.



Jeżeli planujesz zakup karty graficznej, należy oczywiście wziąć pod uwagę jej moc obliczeniową i rozmiar pamięci (duża ilość pamięci RAM jest kluczowa zwłaszcza w widżeniu komputerowym). Równie istotna dla osiągnięcia dobrej wydajności jest **przepustowość pamięci** (ang. *memory bandwidth*), czyli liczba gigabajtów danych wchodzących do pamięci RAM i z niej wychodzących.

Jeżeli zestaw danych jest wystarczająco mały, aby zmieścił się w pamięci, możesz drastycznie przyspieszyć proces uczenia za pomocą metody `cache()`, która umieszcza zawartość w pamięci podręcznej. Należy to robić zazwyczaj po wczytaniu i wstępnym przetworzeniu danych, ale jeszcze przed tasowaniem, powielaniem, dzieleniem na grupy i pobieraniem wstępny. W ten sposób każdy przykład będzie tylko raz odczytywany i wstępnie przetwarzany (a nie raz na epokę), jednak dane będą w każdej epoce inaczej przetasowane, a kolejna grupa będzie przygotowywana nadal z wyprzedzeniem.

Potrafisz już tworzyć wydajne potoki danych wejściowych, pozwalające wczytywać i wstępnie przetwarzać dane z wielu plików tekstowych naraz. Omówiłem najczęściej spotykane metody zestawów danych, ale istnieją także inne, o których warto pamiętać: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()` i `padded_batch()`. Dostępne są również dwie dodatkowe metody klasy, `from_generator()` i `from_tensors()`, które pozwalają utworzyć nowy zestaw danych za pomocą, odpowiednio, generatora liczb losowych i listy tensorów. Więcej szczegółów znajdziesz w dokumentacji API. Zwróć też uwagę, że w module `tf.data.experimental` mamy dostęp do eksperymentalnych funkcji, z których prawdopodobnie wiele trafi do kolejnych wersji podstawowego interfejsu (sprawdź na przykład klasę `CsvDataset` i metodę `make_csv_dataset()`, które służą do określania typu każdej kolumny).

Stosowanie zestawu danych z interfejsem tf.keras

Teraz możemy użyć funkcji `csv_reader_dataset()` do utworzenia zestawu danych dla zbioru uczącego. Nie musimy powtarzać tego procesu, gdyż zajmie się nim interfejs `tf.keras`. Utworzmy również zestawy danych dla zbiorów walidacyjnego i testowego:

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

Teraz wystarczy zbudować i wytrenować model Keras wykorzystujący te zestawy danych⁴. Wystarczy przekazać metodzie `fit()` zestawy danych uczący i walidacyjny zamiast obiektów `X_train`, `y_train`, `X_valid` oraz `y_valid`⁵:

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

⁴ Obsługa zestawów danych jest charakterystyczna dla interfejsu `tf.keras`; omawiane rozwiązanie nie zadziała w innych implementacjach interfejsu Keras.

⁵ Metoda `fit()` zajmie się powtarzaniem zestawu danych. Można ewentualnie wywołać metodę `repeat()` na zestawie danych uczących, dzięki czemu będzie on powtarzany w nieskończoność, a także wyznaczyć argument `steps_per_epoch` podczas wywoływanego metody `fit()`. Rozwiążanie to może się przydawać w pewnych sytuacjach, na przykład jeśli chcesz korzystać z bufora tasowania współdzielonego pomiędzy epokami.

W podobny sposób możemy przekazać zestaw danych do metod `evaluate()` i `predict()`:

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) #Udajemy, że mamy trzy nowe przykłady
model.predict(new_set) #Zestaw danych zawierający nowe przykłady
```

W przeciwieństwie do pozostałych zestawów, `new_set` zazwyczaj nie będzie zawierał etykiet (a nawet jeśli będzie je zawierał, zostaną zignorowane przez Keras). Zwróć uwagę, że w każdym przypadku możesz w razie potrzeby zastąpić zestawy danych tablicami NumPy (oczywiście muszą być one najpierw wczytane i wstępnie przetworzone).

Jeżeli chcesz utworzyć własną, niestandardową pętlę uczenia (tak jak zrobiliśmy to w rozdziale 12.), wystarczy iterować po zbiorze uczącym, co wychodzi bardzo naturalnie:

```
for X_batch, y_batch in train_set:
    [...] # Wykonuje jeden przebieg gradientu prostego
```

W istocie możliwe jest nawet utworzenie funkcji TF (zob. rozdział 12.), realizującej całą pętlę uczenia:

```
@tf.function
def train(model, optimizer, loss_fn, n_epochs, [...]):
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs, [...])
    for X_batch, y_batch in train_set:
        with tf.GradientTape() as tape:
            y_pred = model(X_batch)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        grads = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gratulacje, umiesz już budować potężne potoki danych wejściowych za pomocą interfejsu danych! Do tej pory korzystaliśmy jednak z plików CSV, które są powszechnie, proste i wygodne, ale niezbyt wydajne i nie obsługują zbyt dobrze dużych i skomplikowanych struktur danych (takich jak obrazy czy dźwięki). Dlatego zajmiemy się teraz formatem TFRecord.



Jeżeli jesteś zadowolona/zadowolony z plików CSV (lub dowolnego innego formatu, który wykorzystujesz), nie musisz przestawać się na format TFRecord. Zgodnie z popularnym powiedzeniem: jeśli coś działa, nie próbuj tego naprawiać! Format TFRecord przydaje się wtedy, gdy przyczyną zatorów jest wczytywanie i analiza składowi danych.

Format TFRecord

Format TFRecord stanowi preferowane rozwiązanie, służące do przechowywania i skutecznego odczytywania dużych ilości danych. Jest to bardzo prosty format binarny, który przechowuje jedynie sekwencję rekordów binarnych o różnych długościach (każdy rekord składa się z długości, sumy kontrolnej CRC, która sprawdza, czy długość jest prawidłowa, właściwych danych oraz sumy kontrolnej tych danych). Możesz z łatwością utworzyć plik TFRecord za pomocą klasy `tf.io.TFRecordWriter`:

```
with tf.io.TFRecordWriter("moje_dane.tfrecord") as f:
    f.write(b"To jest pierwszy rekord.")
    f.write(b"A to jest drugi rekord.")
```

Możesz następnie użyć `tf.data.TFRecordDataset` do odczytania co najmniej jednego pliku TFRecord:

```
filepaths = ["moje_dane.tfrecord"]
dataset = tf.data.TFRecordDataset(filepaths)
for item in dataset:
    print(item)
```

Uzyskamy następujący rezultat:

```
tf.Tensor(b' To jest pierwszy rekord.', shape=(), dtype=string)
tf.Tensor(b' A to jest drugi rekord.', shape=(), dtype=string)
```



Klasa `TFRecordDataset` będzie odczytywała pliki pojedynczo, ale może odczytywać wiele plików równolegle i przeplatać ich rekordy poprzez wyznaczenie parametru `num_parallel_reads`. Możesz ewentualnie uzyskać ten sam rezultat, używając metod `list_files()` i `interleave()` w sposób opisany powyżej.

Skompresowane pliki TFRecord

Czasami warto skompresować pliki TFRecord, zwłaszcza gdy muszą być wczytywane poprzez połączenie sieciowe. Możesz utworzyć skompresowany plik TFRecord, wyznaczając argument `options`:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")
with tf.io.TFRecordWriter("plik_skompresowany.tfrecord", options) as f:
    [...]
```

Podczas odczytu skompresowanego pliku TFRecord musisz określić mechanizm kompresji:

```
dataset = tf.data.TFRecordDataset(["plik_skompresowany.tfrecord"],
                                   compression_type="GZIP")
```

Wprowadzenie do buforów protokołów

Każdy rekord może wykorzystywać dowolny wybrany przez Ciebie format binarny, ale pliki TFRecord zwykle zawierają serializowane **bufory protokołów** (ang. *buffer protocol*; w piśmiennictwie anglojęzycznym często są skrótnie nazywane **protobuf**). Jest to przenośny, rozszerzalny i wydajny format binarny zaprojektowany przez firmę Google w 2001 roku, a upubliczony w 2008 roku. Protokoły buforów są obecnie powszechnie stosowane, zwłaszcza w systemie zdalnego wywoływania procedur gRPC (<https://grpc.io/>) firmy Google. Są one definiowane za pomocą prostego języka, którego składnia wygląda następująco:

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

W powyższej definicji wyznaczamy wersję trzecią formatu bufora protokołów i pozwalamy, aby obiekt `Person`⁶ mógł (opcjonalnie) mieć nazwę (`name`) typu `string`, identyfikator (`id`) typu `int32`,

⁶ Obiekty buforów protokołów są przeznaczone do serializacji i transmitowania, dlatego nazywa się je **komunikatami** (ang. *messages*).

a także co najmniej 0 pól email, z których każde byłoby typu `string`. Wartości 1, 2 i 3 to identyfikatory pól — będą one wykorzystywane w reprezentacji binarnej każdego rekordu. Po utworzeniu definicji pliku `.proto` możesz go skompilować. W tym celu kompilator `protoc` musi wygenerować w Pythonie (lub w innym języku) klasy dostępu. Używane przez nas definicje protokołów buforów zostały już skompilowane, a ich klasy języka Python stanowią część modułu TensorFlow, dlatego nie musisz korzystać z kompilatora `protoc`. Musisz jedynie wiedzieć, jak stosować klasy dostępu tych buforów w Pythonie. Na poniższym przykładzie prezentuję podstawowe kwestie stosowania klas dostępu, wygenerowanych dla bufora `Person` (wyjaśnienie kodu znajdziesz w komentarzach):

```
>>> from person_pb2 import Person # Importuje wygenerowaną klasę dostępową
>>> person = Person(name="Al", id=123, email=["a@b.com"]) # Tworzy obiekt Person
>>> print(person) # Wyświetla obiekt Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name # Odczytuje pole
"Al"
>>> person.name = "Alicja" # Modyfikuje pole
>>> person.email[0] # Dostęp do wielokrotnie występujących pól uzyskujemy tak jak do tablic
"a@b.com"
>>> person.email.append("c@d.com") # Dodaje adres e-mail
>>> s = person.SerializeToString() # Serializuje obiekt w postaci bajtowego łańcucha znaków
>>> s
b'\n\x05Alice\x10{\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person() # Tworzy nowy obiekt Person
>>> person2.ParseFromString(s) # Analizuje składnię łańcucha znaków (ma długość 27 bajtów)
27
>>> person == person2 # Obydwa obiekty są teraz sobie równe
True
```

Mówiąc krótko, importujemy klasę `Person` wygenerowaną przez kompilator `protoc`, tworzymy jej wystąpienie i modyfikujemy je na różne sposoby, wyświetlamy jego zawartość, odczytujemy pewne pola i zapisujemy w nich dane, a następnie przeprowadzamy serializację za pomocą metody `SerializeToString()`. Otrzymujemy dane binarne przygotowane do zapisu lub transmitowania w sieci. Gdy odczytujemy lub otrzymujemy takie dane binarne, możemy przeanalizować ich składnię za pomocą metody `ParseFromString()` i otrzymujemy kopię serializowanego obiektu⁷.

Moglibyśmy zapisać serializowany obiekt `Person` w pliku `TFRecord`, a potem go wczytać i przeanalizować składnię — wszystko działałoby jak należy. Jednak metody `SerializeToString()` i `ParseFromString()` nie są operacjami TensorFlow (podobnie jak wszystkie inne operacje w powyższym listingu), zatem nie można ich umieścić w funkcji TensorFlow (chyba że wstawisz je do operacji, co jak już wiemy z rozdziału 12., spowolni kod i ograniczy jego przenośność). Na szczęście TensorFlow zawiera specjalne definicje buforów protokołów, zapewniające analizę składni.

⁷ W tym rozdziale zawarłem absolutne minimum wymagane do zrozumienia buforów protokołów i formatu `TFRecords`. Więcej informacji znajdziesz pod adresem <https://developers.google.com/protocol-buffers/>.

Bufory protokołów w module TensorFlow

Głównym buforem protokołów używanym w pliku TFRecord jest `Example`, reprezentujący jeden przykład w zestawie danych. Zawiera listę ponazywanych cech, gdzie każda cecha może być listą bajtowych łańcuchów znaków, wartości zmiennoprzecinkowych lub wartości stałoprzecinkowych. Oto definicja tego bufora:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

Definicje komunikatów `BytesList`, `FloatList` i `Int64List` są zrozumiałe. Zwróć uwagę, że zapis `[packed = true]` jest używany dla powtarzających się pól numerycznych, co zwiększa wydajność kodowania. Komunikat `Feature` zawiera `BytesList`, `FloatList` lub `Int64List`. Z kolei komunikat `Features` (z „s” na końcu) przechowuje słownik odwzorowujący nazwę cechy na odpowiednią wartość tej cechy. Poza tym bufor `Example` zawiera wyłącznie obiekt `Features`⁸. Oto jak utworzyć odpowiednik wcześniej zdefiniowanego obiektu `Person` za pomocą klasy `tf.train.Example` i zapisać go w pliku TFRecord:

```
from tensorflow.train import BytesList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alicja"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                b"c@d.com"])))
    ))
```

Kod ten jest nieco rozwlekły i powtarzalny, ale raczej prosty do zrozumienia (do tego łatwo można byłoby go umieścić w dodatkowej funkcji). Skoro mamy już bufor `Example`, możemy przeprowadzić jego serializację poprzez wywołanie metody `SerializeToString()` i zapisanie rezultatu w pliku TFRecord:

```
with tf.io.TFRecordWriter("moje_kontakty.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

⁸ Dlaczego więc bufor `Example` został zdefiniowany, skoro zawiera tylko obiekt `Features`? Twórcy modułu TensorFlow mogą kiedyś postanowić wstawić tu więcej pól. Dopóki nowe wersje definicji `Example` będą zawierać pole `features` o niezmienionym identyfikatorze, pozostanie wstępnie kompatybilny. Taka rozszerzalność stanowi jedną z największych zalet buforów protokołów.

Zazwyczaj proces twórczy nie kończy się na jednym buforze Example! Należałyby napisać skrypt konwersji, który odczytuje dane z bieżącego formatu (np. plików CSV), tworzy bufor Example dla każdego przykładu, serializuje takie bufore i zapisuje je w kilku plikach TRFRecord, a najlepiej jeszcze je tasuje w którymś momencie. Jest to dość pracochłonne, dlatego upewnij się, że nie masz innego wyjścia (być może Twój potok danych całkiem dobrze współdziała z formatem CSV).

Gdy już mamy taki ładny plik TFRecord zawierający serializowany bufor Example, spróbujmy go wczytać.

Wczytywanie i analizowanie składni obiektów Example

Do wczytania serializowanych buforów Example użyjemy ponownie `tf.data.TFRecordDataset`, a przeanalizujemy składnię każdego z nich za pomocą funkcji `tf.io.parse_single_example()`. Jest to operacja TensorFlow, więc możemy ją umieścić w funkcji TF. Wymaga ona przynajmniej dwóch argumentów: skalarnego tensora typu łańcuchowego zawierającego serializowane dane i opisu każdej cechy. Opisem takim jest słownik odzwierciedlający nazwę każdej cechy albo na deskryptor `tf.io.FixedLenFeature` określający wymiary, typ i domyślną wartość cechy, albo na deskryptor `tf.io.VarLenFeature` określający jedynie typ (jeżeli długość listy cechy może być zróżnicowana, jak w przypadku cechy "emails").

Poniższy listing definiuje słownik opisów, następnie iteruje po zestawie `TFRecordDataset`, po czym analizuje składnię serializowanego bufora Example przechowywanego w tym zestawie danych:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(["moje_kontakty.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

Cechy o stałej długości są przetwarzane jako standardowe tensorы, natomiast zmienne o zmiennej długości są przetwarzane jako tensorы rzadkie. Możesz przekształcić tensor rzadki w tensor gęsty za pomocą funkcji `tf.sparse.to_dense()`, w tym przypadku jednak łatwiej jest uzyskać po prostu dostęp do ich wartości:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array(['a@b.com', 'c@d.com'], [...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array(['a@b.com', 'c@d.com'], [...])>
```

Komunikat `BytesList` może zawierać dowolne dane binarne, włącznie z obiektami serializowanymi. Możesz na przykład użyć funkcji `tf.io.encode_jpeg()` do zakodowania obrazu w formacie JPEG i umieścić te dane binarne w komunikacie `BytesList`. Gdy później kod będzie przetwarzał plik TRFRecord, najpierw przeprowadzi analizę składni bufora Example, po czym będzie musiał wywołać funkcję `tf.io.decode_jpeg()` w celu przeanalizowania danych i uzyskania pierwotnego obrazu (ewentualnie możesz wykorzystać funkcję `tf.io.decode_image()`, która może dekodować obrazy w formatach BMP, GIF, JPEG lub PNG). Możesz również przechować dowolny tensor w komunikacie `BytesList` poprzez serializację tego tensora za pomocą funkcji `tf.io.serialize_tensor()`, a następnie

wstawienie uzyskanego bajtowego łańcucha znaków do cechy w BytesList. W trakcie analizowania składni pliku TFRecord możesz przetwarzać te dane przy użyciu funkcji `tf.io.parse_tensor()`.

Nie musisz analizować składni buforów Example jednego za drugim za pomocą funkcji `tf.io.parse_single_example()`, lecz możesz przetwarzać całe ich grupy przy użyciu funkcji `tf.io.parse_example()`:

```
dataset = tf.data.TFRecordDataset(["moje_kontakty.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)
```

Jak widać, bufor Example powinien wystarczyć w większości przypadków. Jednak zarządzanie listami list może być nieco niewygodne. Założymy na przykład, że chcesz klasyfikować dokumenty tekstowe. Każdy dokument może być reprezentowany jako lista zdań, w której każde zdanie jest reprezentowane jako lista słów. Poza tym każdy dokument może zawierać listę komentarzy, gdzie każdy komentarz jest reprezentowany jako lista słów. Mogą występować również jakieś dane kontekstowe, takie jak autor, tytuł czy data publikacji. W takich sytuacjach przydaje się bufor SequenceExample.

Obsługa list za pomocą bufora protokołów SequenceExample

Oto definicja bufora SequenceExample:

```
message FeatureList { repeated Feature feature = 1; };
message FeatureLists { map<string, FeatureList> feature_list = 1; };
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

Bufor SequenceExample zawiera obiekt Features, przeznaczony dla danych kontekstowych, i obiekt FeatureLists, przechowujący co najmniej jeden obiekt FeatureList (np. obiekt FeatureList o nazwie "content" i drugi, "comments"). Każdy obiekt FeatureList zawiera liczbę obiektów Feature, z których każdy może być listą bajtowych łańcuchów znaków, listą 64-bitowych wartości stałoprzecinkowych lub listą wartości zmienoprzecinkowych (w naszym przykładzie każdy obiekt Feature reprezentowałby zdanie lub komentarz, być może w postaci listy identyfikatorów wyrazów). Tworzenie, serializacja i analiza składniowa bufora SequenceExample przypominają tworzenie, serializację i analizę składniową bufora Example, w tym przypadku jednak musisz używać funkcji `tf.io.parse_single_sequence_example()` do przetworzenia pojedynczego bufora SequenceExample lub funkcji `tf.io.parse_sequence_example()` do przetwarzania ich grupy. Obydwie te funkcje zwracają krotkę zawierającą cechy kontekstowe (w postaci słownika) i listy cech (również jako słownik). Jeżeli listy cech zawierają sekwencje o różnych rozmiarach (tak jak w poprzednim przykładzie), być może zechcesz przekształcić je w tensory nierówne za pomocą funkcji `tf.RaggedTensor.from_sparse()` (pełny kod znajdziesz w notatniku Jupyter):

```
parsed_context, parsed_feature_lists = tf.io.parse_single_sequence_example(
    serialized_sequence_example, context_feature_descriptions,
    sequence_feature_descriptions)
parsed_content = tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Potrafisz już skutecznie przechowywać, wczytywać i analizować dane pod kątem składni, zatem kolejnym etapem jest ich przygotowanie w sposób umożliwiający wprowadzenie ich do sieci neuronowej.

Wstępne przetwarzanie cech wejściowych

Przygotowywanie danych do sieci neuronowej wymaga przekształcenia wszystkich cech w cechy numeryczne, zazwyczaj ich normalizacji itd. W szczególności jeśli dane składają się z cech kategorialnych lub tekstowych, muszą zostać przekształcone w wartości liczbowe. Można to zrobić z wyprzedzeniem, w fazie przygotowywania plików danych, za pomocą dowolnego narzędzia (np. biblioteki NumPy, pandas czy Scikit-Learn). Ewentualnie możesz na bieżąco wstępnie przetwarzać dane podczas wczytywania ich do interfejsu danych (np. za pomocą metody `map()` zestawu danych, tak jak robiliśmy to wcześniej) albo wstawić warstwę wstępnego przetwarzania bezpośrednio w modelu. Zajmijmy się tym ostatnim rozwiązaniem.

Poniżej widzimy przykład zaimplementowania warstwy standaryzacji przy użyciu warstwy Lambda. Od każdej cechy zostaje odjęta średnia i zostaje podzielona przez odchylenie standardowe (wraz z niewielkim członem wygładzającym, który pozwala uniknąć dzielenia przez zero):

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...] # Pozostałe warstwy
])
```

Nie było zbyt trudno! Być może jednak wolisz korzystać z ładnej, niestandardowej, samodzielnej warstwy (coś w stylu klasy `StandardScaler` modułu Scikit-Learn) zamiast umieszczać wszędzie zmienne globalne, takie jak `means` czy `stds`:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) / (self.stds_ + keras.backend.epsilon())
```

Zanim będzie można skorzystać z warstwy standaryzacji, trzeba ją dostosować do zestawu danych poprzez wywołanie metody `adapt()` i przekazanie jej próby danych. Dzięki temu każda cecha będzie wykorzystywała odpowiednią średnią i odpowiednie odchylenie standardowe:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

Próba ta musi być wystarczająco duża, aby reprezentowała zestaw danych, nie musi to być jednak pełen zestaw danych uczących — zasadniczo powinno wystarczyć kilkaset losowo dobranych przykładów (zależy to jednak od realizowanego zadania). Teraz możesz wykorzystać warstwę przetwarzania wstępnego jak zwykłą warstwę:

```
model = keras.Sequential()
model.add(std_layer)
[...] # Pozostała część modelu
model.compile([...])
model.fit([...])
```

Jeżeli uważasz, że interfejs Keras powinien zawierać tego typu warstwę standaryzacji, mam dla Ciebie dobre wieści: prawdopodobnie jeszcze przed wydrukowaniem niniejszej książki zostanie udostępniona warstwa keras.layers.Normalization. Będzie ona działała bardzo podobnie do niestandardowej warstwy Standardization: najpierw zostanie utworzona warstwa, potem zostanie ona dostosowana do zestawu danych poprzez przekazanie próby danych do metody adapt(), po czym będzie gotowa do użytku.

Przyjrzyjmy się teraz cechom kategorialnym. Zaczniemy od zakodowania ich jako wektorów gorącojedynkowych.

Kodowanie cech kategorialnych za pomocą wektorów gorącojedynkowych

Przyjrzyjmy się cesze ocean_proximity z omówionego w rozdziale 2. zestawu danych California Housing: jest to cecha kategorialna zawierająca pięć rodzajów wartości ("<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY" i "ISLAND"). Zanim przekażemy tę cechę do sieci neuronowej, musimy ją odpowiednio zakodować. Występuje tu niewiele kategorii, dlatego możemy skorzystać z kodowania gorącojedynkowego. W tym celu musimy najpierw rzutować każdą kategorię na wartość indeksu (od 0 do 4), w czym pomoże nam tablica przeglądowa:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

Przeanalizujmy ten kod:

- Definiujemy najpierw **wokabularz** (ang. *vocabulary*): jest to lista wszystkich możliwych kategorii.
- Następnie wyznaczamy tensor z odpowiednimi indeksami (od 0 do 4).
- Teraz tworzymy inicjalizator tablicy przeglądowej i przekazujemy mu listę kategorii wraz z przyisanymi do nich indeksami. W omawianym przykładzie dysponujemy już tymi danymi, dla tego wykorzystujemy `KeyValueTensorInitializer`; gdyby jednak kategorie były umieszczone w pliku tekstowym (po jednej w każdym wierszu), to użylibyśmy inicjalizatora `TextFileInitializer`.
- W dwóch ostatnich wierszach tworzymy tablicę przeglądową poprzez wprowadzenie inicjalizatora i określenie liczby bloków **pozawokabularzowych** (ang. *out-of-vocabulary* — oov). Jeżeli zostanie znaleziona kategoria niewystępująca w słowniku, tablica będzie w stanie obliczyć funkcję skrótu (hasz) tej kategorii i wykorzystać ją do przydzielenia tej kategorii do jednego z bloków oov. Ich indeksy występują po indeksach znanych kategorii, więc w naszym przykładzie indeksy dwóch bloków oov miałyby wartości 5 i 6.

Dlaczego korzystamy z bloków pozawokabularzowych? W przypadku dużej liczby kategorii (np. kody pocztowe, miasta, wyrazy, produkty czy użytkownicy), a także dużego rozmiaru lub znacznej zmienności zestawu danych, to uzyskanie pełnej listy kategorii może nie być zbyt wygodne. Jednym z rozwiązań jest zdefiniowanie słownika na podstawie próby danych (a nie całego zestawu danych) i dodanie jakieś liczby bloków oov przeznaczonych dla kategorii, które nie pojawiły się w tej próbie. Jeżeli spodziewałeś się, że w trakcie uczenia zostanie odkrytych wiele nieznanych kategorii, to powinnas/powinieneś przygotować dużo bloków pozawokabularzowych. Z kolei jeśli nie wyznaczyłeś

wystarczająco wiele, zaczną pojawiać się konflikty: różne kategorie będą umieszczane w tym samym bloku, więc sieć neuronowa nie będzie w stanie ich rozróżniać (a przynajmniej nie na podstawie danej cechy).

Użyjmy teraz tablicy przeglądowej do zakodowania małej grupy cech kategorialnych w wektorach gorącojedynkowych:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab) + num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

Jak widać, wartość "NEAR BAY" została przypisana do indeksu 3, nieznana kategoria "DESERT" jest przypisana do jednego z dwóch bloków oov (o indeksie 5), natomiast wartość "INLAND" została dwukrotnie przypisana do indeksu 1. Następnie użyliśmy funkcji `tf.one_hot()` do zakodowania tych indeksów w wektorach gorącojedynkowych. Zwróć uwagę, że musieliszmy podać tej funkcji całkowitą liczbę indeksów, czyli rozmiar wokabularza i liczbę bloków pozawokabularzowych. Umiesz już kodować cechy kategorialne w wektory gorącojedynkowe za pomocą modułu TensorFlow!

Podobnie jak wcześniej, umieszczenie tego mechanizmu w ładnej klasie niezależnej nie powinno sprawiać większych problemów. Jej klasa `adapt()` przyjmowałaby próbę danych i wydobywałały wszystkie zawarte w nich kategorie. Zostałaby wygenerowana tablica przeglądowa, przypisująca każdą kategorię do indeksu (w tym nieznane kategorie do bloków oov). Następnie jej metoda `call()` wykorzystałaby tę tablicę przeglądową do przypisywania wprowadzanych kategorii do odpowiednich indeksów. Dobrych wieści ciąg dalszy: prawdopodobnie w czasie, gdy będziesz czytać niniejszą książkę, w interfejsie Keras znajdziesz się klasa `keras.layers.TextVectorization`, która będzie realizować dokładnie takie samo zadanie: jej metoda `adapt()` będzie wydobywać słownik z próby danych, a metoda `call()` posłuży do przekształcenia każdej kategorii w odpowiedni indeks. Można by umieścić tę warstwę na początku modelu, po niej wstawić warstwę Lambda, realizującą funkcję `tf.one_hot()`, której zadaniem jest przekształcanie tych indeksów w wektory gorącojedynkowe.

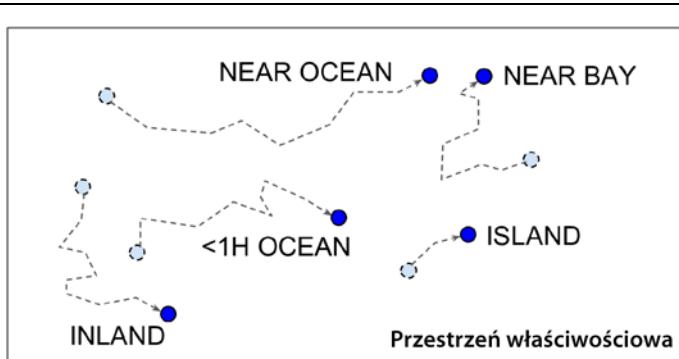
Być może nie jest to jednak najlepsze rozwiązanie. Rozmiar każdego wektora gorącojedynkowego to suma długości wokabularza i liczby bloków pozawokabularzowych. Nie ma problemu, jeśli występuje tylko kilka kategorii, ale w przypadku dużego wokabularza o wiele wydajniej jest kodować je za pomocą **wektorów właściwościowych** (ang. *embeddings*).



Zgodnie z niepisaną regułą, jeżeli występuje mniej niż 10 kategorii, to zazwyczaj kodowanie gorącojedynkowe powinno wystarczyć (ale czas działania modelu może się wydłużyć!). W przypadku liczby kategorii przekraczającej 50 (często tak jest, gdy korzystasz z bloków oov), to lepiej użyć wektorów właściwościowych. W zakresie od 10 do 50 kategorii warto wypróbować obydwa rozwiązania i sprawdzić, które będzie sprawdzało się lepiej.

Kodowanie cech kategorialnych za pomocą wektorów właściwościowych

Wektorem właściwościowym (osadzeń) nazywamy modyfikowalny wektor gęsty reprezentujący kategorię. Domyślnie wektory właściwościowe są inicjalizowane losowo, więc na przykład kategoria "NEAR BAY" mogłaby być reprezentowana początkowo przez wektor losowy, taki jak $[0,131, 0,890]$, a kategoria "NEAR OCEAN" mogłaby być reprezentowana przez inny wektor losowy, jak choćby $[0,631, 0,791]$. W tym przykładzie wykorzystujemy dwuwymiarowe wektory właściwościowe, ale liczba wymiarów jest hiperparametrem, który możemy dostosować. Wektory właściwościowe są modyfikowalne, zatem będą stopniowo uzyskiwać coraz bardziej optymalne wartości w czasie uczenia; gdy zaś reprezentują w miarę podobne kategorie (tak jak w omawianym przykładzie), algorytm gradientu prostego będzie z pewnością zbliżać je do siebie i oddalać od kategorii "INLAND" (rysunek 13.4). Rzeczywiście, im lepsza reprezentacja, tym łatwiej sieci neuronowej będzie uzyskiwać dokładne prognozy, dlatego proces uczenia dąży przeważnie do uczynienia z wektorów właściwościowych przydatnych reprezentacji kategorii. Jest to tak zwane **uczenie reprezentacji** (ang. *representation learning*; w rozdziale 17. poznasz inne rodzaje uczenia reprezentacji).



Rysunek 13.4. Wektory właściwościowe będą stopniowo udoskonalane w trakcie uczenia

Reprezentacje właściwościowe słów

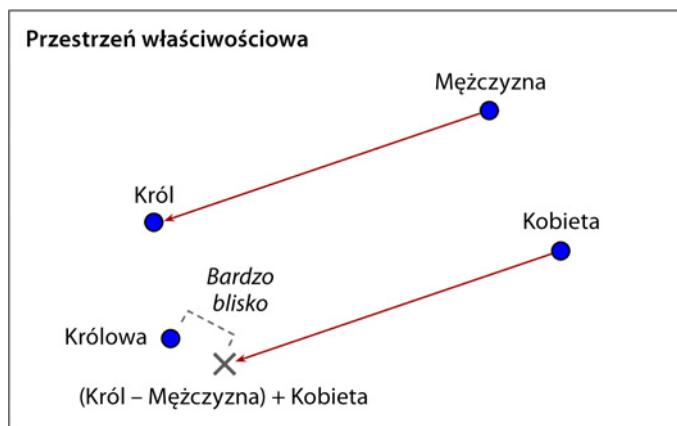
Zasadniczo wektory właściwościowe nie tylko stanowią przydatne reprezentacje w bieżącym zadaniu, ale całkiem często można je z powodzeniem wielokrotnie wykorzystywać w innych zadaniach. Najpopularniejszym przykładem takiego wielokrotnego użytkowania są **reprezentacje właściwościowe słów (osadzanie słów)** (ang. *word embeddings*; tzn. wektory właściwościowe reprezentujące pojedyncze wyrazy): jeżeli realizujesz jakieś zadanie przetwarzania języka naturalnego, to nieraz lepiej wykorzystać uprzednio wytrenowane reprezentacje właściwościowe słów zamiast uczyć własne od podstaw.

Koncepcja wykorzystywania wektorów do reprezentowania słów sięga lat 60. ubiegłego wieku i od tamtej pory wymyślono wiele zaawansowanych technik służących do generowania przydatnych wektorów, również za pomocą sieci neuronowych. Prawdziwy przełom nastąpił jednak w 2013 roku, gdy Tomáš Mikolov wraz z innymi naukowcami z firmy Google opublikował artykuł (<https://arxiv.org/abs/1310.4546>)⁹

⁹ Tomáš Mikolov i in., *Distributed Representations of Words and Phrases and Their Compositionalities*, „Proceedings of the 26th International Conference on Neural Information Processing Systems” 2 (2013), s. 3111 – 3119.

opisujący wydajną technikę poznawania reprezentacji właściwościowych słów za pomocą sieci neuronowych, znacznie przewyższającą wcześniejsze rozwiązań. Autorzy byli w stanie wyznaczyć reprezentacje właściwościowe dla bardzo dużego korpusu tekstowego: sieć neuronowa została nauczona przewidywania wyrazów występujących w pobliżu danego słowa, co pozwoliło uzyskać zdumiewające reprezentacje właściwościowe. Przykładowo synonimy mają bardzo podobne reprezentacje właściwościowe, a powiązane semantycznie wyrazy, takie jak Francja, Hiszpania czy Włochy, zostają pogrupowane w skupienia.

Bliskość wyrazów to jednak nie wszystko: reprezentacje właściwościowe słów zostają również uporządkowane wzduł osi znaczeniowych w przestrzeni właściwościowej. Wyjaśnię to na sławnym przykładzie: jeżeli obliczysz reprezentacje *Król* – *Mężczyzna* + *Kobieta* (poprzez dodawanie i odejmowanie wektorów właściwościowych symbolizujących te słowa), to rezultat będzie bardzo zbliżony do reprezentacji właściwościowej wyrazu *Królowa* (rysunek 13.5). Inaczej mówiąc, wektory właściwościowe słów kodują pojęcie płci! Na podobnej zasadzie po przeprowadzeniu operacji *Madryt* – *Hiszpania* + *Francja* powinniśmy wylądować blisko reprezentacji właściwościowej wyrazu *Paryż*, co oznaczałoby, że w wektorach właściwościowych zawarte jest także znaczenie stolic.



Rysunek 13.5. Reprezentacje właściwościowe podobnych słów zazwyczaj znajdują się blisko siebie, a pewne osie mogą kodować pojęcia znaczeniowe

Niestety reprezentacje właściwościowe słów przejmują czasami nasze najgorsze uprzedzenia. Na przykład mimo że poprawnie „rozumieją”, że *Mężczyzna* ma się do *Króla* tak, jak *Kobieta* ma się do *Królowej*, to „odkrywają” także, że *Mężczyzna* ma się tak do *Lekarza*, jak *Kobieta* do *Pielęgniarki* — jest to dosyć seksistowskie! Aby oddać sprawiedliwość, wypada stwierdzić, że ten akurat przykład jest prawdopodobnie przejakrawiony, o czym wspomina Malvina Nissim i in. w publikacji z 2019 roku (<https://arxiv.org/abs/1905.09866>)¹⁰. Tak czy siak, zapewnienie uczciwości w algorytmach uczenia głębokiego stanowi istotny i aktywnie analizowany obszar badań

Zrozumiemy lepiej mechanizm działania wektorów właściwościowych, jeżeli zaimplementujemy je własnoręcznie (następnie zrobimy to samo za pomocą prostej warstwy Keras). Musimy najpierw utworzyć **macierz właściwościową** (**macierz osadzeń**; ang. *embedding matrix*) zawierającą inicjalizowane losowo wektory właściwościowe poszczególnych kategorii; każda kategoria i każdy

¹⁰ Malvina Nissim i in., *Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor*, arXiv preprint arXiv:1905.09866 (2019).

blok pozawokabularzowy mają wyznaczony własny wiersz, natomiast każda kolumna symbolizuje jeden wymiar wektora właściwościowego:

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets, embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

W przykładzie tym wykorzystujemy dwuwymiarowe wektory właściwościowe, ale zgodnie ze sprawdzonym rozwiążaniem zawierają one od 10 do 300 wymiarów, w zależności od zadania i rozmiaru wokabularza (należy dostroić ten hiperparametr).

Mamy tu do czynienia z losową macierzą właściwościową o wymiarach 6×2 , która jest przechowywana w zmiennej (dzięki czemu może być modyfikowana przez algorytm gradientu prostego w czasie uczenia):

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Zakodujmy tę samą grupę cech kategorialnych co poprzednio, tym razem za pomocą wektorów właściwościowych:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

Funkcja `tf.nn.embedding_lookup()` jedynie przegląda wiersze o wyznaczonych indeksach w macierzy właściwościowej. Na przykład zgodnie z tablicą przeglądową kategoria "INLAND" jest przypisana do indeksu 1, zatem funkcja `tf.nn.embedding_lookup()` zwraca (dwukrotnie) wektor właściwościowy umieszczony w pierwszym wierszu macierzy właściwościowej: [0.3528825, 0.46448255].

Interfejs Keras zawiera warstwę `keras.layers.Embedding`, obsługującą macierz właściwościową (która jest domyślnie modyfikowalna); po utworzeniu warstwy macierz właściwościowa zostaje losowo zainicjalizowana, a po wywołaniu tej warstwy z jakimiś indeksami kategorii zwracane są rzędy macierzy właściwościowej wyznaczane przez te indeksy:

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab) + num_oov_buckets,
... output_dim=embedding_dim)
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
```

```
[-0.01471175, -0.00355174]], dtype=float32)>
```

Połączmy wszystkie te elementy i utwórzmy model Keras, który przetwarza cechy kategorialne (wraz z tradycyjnymi cechami numerycznymi) i wyznacza wektory właściwościowe każdej kategorii (a także każdego bloku pozawokabularzowego):

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

Model ten przyjmuje dwa sygnały wejściowe: standardowy sygnał zawierający osiem cech numerycznych opisujących każdy przykład, a także sygnał kategorialny (po jednej cenie kategorialnej na każdy przykład). Za pomocą warstwy Lambda sprawdzany jest indeks każdej kategorii, a potem wektory właściwościowe przypisane do tych indeksów. Następnie wektory właściwościowe są łączone ze standardowym sygnałem, co pozwala uzyskać sygnał kodowany, przygotowany do przekazania właściwej sieci neuronowej. Moglibyśmy w tym miejscu wstawić dowolną sieć neuronową, ale dodamy jedynie gęstą warstwę wyjściową i utworzymy model Keras.

Gdy zostanie udostępniona warstwa `keras.layers.TextVectorization`, będzie można wywołać jej metodę `adapt()` po to, aby został wydobyty wokabularz z próby danych (tablica przeglądowa zostanie utworzona automatycznie). Możesz następnie dodać tę warstwę do modelu, gdzie zostanie przeprowadzone tablicowanie (możesz nią zastąpić warstwę Lambda użytą w powyższym przykładzie).



Warstwa kodowania gorącojedynkowego połączona z warstwą `Dense` (pozbawioną funkcji aktywacji i obciążień) jest równoważna warstwie `Embedding`. Jednak warstwa `Embedding` wykonuje znacznie mniej obliczeń (różnica wydajności staje się coraz wyraźniejsza w miarę powiększania macierzy właściwościowej). Macierz wag w warstwie `Dense` pełni funkcję macierzy właściwościowej. Na przykład korzystanie z wektorów gorącojedynkowych o rozmiarze 20 i macierzy `Dense` składającej się z 10 jednostek jest równoważne stosowaniu warstwy `Embedding` z parametrami `input_dim=20` i `output_dim=10`. W konsekwencji nie ma sensu wprowadzać więcej wymiarów właściwościowych, niż znajduje się jednostek w następnej warstwie.

Przyjrzyjmy się teraz nieco dokładniej warstwom przetwarzania wstępnego w interfejsie Keras.

Warstwy przetwarzania wstępnego w interfejsie Keras

Zespół twórców modułu TensorFlow dąży do wprowadzenia zestawu standardowych warstw przetwarzania wstępnego w interfejsie Keras (<https://github.com/keras-team/governance/blob/master/rfcs/20190502-preprocessing-layers.md>). Prawdopodobnie zestaw ten będzie już dostępny w momencie zakupu tej książki, jednak do tego czasu API może zostać nieco zmodyfikowany, dlatego jeśli natrafisz na jakiś niespodziewany problem, zajrzyj do notatnika Jupyter poświęconego temu rozdziałowi. Ten nowy interfejs prawdopodobnie zastąpi interfejs kolumn cech (ang. *Feature Columns API*), który jest mniej przystępny i nieintuicyjny (jeżeli mimo to chcesz dowiedzieć się więcej na temat interfejsu kolumn cech, zajrzyj do wspomnianego notatnika).

Znasz już dwie z tych warstw: `keras.layers.Normalization`, która przeprowadza standaryzację cech (będzie ona równoważna zdefiniowanej wcześniej warstwie `Standardization`), i `TextVectorization`, umożliwiającą kodowanie każdego słowa w sygnałach wejściowych na indeksy w wokabularzu. W obydwa przypadkach tworzysz warstwę, wywołujesz jej metodę `adapt()` z próbą danych, a następnie korzystasz normalnie z tej warstwy w modelu. Pozostałe warstwy wstępnego przetwarzania działają zgodnie z identycznym schematem.

Dostępna będzie również warstwa `keras.layers.Discretization`, rozbijająca dane ciągle na zakresy, z których każdy będzie kodowany jako wektor gorącojedynkowy. Na przykład możesz wykorzystywać ją do dyskretyzowania cen na trzy kategorie (niska, średnia, wysoka), które można następnie zakodować jako wektory, odpowiednio, `[1, 0, 0]`, `[0, 1, 0]` i `[0, 0, 1]`. Co prawda tracimy w ten sposób mnóstwo informacji, ale w pewnych sytuacjach rozwiążanie to ułatwia modelowi wykrywanie wzorców, które nie byłyby oczywiste w przypadku danych ciągłych.



Warstwa `Discretization` nie będzie różniczkowalna i powinna być umieszczana na początku modelu. Istotnie, warstwy wstępnego przetwarzania będą zamrożone w trakcie uczenia, dlatego algorytm gradientu prostego nie powinien mieć wpływu na ich parametry, nie muszą być więc różniczkowalne. Oznacza to także, że nie należy stosować warstwy `Embedding` bezpośrednio w niestandardowej warstwie wstępnego przetwarzania, jeżeli chcesz, aby była ona modyfikowalna — w takiej sytuacji należy ją dodać osobno do modelu, tak jak pokazałem w poprzednim przykładzie.

Możliwe będzie także sekwencyjne łączenie wielu warstw wstępnego przetwarzania za pomocą klasy `PreprocessingStage`. Na przykład korzystając z poniższego listingu, utworzymy potok wstępnego przetwarzania, w którym dane będą najpierw normalizowane, a potem dyskretyzowane (może to trochę przypominać potoki Scikit-Learn). Po dostosowaniu tego potoku do próby danych można go użyć jak normalnej warstwy w modelach (pamiętaj jednak, że musi być umieszczony na początku modelu, gdyż zawiera nieróżniczkowalną warstwę wstępnego przetwarzania):

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization, discretization])
pipeline.adapt(data_sample)
```

Warstwa `TextVectorization` także będzie miała możliwość generowania wektorów zliczeń słów zamiast indeksów wyrazów. Na przykład jeżeli wokabularz składa się z trzech słów, powiedzmy `["po", "grzmi", "raz"]`, to tekst `"raz po raz"` będzie odwzorowany na wektor `[1, 0, 2]`: wyraz `"po"` pojawia się raz, `"grzmi"` nie występuje wcale, a `"raz"` zostaje użyty dwukrotnie. Taka reprezentacja jest nazywana „**workiem słów**” (ang. *bag of words*), gdyż nie przechowuje ona informacji o kolejności wyrazów. W większości tekstów takie powszechnie spotykane słowa jak `"po"` będą miały duże wartości, pomimo tego, że są przeważnie najmniej interesujące (np. wyraźnie widać, że w tekście `"raz po raz grzmi"` wyraz `"grzmi"` jest najistotniejszy właśnie dlatego, że nie występuje zbyt często). Zatem zliczenia wyrazów powinny być normalizowane w taki sposób, aby zredukować istotność najpowszechniejszych słów. Często spotykanym rozwiązaniem jest dzielenie każdego zliczenia przez logarytm całkowitej liczby przykładów uczących, w których występuje dane słowo. Jest to technika **ważenie częstości termów × odwrotna częstość w tekście** (ang. *Term-Frequency × Inverse-Document-Frequency* — TF-IDF). Wyobraźmy sobie na przykład, że słowa `"po"`, `"grzmi"`

i "raz" występują w zestawie uczącym, odpowiednio, w 200, 10 i 100 przykładach tekstowych. Wówczas wektor końcowy będzie miał wartości $[1/\log(200), 0/\log(10), 2/\log(100)]$, co w przybliżeniu jest równe $[0.19, 0, 0.43]$. Warstwa TextVectorization będzie (prawdopodobnie) obsługiwała technikę TF-IDF.



Jeżeli standardowe warstwy przetwarzania wstępnego są niewystarczające do realizacji zadania, nadal masz możliwość utworzenia niestandardowej warstwy tego typu, tak jak to zrobiliśmy wcześniej z klasą Standarization. Utwórz podkласę klasy keras.layers.PreprocessingLayer, zawierającą metodę adapt(), która powinna przyjmować argument data_sample, a także, dodatkowo, nadobowiązkowy argument reset_state — jeżeli będzie miał on wartość True, to metoda adapt() powinna wyzerować każdy istniejący stan przed obliczeniem stanu nowego, a w przypadku wartości False metoda ta powinna zaktualizować bieżący stan.

Jak widać, warstwy te znacznie ułatwiają fazę wstępnego przetwarzania! Bez względu na to, czy będziesz tworzyć własne warstwy wstępnego przetwarzania lub korzystać z dostępnych w interfejsie Keras (a nawet jeśli wybierzesz interfejs kolumn cech), to operacje wstępnego przetwarzania będą realizowane na bieżąco. Jednak w trakcie uczenia bardziej opłacalne byłoby wstępne przetwarzanie danych z wyprzedzeniem. Przekonajmy się, dlaczego ma to znaczenie i jak możemy to zrobić.

TF Transform

Jeżeli wstępne przetwarzanie jest kosztowne obliczeniowo, to realizacja tego procesu przed uczeniem zamiast na bieżąco może znacznie przyspieszyć model: dane będą wstępnie przetwarzane tylko raz na każdy przykład **przed** treningiem, a nie raz na przykład i na epokę **w trakcie** uczenia. Jak już wspomniałem, jeżeli zestaw danych mieści się w pamięci operacyjnej, możesz skorzystać z metody cache(). Jeżeli jednak jest zbyt duży, mogą pomóc takie narzędzia jak Apache Beam czy Spark. Pozwalają one korzystać z wydajnych potoków przetwarzania dużych ilości danych, nawet rozproszonych na wielu serwerach, dzięki czemu możemy wstępnie przetwarzać wszystkie dane uczące jeszcze przed rozpoczęciem treningu.

Jest to znakomite rozwiązanie, które rzeczywiście może przyspieszyć proces uczenia, pojawia się jednak pewien problem. Otóż założmy, że po wytrenowaniu modelu chcesz go wdrożyć w aplikacji mobilnej. W takim przypadku musisz umieścić w tej aplikacji kod zajmujący się wstępny przetwarzaniem danych, zanim zostaną przekazane do modelu. Powiedzmy także, że chcesz wdrożyć model za pomocą biblioteki TensorFlow.js, dzięki czemu działałby w przeglądarce. Również w tym przypadku trzeba byłoby przygotować kod przetwarzania wstępne danych. Zarządzanie tyloma elementami mogłoby stać się koszmarem: gdybyśmy chcieli zmodyfikować logikę wstępnego przetwarzania, musielibyśmy zaktualizować kod Apache Beam, aplikacji mobilnej i JavaScriptu. Jest to nie tylko czasochłonne, lecz także podatne na błędy: być może pojawiłyby się subtelne różnice pomiędzy opercjami wstępnego przetwarzania wykonywanymi przed uczeniem a realizowanymi w aplikacji lub przeglądarce. Takie **przeważenie uczenia w stosunku do eksploracji** (ang. *training/serving skew*) prowadzi do błędów i pogorszenia wydajności.

Jednym ze sposobów rozwiązania tego problemu mogłyby być wytrenowanie modelu (wyuczonego na danych wstępnie przetworzonych za pomocą kodu Apache Beam lub Spark) i wprowadzenie do niego, jeszcze przed wdrożeniem go do aplikacji lub przeglądarki, dodatkowych warstw wstępnego przetwarzania, które przetwarzałyby dane na bieżąco. Jest to zdecydowanie lepsze rozwiązanie, ponieważ teraz potrzebne są tylko dwie wersje kodu wstępnego przetwarzania: kod Apache Beam lub Spark i kod warstw wstępnych przetwarzania.

A gdybyśmy mogli zdefiniować operacje wstępnego przetwarzania tylko raz? Do tego właśnie została zaprojektowana biblioteka TF Transform. Stanowi ona część całościowej platformy TensorFlow Extended (TFX; <https://www.tensorflow.org/tfx>), przygotowującej modele TensorFlow do środowiska produkcyjnego. Aby móc wykorzystać składową platformy TFX, taką jak TF Transform, musisz ją zainstalować, nie należy ona bowiem do modułu TensorFlow. Następnie definiujesz funkcję wstępnego przetwarzania tylko raz (w Pythonie), gdzie funkcje biblioteki TF Transform odpowiadają za skalowanie, rozdzielanie na grupy itd. W razie potrzeby możesz również korzystać z dowolnej operacji TensorFlow. Oto przykład funkcji wstępnego przetwarzania dla dwóch cech:

```
import tensorflow_transform as tft

def preprocess(inputs): # inputs = grupa cech wejściowych
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

Biblioteka TF Transform umożliwia zastosowanie tej funkcji preprocess() na całym zestawie uczącym za pomocą narzędzia Apache Beam (zawiera ono klasę AnalyzeAndTransformDataset, którą можemy właśnie w tym celu wykorzystać w potoku Apache Beam). Równocześnie będą także obliczane wszystkie niezbędne statystyki na całym zestawie uczącym: w omawianym przykładzie średnia i odchylenie standardowe cechy housing_median_age, a także wokabularz cechy ocean_proximity. Składniki obliczające te statystyki są nazywane **analitykami** (ang. *analyzers*).

Co ważne, biblioteka TF Transform wygeneruje także równoważną funkcję TensorFlow, którą można dołączyć do wdrażanego modelu. Taka funkcja TF zawiera stałe odpowiadające wszystkim niezbędnym statystykom obliczonym przez narzędzie Apache Beam (średnią, odchylenie standardowe i wokabularz).

Dzięki interfejsowi danych, formatowi TFRecords, warstwom przetwarzania wstępnego Keras i bibliotece TF Transform możesz tworzyć wysoce skalowalne potoki danych wejściowych pozwalające na uczenie modeli oraz cieszyć się zaletami szybkiego i przenośnego wstępnego przetwarzania danych w środowisku produkcyjnym.

A gdybyśmy chcieli korzystać wyłącznie ze standardowego zestawu danych? Wówczas wszystko jest znacznie prostsze: użąd projektu TFDS!

Projekt TensorFlow Datasets (TFDS)

Projekt TensorFlow Datasets (<https://www.tensorflow.org/datasets>) pozwala bardzo szybko pobierać popularne zestawy danych, od tak małych jak MNIST czy Fashion MNIST aż do tak rozbu-dowanych jak ImageNet (lepiej przygotuj miejsce na dysku!). Dostępne są zestawy obrazów, tekstów (w tym przekładów), dźwięków i filmów. Pełną listę (wraz z opisami) zestawów danych znajdziesz na stronie <https://www.tensorflow.org/datasets/catalog/overview>.

Projekt TFDS nie jest dołączony do modułu TensorFlow, dlatego należy zainstalować bibliotekę tensorflow-datasets (np. za pomocą polecenia pip). Następnie wywołaj funkcję `tfds.load()`, co spowoduje pobranie interesujących Cię zestawów danych (jeśli nie zostały już pobrane wcześniej) i zwrócenie danych jako słownika zestawów danych (zazwyczaj po jednym słowniku na zbiór uczący i testowy. Pobierzmy na przykład zestaw danych MNIST:

```
import tensorflow_datasets as tfds

dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

Możesz teraz wprowadzić dowolne przekształcenie (najczęściej tasowanie, dzielenie na grupy i pobie-ranie wstępne) i model będzie już gotowy do uczenia. Oto prosty przykład:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```



Funkcja `load()` tasuje każdy otrzymany pakiet danych (jedynie w zbiorze uczącym). Może to nie wystarczyć, dlatego najlepiej potasować dane uczące trochę bardziej.

Zwróć uwagę, że każdy element w zestawie danych jest słownikiem zawierającym zarówno cechy, jak i etykiety. Interfejs Keras oczekuje jednak dwuelementowej krotki (cech i etykiet). Możesz przekształcić zestaw danych za pomocą metody `map()`, na przykład:

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"], items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Może nas w tym wyrzucić funkcja `load()`, jeżeli wyznaczymy parametr `as_supervised=True` (oczywiście działa to wyłącznie w przypadku oznakowanych zestawów danych). Możesz także zdefiniować rozmiar grupy. Teraz możesz przekazać zestaw danych bezpośrednio do modelu `tf.keras`:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

Informacje zawarte w tym rozdziale były dość techniczne i może się wydawać, że odstaje on nieco od abstrakcyjnego piękna sieci neuronowych, w rzeczywistości jednak z uczeniem głębokim nieraz

związane są olbrzymie ilości danych, a znajomość sposobów ich wydajnego wczytywania, analizowania i wstępnego przetwarzania jest kluczowa. W następnym rozdziale zajmiemy się splotowymi sieciami neuronowymi, które są uznawane za jedne z najskuteczniejszych struktur przetwarzających obrazy i nie tylko.

Ćwiczenia

1. Dlaczego warto korzystać z interfejsu danych (Data API)?
2. Jakie korzyści płyną z rozbijania dużych zestawów danych na wiele plików?
3. W jaki sposób można stwierdzić w fazie uczenia, że zator znajduje się w potoku danych wejściowych? Jak możemy go usunąć?
4. Czy w pliku TFRecord można zapisywać dowolne dane binarne, czy tylko serializowane bufore protokołów?
5. Dlaczego warto przekształcić wszystkie dane do formatu bufora protokołów Example? Dlaczego nie lepiej jest wykorzystać własną definicję bufora protokołów?
6. Kiedy można wykorzystać kompresję podczas stosowania formatu TFRecord? Dlaczego nie robimy tego systematycznie?
7. Dane mogą być wstępnie przetwarzane bezpośrednio podczas zapisywania plików danych, wewnętrz potoku `tf.data`, w warstwach wstępniego przetwarzania wewnętrz modelu lub za pomocą biblioteki TF Transform. Wymień wady i zalety każdego z tych rozwiązań.
8. Wymień kilka popularnych technik kodowania cech kategoryalnych. Jak możemy kodować tekst?
9. Wczytaj zestaw danych MNIST (omówiony w rozdziale 10.), podziel go na zbiory uczący, walidacyjny i testowy, przetasuj zbiór uczący, zapisz każdy zbiór danych w wielu plikach TFRecord. Każdy rekord powinien być serializowanym buforem protokołów Example zawierającym dwie cechy: serializowany obraz (do serializowania każdego obrazu wykorzystaj funkcję `tf.io.serialize_tensor()`) i etykietę¹¹. Następnie wykorzystaj moduł `tf.data` do utworzenia wydajnego zestawu danych dla każdego zbioru. Na koniec wytrenuj te zestawy danych za pomocą modelu Keras (ustandardyzuj każdą cechę wejściową za pomocą warstwy przetwarzania wstępnego). Spróbuj zmaksymalizować skuteczność potoku danych wejściowych (dane profilu możesz zwizualizować za pomocą narzędzia TensorBoard).
10. W tym ćwiczeniu pobierzesz zestaw danych, podzielisz go, utworzysz `tf.data.Dataset` w celu wydajnego wczytywania i wstępnego przetwarzania tych zbiorów, a następnie zbudujesz i wytrenujesz model klasyfikatora binarnego zawierający warstwę Embedding:
 - a. Pobierz zestaw danych Large Movie Review (<http://ai.stanford.edu/~amaas/data/sentiment/>), składający się z 50 000 recenzji filmów dostępnych w serwisie Internet Movie Database (<https://www.imdb.com/>). Dane są zorganizowane w dwóch katalogach, `train` i `test`, z których każdy zawiera podkatalog `pos`, przechowujący 12 500 recenzji pozytywnych, oraz podkatalog `neg` z taką samą liczbą recenzji negatywnych. Każda recenzja umieszczona jest w odrębnym

¹¹ W przypadku dużych obrazów możesz wykorzystać funkcję `tf.io.encode_jpeg()`, dzięki czemu zaoszczędzisz mnóstwo miejsca kosztem nieznacznego pogorszenia jakości obrazów.

pliku tekstowym. Znajdziemy tu także inne pliki i foldery (w tym wstępnie przetworzone „worki słów”), ale na razie nie będą nas one interesowały.

- b.** Podziel zbiór testowy na podzbiory walidacyjny (15 000 przykładów) i testowy (10 000 przykładów).
- c.** Za pomocą biblioteki `tf.data` utwórz wydajny zestaw danych dla każdego podzbioru.
- d.** Utwórz model klasyfikacji binarnej, w którym warstwa `TextVectorization` będzie wstępnie przetwarzala każdą recenzję. Jeżeli warstwa `TextVectorization` nie będzie jeszcze dostępna (lub jeśli lubisz wyzwania), spróbuj utworzyć własną niestandardową warstwę wstępnego przetwarzania — możesz wykorzystać funkcje z pakietu `tf.strings`, np. `lower()`, aby ujednolicić cały tekst do małych liter, `regex_replace()`, aby zastąpić wszystkie znaki interpunkcyjne odstępami, czy `split()`, aby rozdzielić wyrazy w miejscu spacji. Wykorzystaj tablicę przeglądową do wyznaczenia indeksów wyrazów, które muszą być przygotowane w metodzie `adapt()`.
- e.** Dodaj warstwę `Embedding` i oblicz średnią reprezentację właściwościową dla każdej recenzji pomnożoną przez pierwiastek kwadratowy z liczby słów (zob. rozdział 16.). Taka przeskalowana, średnia reprezentacja właściwościowa może zostać następnie przekazana do dalszej części modelu.
- f.** Wytrenuj model i sprawdź, jaką uzyskuje dokładność. Spróbuj tak zoptymalizować potoki, aby maksymalnie przyspieszyć proces uczenia.
- g.** Ułatw sobie życie poprzez pobranie tego samego zestawu danych za pomocą projektu TFDS: `tfds.load("imdb_reviews")`.

Rozwiązania ćwiczeń znajdziesz w dodatku A.

Głębokie widzenie komputerowe za pomocą splotowych sieci neuronowych

Chociaż superkomputer Deep Blue zaprojektowany przez firmę IBM już w 1996 roku pokonał szachowego mistrza świata Garriego Kasparowa, aż do niedawna komputery nie były w stanie niezawodnie wykonywać pozornie trywialnych zadań, takich jak wykrywanie szczeniaczków na zdjęciach lub rozpoznawanie wypowiadanych słów. Dlaczego nie sprawiają one ludziom problemu? Odpowiedź kryje się w fakcie, że proces postrzegania odbywa się w dużej mierze poza naszą świadomością, w wyspecjalizowanych modułach wzrokowych, słuchowych i innych mieszczących się w naszym mózgu. Zanim informacja sensoryczna dotrze do świadomości, zostaje „ubrana” w zaawansowane cechy; na przykład spoglądając na zdjęcie uroczego szczeniaczka, *nie* możesz wybrać, aby nie widzieć tego szczeniaczka ani *nie* zauważyc, że jest uroczy. Tak samo nie potrafisz wyjaśnić, *w jaki sposób* rozpoznałaś/rozpoznałeś w ogóle szczeniaka na zdjęciu; przecież jest dla Ciebie oczywiste, co widzisz na zdjęciu. Dlatego nie możemy ufać naszym subiektywnym wrażeniom — w procesie postrzegania nie ma nic trywialnego, a żeby to sobie uświadomić, musimy przyjrzeć się mechanizmom działania modułów sensorycznych.

Splotowe (konwolucyjne) sieci neuronowe (ang. *convolutional neural networks* — CNN) stanowią wynik badań nad korą wzrokową i od lat 80. ubiegłego wieku są używane do rozpoznawania obrazów. W ciągu kilku ostatnich lat sieci CNN zdobyły osiągnąć wyniki przekraczające ludzkie możliwości w pewnych skomplikowanych zadaniach wizualnych; stało się to możliwe dzięki wzrostowi mocy obliczeniowej, ilości dostępnych danych uczących i opisanym w rozdziale 11. sposobom uczenia sztucznych sieci neuronowych. Stanowią one podstawę usług wyszukiwania obrazów, samochodów inteligentnych, zautomatyzowanych systemów klasyfikowania filmów itd. Co więcej, splotowe sieci neuronowe nie ograniczają się wyłącznie do postrzegania obrazów: okazują się również skuteczne w innych zadaniach, takich jak **rozpoznawanie mowy** (ang. *voice recognition*) czy **przetwarzanie języka naturalnego** (ang. *natural language processing* — NLP); na razie jednak skoncentrujemy się na zastosowaniach wzrokowych.

W tym rozdziale wyjaśnię genezę sieci CNN, ich budowę oraz sposób implementacji za pomocą modułu TensorFlow i interfejsu Keras. Następnie przedstawię jedne z najlepszych architektur splotowych sieci neuronowych, a także przyjrzymy się innym zadaniom wizualnym, takim jak wykrywanie obiektów (klasyfikowanie wielu obiektów widocznych na obrazie i umieszczanie ramek

ograniczających wokół nich) czy segmentacja semantyczna (klasyfikowanie każdego piksela zgodnie z klasą obiektu, do którego przynależy ten piksel).

Struktura kory wzrokowej

David H. Hubel i Torsten Wiesel przeprowadzili szereg eksperymentów na kotach w latach 1958 (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1357023/pdf/jphysiol01301-0020.pdf>)¹ i 1959 (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1363130/pdf/jphysiol01298-0128.pdf>)² (a kilka lat później także na małpach, <https://physsoc.onlinelibrary.wiley.com/doi/epdf/10.1111/jphysiol.1968.sp008455>³), dzięki którym poznaliśmy strukturę kory wzrokowej (autorzy za swój wkład w rozwój nauki otrzymali w 1981 roku Nagrodę Nobla w dziedzinie fizjologii lub medycyny). Udowodnili oni w szczególności, że wiele neuronów składających się na korę wzrokową wyznacza **lokalne pola recepcyjne**, tzn. że reagują jedynie na bodźce wzrokowe miesiące się w określonym rejonie pola wzrokowego (zobacz rysunek 14.1, na którym lokalne pola recepcyjne pięciu neuronów zostały ukazane w postaci przerywanych okręgów). Pola recepcyjne poszczególnych neuronów mogą się na siebie nakładać i łącznie tworzą całe pole wzrokowe.

Do tego badacze pokazali także, że pewne neurony reagują wyłącznie na obrazy składające się z linii poziomych, a inne są pobudzane przez linie ułożone w inny sposób (dwa neurony mogą mieć to samo pole recepcyjne, ale reagować na inne ułożenie linii). Zauważono także, że niektóre komórki nerwowe mają większe pola recepcyjne i wykrywają bardziej skomplikowane kształty, stanowiące połączenie bardziej ogólnych wzorów. Poczynione obserwacje doprowadziły badaczy do wniosku, że neurony odpowiedzialne za rozpoznawanie bardziej skomplikowanych kształtów znajdują się na wyjściu sąsiadujących neuronów reagujących na prostsze bodźce wzrokowe (zwróć uwagę, że na rysunku 14.1 każdy neuron jest połączony tylko z kilkoma neuronami z niższej warstwy). Taka architektura pozwala wykrywać wszelkie rodzaje skomplikowanych kształtów w obszarze pola wzrokowego.

Te badania kory wzrokowej stanowiły inspirację dla modelu neokognitronu (<http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf>)⁴ (zaprezentowanego w 1980 roku), który stopniowo przeistoczył się w architekturę obecnie znaną jako **splotowe sieci neuronowe**. Wydatny w tym udział miała publikacja z 1998 roku (<http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>)⁵, w której Yann LeCun i in. zaprezentowali słynną sieć **LeNet-5**, która była powszechnie używana do rozpoznawania odręcznie pisanych cyfr na czekach. Architektura ta zawiera część dobrze

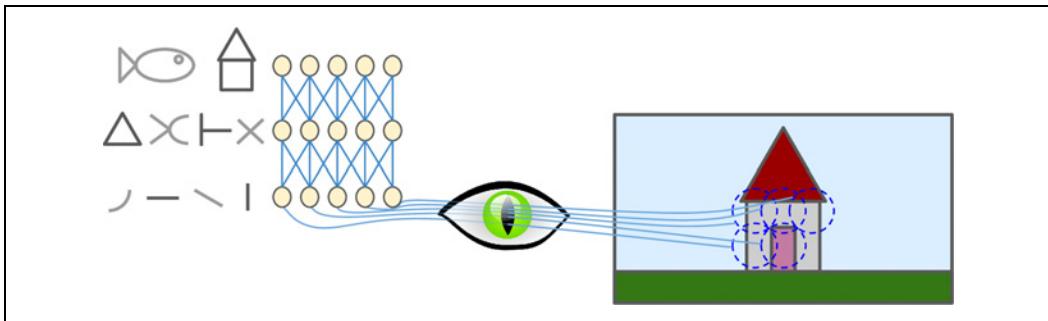
¹ David H. Hubel, *Single Unit Activity in Striate Cortex of Unrestrained Cats*, „The Journal of Physiology” 147 (1959), s. 226 – 238.

² David H. Hubel i Torsten N. Wiesel, *Receptive Fields of Single Neurons in the Cat’s Striate Cortex*, „The Journal of Physiology” 148 (1959), s. 574 – 591.

³ David H. Hubel i Torsten N. Wiesel, *Receptive Fields and Functional Architecture of Monkey Striate Cortex*, „The Journal of Physiology” 195 (1968), s. 215 – 243.

⁴ Kunihiko Fukushima, *Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position*, „Biological Cybernetics” 36 (1980), s. 193 – 202.

⁵ Yann LeCun i in., *Gradient-Based Learning Applied to Document Recognition*, „Proceedings of the IEEE” 86, no. 11 (1998), s. 2278 – 2324.



Rysunek 14.1. Neurony biologiczne w korze wzrokowej reagują na określone wzorce w niewielkich obszarach pola wzrokowego, zwanych polami receptivejnymi; w miarę przepływu sygnału wzrokowego przez kolejne moduły w mózgu neurony rozpoznają coraz bardziej skomplikowane wzorce wykrywane w coraz większych polach receptivejnych

nam już znanych elementów, np. w pełni połączone warstwy i sigmoidalną funkcję aktywacji, ale znajdują się w niej również nowe składniki, mianowicie **warstwy splotowe** i **warstwy łączące**. Przyjrzyjmy się im teraz uważniej.

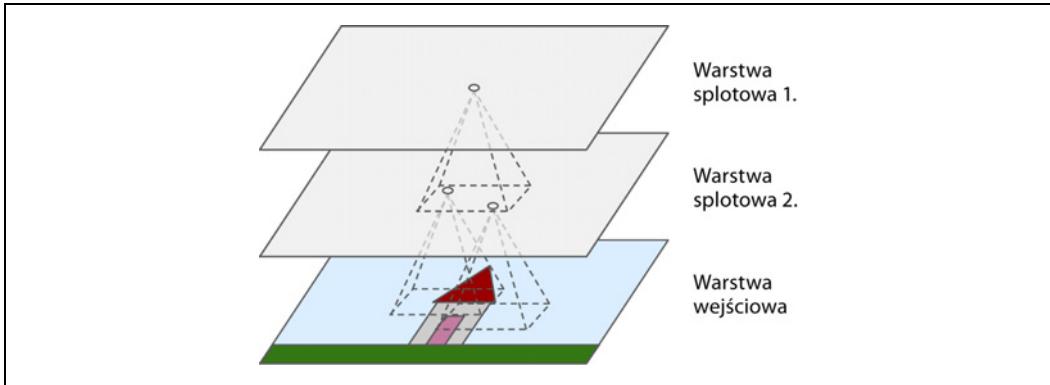


Dlaczego nie możemy stosować standardowej głębokiej sieci neuronowej zawierającej w pełni połączone warstwy do rozpoznawania obrazów? Niestety mimo że dobrze sobie one radzą z niewielkimi obrazami (np. zbiorem danych MNIST), nie nadają się do większych obrazów z powodu olbrzymiej liczby parametrów. Na przykład obraz o rozmiarze 100×100 zawiera 10 000 pikseli, a jeżeli pierwsza warstwa sieci składa się zaledwie z tysiąca neuronów (co już samo w sobie znacznie ogranicza ilość informacji dostarczanych do następnej warstwy), oznacza to łącznie 10 milionów połączeń, a to dotyczy zaledwie pierwszej warstwy. Problem ten zostaje rozwiązany w sieciach CNN poprzez wprowadzenie częściowo połączonych warstw i współdzielenia wag.

Warstwy splotowe

Najistotniejszym składnikiem sieci CNN jest **warstwa splotowa** (*konwolucyjna*; ang. *convolutional layer*)⁶: neurony w pierwszej warstwie splotowej nie są połączone z każdym pikselem obrazu wejściowego (w przeciwieństwie do sieci opisanych w poprzednich rozdziałach), lecz wyłącznie z pikselami znajdującymi się w ich polu receptivejnym (rysunek 14.2). Z kolei każdy neuron w drugiej warstwie splotowej łączy się wyłącznie z neuronami znajdującymi się w niewielkim obszarze pierwszej warstwy. Dzięki temu sieć może koncentrować się na ogólnych cechach w pierwszej warstwie ukrytej, następnie łączyć je w bardziej złożone kształty w drugiej warstwie ukrytej itd. Taka hierarchiczna struktura jest powszechnie spotykana na zdjęciach, co stanowi jedną z przyczyn tak dużej skuteczności sieci CNN w rozpoznawaniu obrazów.

⁶ Splot (konwolucja) jest operacją matematyczną pomiędzy dwiema funkcjami, mierzącą całkę z ich iloczynu punktowego. Jest on ściśle powiązany z transformacjami Fouriera i Laplace'a oraz powszechnie używany w przetwarzaniu sygnałów. W rzeczywistości w sieciach CNN stosowane są operacje korelacji krzyżowej, bardzo przypominające konwolucję (więcej szczegółów znajdziesz pod adresem <https://en.wikipedia.org/wiki/Convolution> oraz, w języku polskim, na stronie [https://pl.wikipedia.org/wiki/Splot_\(analiza_matematyczna\)](https://pl.wikipedia.org/wiki/Splot_(analiza_matematyczna))).

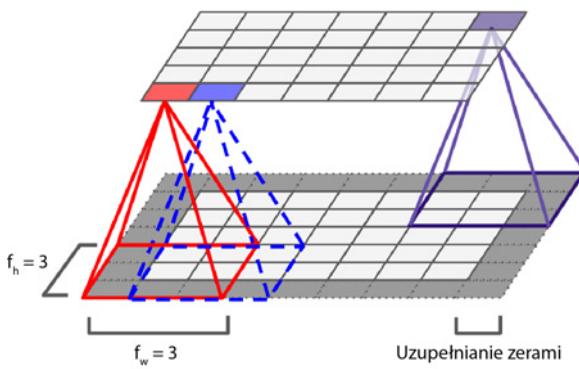


Rysunek 14.2. Warstwy CNN z prostokątnymi lokalnymi polami recepcyjnymi



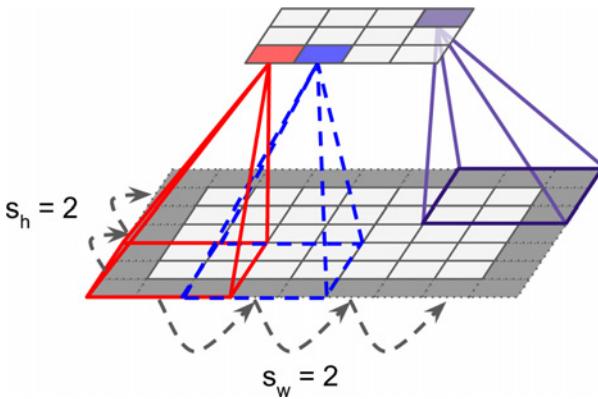
Do tej pory wszystkie analizowane przez nas wielowarstwowe sieci neuronowe miały warstwy składające się z długich rzędów neuronów, a przed przekazaniem obrazu do sieci musielismy go przekształcić do postaci jednowymiarowej. Od teraz każda warstwa będzie dwuwymiarowa, co ułatwi nam obserwowanie związków pomiędzy neuronami a ich danymi wejściowymi.

Neuron znajdujący się w wierszu i oraz kolumnie j danej warstwy jest połączony z wyjściami neuronów poprzedniej warstwy zlokalizowanymi w rzędach od i do $i+f_h-1$ i kolumnach od j do $j+f_w-1$, gdzie f_h i f_w oznaczają, odpowiednio, wysokość i szerokość pola recepcyjnego (rysunek 14.3). W celu uzyskania takich samych wymiarów każdej warstwy najczęściej są dodawane zera wokół wejść, co zostało pokazane na rysunku 14.3. Proces ten nazywamy **uzupełnianiem zerami** (ang. *zero padding*).



Rysunek 14.3. Związek pomiędzy warstwami a uzupełnianiem zerami

Możliwe jest również łączenie bardzo dużej warstwy wejściowej ze znacznie mniejszą kolejną warstwą poprzez rozdzielenie pól recepcyjnych, tak jak zaprezentowano na rysunku 14.4. Rozwiążanie to zmniejsza drastycznie złożoność obliczeniową modelu. Odległość pomiędzy dwoma kolejnymi polami recepcyjnymi nosi nazwę **kroku** (ang. *stride*). Na widocznym schemacie warstwa wejściowa o wymiarach 5×7 (plus uzupełnianie zerami) łączy się z warstwą o rozmiarze 3×4 za pomocą pól recepcyjnych będących kwadratami 3×3 i kroku o wartości 2 (w omawianym przykładzie krok jest taki sam



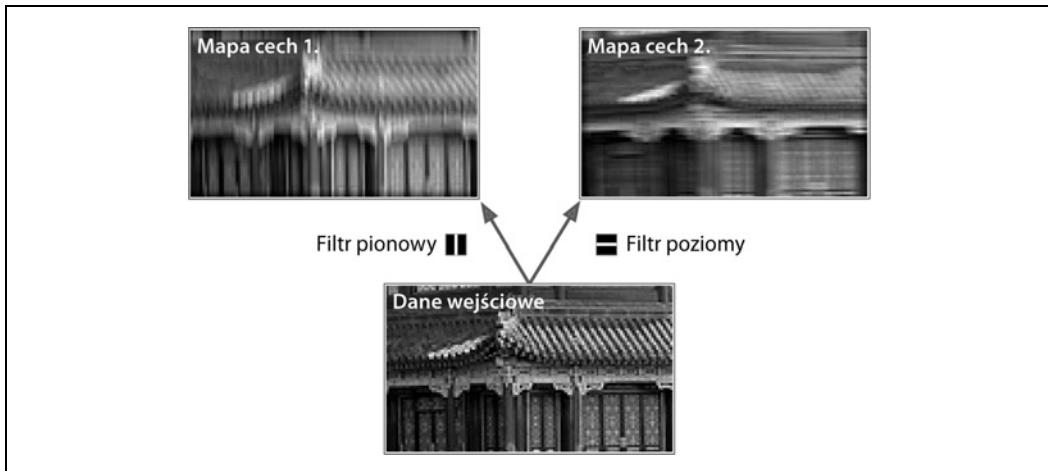
Rysunek 14.4. Redukcja wymiarowości za pomocą kroku o wartości 2

w obydwu wymiarach, ale nie jest to wcale regułą). Neuron zlokalizowany w rzędzie i oraz kolumnie j górnej warstwy łączy się z wyjściami neuronów dolnej warstwy mieszczącymi się w rzędach od $i \times s_h$ do $i \times s_h + f_h - 1$ i w kolumnach od $j \times s_w$ do $j \times s_w + f_w - 1$, gdzie s_h i s_w definiują wartości kroków odpowiednio w kolumnach i rzędach.

Filtры

Wagi neuronu mogą być przedstawiane jako niewielki obraz o rozmiarze pola recepcyjnego. Na przykład na rysunku 14.5 widzimy dwa możliwe zbiory wag, tak zwane **filtre** (lub **jądra splotowe**; ang. *convolution kernels*). Pierwszy filtr jest symbolizowany jako czarny kwadrat z białą pionową linią przechodzącą przez jego środek (jest to macierz o wymiarach 7×7 wypełniona zerami oprócz środkowej kolumny, która zawiera jedynki); neurony zawierające te wagi będą ignorować wszystkie elementy w polu recepcyjnym oprócz znajdujących się w środkowej pionowej linii (dane wejściowe znajdujące się poza tą linią będą przemnażane przez 0). Drugi filtr wygląda podobnie; różnica polega na tym, że środkowa linia jest ułożona poziomo. Także w tym wypadku będą brane pod uwagę jedynie dane wejściowe znajdujące się w tej linii.

Jeśli wszystkie neurony w danej warstwie będą korzystać z tego samego filtra „pionowego” (i takiego samego członu obciążenia), a my wczytamy do sieci obraz zaprezentowany na dole rysunku 14.5, to uzyskamy obraz widoczny w lewym górnym rogu rysunku. Zauważ, że po zastosowaniu tego filtru pionowe białe linie stają się wyraźniej widoczne, natomiast pozostała część obrazu zostaje rozmazana. Na zasadzie analogii otrzymujemy obraz widoczny w prawym górnym rogu rysunku po zastosowaniu filtru „poziomego”; teraz z kolei białe poziome linie zostają wyostrzone, a reszta obrazu ulega zamazaniu. Zatem warstwa wypełniona neuronami wykorzystującymi ten sam filtr daje nam **mapę cech** (ang. *feature map*), dzięki której możemy dostrzec elementy najbardziej przypominające dany filtr. Nie musisz oczywiście definiować filtrów własnoręcznie — sieć CNN w czasie uczenia wyszukuje filtry najbardziej przydatne do danego zadania i uczy się łączyć je w bardziej złożone wzorce.



Rysunek 14.5. Uzyskiwanie dwóch map cech za pomocą dwóch różnych filtrów

Stosy map cech

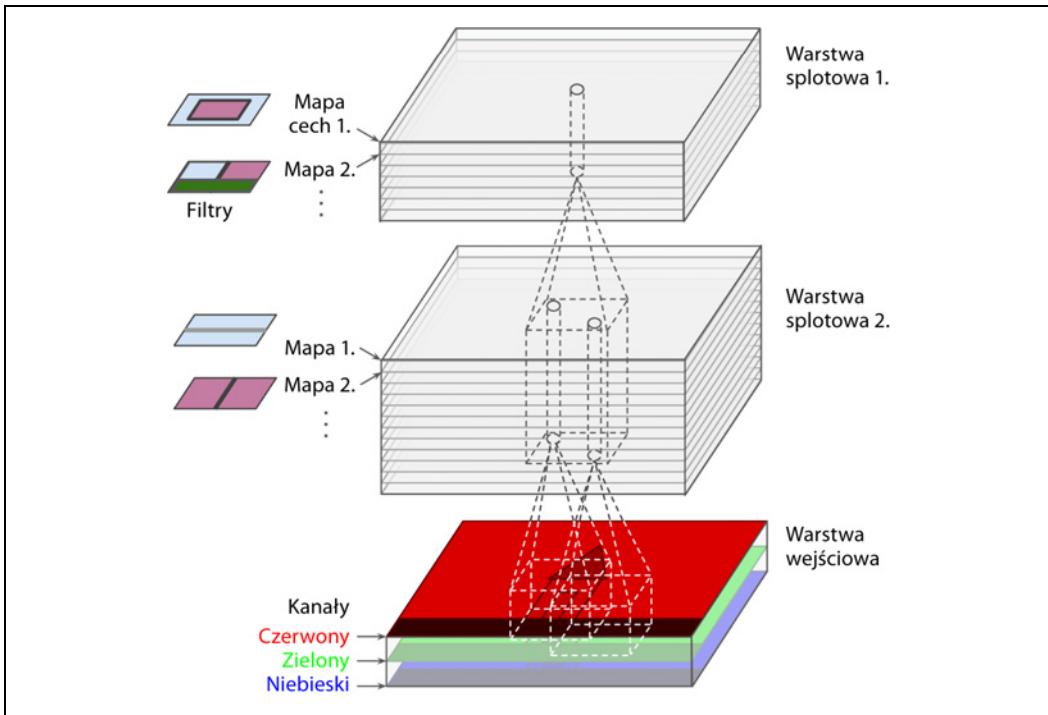
Do tej pory dla uproszczenia przedstawialiśmy każdą warstwę splotową w postaci cienkiej, dwuwymiarowej warstwy, ale w rzeczywistości składa się ona z kilku map cech o identycznych rozmiarach, dlatego trójwymiarowe odwzorowanie jest bliższe rzeczywistości (rysunek 14.6). W zakresie jednej mapy cech każdy neuron jest przydzielony do jednego piksela, a wszystkie tworzące ją neurony współdzielą te same parametry (wagi i człon obciążenia). Neurony w innych mapach cech mają odmienne wartości parametrów. Pole recepcyjne neuronu nie ulega zmianie, ale „przebiega” przez wszystkie mapy cech poprzednich warstw. Krótko mówiąc, warstwa splotowa równocześnie stosuje różne filtry na wejściach, dzięki czemu jest w stanie wykrywać wiele cech w dowolnym obszarze obrazu.



Dzięki temu, że wszystkie neurony w mapie cech stosują te same parametry, znacznie zmniejsza się ich liczba w modelu, jednak co ważniejsze, oznacza to, że gdy sieć CNN nauczy się rozpoznawać wzorzec w jednym miejscu, będzie w stanie to robić również w innych lokacjach. Dla porównania, sieć GSN po nauczeniu się rozpoznawania wzorca w jednym miejscu nie potrafi tego przełożyć na inne obszary.

Co więcej, obrazy wejściowe także składają się z kilku warstw podrzędnych, po jednej na każdy **kanał barw** (ang. *color channel*). Standardowo występują trzy kanały barw — czerwony, zielony i niebieski (ang. *red, green, blue* — RGB). Obrazy czarno-białe (w odcieniach szarości) zawierają tylko jeden kanał, ale istnieją też takie zdjęcia, które mogą mieć ich znacznie więcej — np. fotografie satelitarne utrwalające dodatkowe częstotliwości fal elektromagnetycznych (takie jak podczerwień).

W szczególności neuron zlokalizowany w rzędzie i oraz kolumnie j mapy cech k w danej warstwie splotowej l jest połączony z neuronami wcześniejszej warstwy $l-1$ umieszczonymi w rzędach od $i \times s_h$ do $i \times s_h + f_h - 1$ i kolumnach od $j \times s_w$ do $j \times s_w + f_w - 1$ we wszystkich mapach cech (warstwy $l-1$). Zwróć uwagę, że wszystkie neurony znajdujące się w tym samym rzędzie i oraz kolumnie j , ale w innych mapach cech są połączone z wyjściami dokładnie tych samych neuronów poprzedniej warstwy.



Rysunek 14.6. Warstwy splotowe zawierające wiele map cech, a także zdjęcie z trzema kanałami barw

Powyższy opis został podsumowany w równaniu 14.1: widoczny duży wzór matematyczny służy do obliczania wyniku danego neuronu w warstwie splotowej. Z powodu dużej liczby indeksów równanie to nie wygląda zbyt elegancko, ale za jego pomocą jesteśmy w stanie uzyskać sumę ważoną wszystkich danych wejściowych wraz z członem obciążenia.

Równanie 14.1. Obliczanie wartości wyjściowej neuronu w warstwie splotowej

$$z_{i,j,k} = b_k \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{h'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{gdzie} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

W tym równaniu:

- $z_{i,j,k}$ jest wyjściem neuronu znajdującego się w rzędzie i , kolumnie j i mapie cech k warstwy splotowej l ;
- jak już zostało wyjaśnione, s_h i s_w to kroki pionowy i poziomy, f_h i f_w są wysokością i szerokością pola receptivejnego, natomiast $f_{h'}$ oznacza liczbę map cech w poprzedniej warstwie ($l-1$);
- $x_{i,j,k}$ jest wyjściem neuronu zlokalizowanego w warstwie $l-1$, rzędzie i' , kolumnie j' , mapie cech k (lub kanale k' , jeżeli poprzednia warstwa była warstwą wejściową);
- b_k to człon obciążenia dla mapy cech k (w warstwie l); można go interpretować jako „pokrętło jasności” mapy cech k ;

- $w_{u,v,k;k'}$ jest wagą połączenia pomiędzy dowolnym neuronem w mapie cech k warstwy l a jego wejściem mieszczącym się w wierszu u , kolumnie v (względem pola recepcyjnego neuronu) a mapą cech k' .

Implementacja w module TensorFlow

W module TensorFlow każdy obraz wejściowy jest zazwyczaj przedstawiany jako trójwymiarowy tensor o wymiarach [wysokość, szerokość, kanały]. Z kolei minigrupa ma formę czterowymiarowego tensora — [rozmiar minigrupy, wysokość, szerokość, kanały]. Wagi warstwy splotowej również przyjmują kształt czterowymiarowego tensora — [f_{lb}, f_{lw}, f_n, f_n]. Dla odmiany człon obciążenia warstwy splotowej jest po prostu jednowymiarowym tensorem o postaci $[f_n]$.

Przeanalizujmy prosty przykład. Za pomocą poniższego kodu wczytujemy dwa przykładowe obrazy, używając w tym celu funkcji `load_sample_image()` (wstawiamy w ten sposób dwa zdjęcia — chińskiej świątyni i kwiatu). Następnie tworzymy dwa filtry po czym zostaje wygenerowany obraz jednej z wynikowych map cech. Zauważ, że należy zainstalować pakiet Pillow za pomocą polecenia pip, aby móc skorzystać z funkcji `load_sample_image()`.

```
from sklearn.datasets import load_sample_image

# Wczytuje przykładowe obrazy
china = load_sample_image("Chiny.jpg") / 255
flower = load_sample_image("kwiat.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Tworzy dwa filtry
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, :, 0] = 1 # Linia pionowa
filters[3, :, :, 1] = 1 # Linia pozioma

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

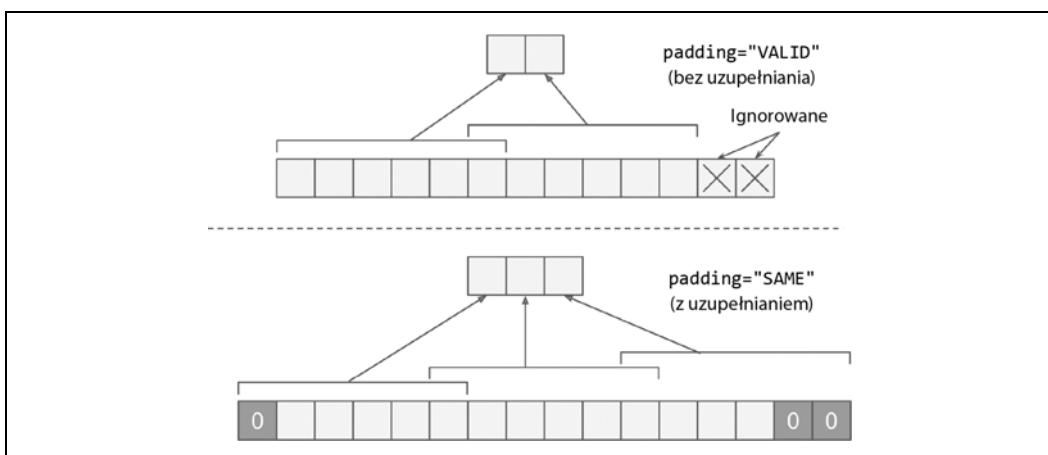
plt.imshow(outputs[0, :, :, 1], cmap="gray") # Wyświetla drugą mapę cech z pierwszego rysunku
plt.show()
```

Przeanalizujmy powyższy listing:

- Intensywność piksela dla każdego kanału barw reprezentowana jest w postaci bajtowej, w zakresie od 0 do 255, zatem aby otrzymać wartość zmiennoprzecinkową w zakresie od 0 do 1, wystarczy przeskalać te cechy poprzez podzielenie ich przez 255.
- Tworzymy następnie dwa filtry 7×7 (jeden z nich będzie zawierał pośrodku linię pionową, a drugi linię poziomą).
- Stosujemy te filtry na obydwu obrazach za pomocą funkcji `tf.nn.conv2d()`, która stanowi część ogólnego interfejsu uczenia głębokiego TensorFlow. W tym przykładzie wykorzystujemy uzupełnianie zerami (`padding="SAME"`) i krok równy 1.
- Na koniec rysujemy wykres jednej z wygenerowanych map cech (podobny do pokazanego na rysunku 14.5).

Wiersz `tf.nn.conv2d()` wymaga dodatkowego wyjaśnienia:

- `images` jest minigrupą danych wejściowych (wiemy już, że ma ona postać czterowymiarowego tensora).
- Obiekt `filters` stanowi zbiór używanych przez nas filtrów (również wiemy, że jest on czterowymiarowym tensorem).
- `strides` przyjmuje wartość 1, ale może to być również czteroelementowy, jednowymiarowy tensor, w którym dwa środkowe elementy określają rozmiar kroku w kierunku pionowym i poziomym (s_h i s_w). Obecnie elementy krańcowe muszą przyjmować wartość 1. Kiedyś mogły przydać się do określania kroków wobec minigrupy (pozwalałoby to pomijać pewne przykłady) i wobec kanału (w celu pomijania niektórych kanałów/map cech z wcześniejszych warstw).
- Parametr `padding` może przyjmować wyłącznie wartości `VALID` lub `SAME`:
 - Po wstawieniu wartości `VALID` warstwa splotowa *nie* używa uzupełniania zerami oraz może ignorować niektóre dolne rzędy i kolumny po prawej stronie obrazu, w zależności od rozmiaru kroku (rysunek 14.7; dla uproszczenia pokazany został jedynie wymiar horyzontalny, ale tak samo sytuacja wygląda w przypadku wymiaru wertykalnego). Oznacza to, że pole recepcyjne każdego neuronu mieści się wyłącznie na prawidłowych pozycjach wewnętrz obrazu wejściowego (nie wykracza poza ten obraz), stąd nazwa **valid**.



Rysunek 14.7. Opcje uzupełniania zerami — szerokość danych wejściowych: 13; szerokość filtra: 6; skok: 5

- Po wstawieniu wartości `SAME` uzupełnianie zerami będzie stosowane w razie potrzeby. W takim przypadku liczba neuronów wyjściowych jest równa zaokrąglonemu w góre ilorazowi neuronów wejściowych i rozmiaru kroku. Na przykład jeżeli rozmiar wejściowy wynosi 13, a krok 5 (rysunek 14.7), to rozmiar wyjściowy będzie równy 3 (tzn. $13/5 \approx 3$). Następnie są dodawane zera wokół danych wejściowych w taki sposób, żeby wyszło jak najrówniej. Gdy `strides=1`, to sygnał wyjściowy warstwy będzie miał takie same wymiary przestrzenne (szerokość i wysokość) jak sygnał wejściowy, stąd nazwa **same**.

W tym prostym przykładzie stworzyliśmy własnoręcznie filtry, ale w rzeczywistej sieci CNN możesz definiować filtry jako zmienne modyfikowalne, dzięki czemu sieć samodzielnie wyszuka ich najlepsze konfiguracje. Nie musimy tworzyć własnoręcznie zmiennych — wyręczy nas w tym warstwa keras.layers.Conv2d:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

Listing ten tworzy warstwę Conv2D zawierającą 32 filtry o rozmiarze 3×3 , kroku 1 (zarówno w poziomie, jak i w pionie) i z uzupełnianiem zerami typu "same", natomiast sygnały wyjściowe zostają potraktowane funkcją aktywacji ReLU. Wyraźnie widać, że warstwy splotowe zawierają kilka hiperparametrów — należy dobrąć liczbę filtrów, ich wysokość i szerokość, krok oraz typ uzupełniania zerami. Jak zawsze możemy wyszukać ich odpowiednie wartości za pomocą sprawdzianu krzyżowego, ale proces ten jest bardzo czasochłonny. W dalszej części rozdziału przyjrzymy się popularnym architekturom sieci CNN, aby mieć pogląd na to, jakie wartości hiperparametrów najlepiej się sprawują w praktycznych zastosowaniach.

Zużycie pamięci operacyjnej

Innym problemem dotyczącym sieci CNN jest fakt, że warstwy splotowe wymagają olbrzymich ilości pamięci operacyjnej, zwłaszcza w trakcie uczenia, gdyż przebieg algorytmu propagacji wstecznej wymaga, aby wszystkie pośrednie wartości zostały wyliczone w czasie przebiegu do przodu.

Przeanalizujmy na przykład warstwę splotową zawierającą filtry 5×5 , generującą 200 map cech o rozmiarze 150×100 , o kroku 1 i uzupełnianiu zerami typu "same". Jeżeli dane wejściowe mają postać obrazu RGB (trójkanałowego) o wymiarach 150×100 , to liczba parametrów wynosi $(5 \times 5 \times 3 + 1) \times 200 = 15 \cdot 200$ (wartość +1 odpowiada członom obciążenia), czyli całkiem mało w porównaniu do w pełni połączonej warstwy⁷. Jednak każda z 200 map cech składa się z 150×100 neuronów, z których każdy musi obliczyć sumę ważoną $5 \times 5 \times 3 = 75$ wejść: oznacza to 225 milionów operacji zmiennoprzecinkowych. Znowu nie tak źle, jak w przypadku w pełni połączonej warstwy, ale i tak wymagające obliczeniowo. Dodatkowo, jeśli mapy cech są reprezentowane przez wartości typu float32, to wynik warstwy splotowej będzie zajmował $200 \times 150 \times 100 \times 32 = 96$ milionów bitów (12 MB) pamięci operacyjnej⁸. Dotyczy to zaledwie jednej próbki! Jeżeli minigrupa danych uczących składa się ze 100 przykładów, to ta warstwa zajmie 1,2 GB pamięci RAM!

W trakcie procesu wnioskowania (tj. podczas wyliczania prognoz dla nowej próbki) pamięć operacyjna zajmowana przez jedną warstwę może zostać zwolniona od razu po uzyskaniu wyników przez następną warstwę, dlatego potrzebuje tylko tyle pamięci RAM, żeby zmieścić dwie sąsiadujące ze sobą warstwy. Jednakże na etapie uczenia wszystkie wartości wyliczone podczas przebiegu do przodu muszą zostać zachowane na potrzeby algorytmu propagacji wstecznej, dlatego mimo wszystko minimalna ilość pamięci operacyjnej musi być równa (przynajmniej) całkowitej pamięci wymaganej do zmieszczenia wszystkich warstw.

⁷ W pełni połączona warstwa składająca się z 150×100 neuronów, z których każda byłaby połączona z $150 \times 100 \times 3$ wejściami, zawierałaby $150^2 \times 100^2 \times 3 = 675$ milionów parametrów!

⁸ W międzynarodowym układzie jednostek miar fizycznych (SI) 1 MB = 1000 kB = 1000 × 1000 bajtów = 1000 × 1000 × 8 bitów.



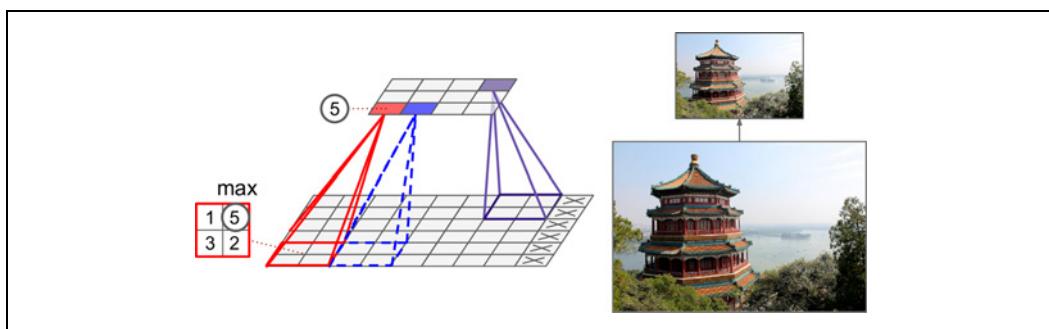
Jeżeli proces uczenia jest przerwany z powodu niedostatku pamięci, możesz spróbować zmniejszyć rozmiar minigrupy. Ewentualnie spróbuj zredukować wymiarowość za pomocą kroku lub usuwając kilka cech. Jeszcze innym rozwiązaniem jest zmniejszenie dokładności danych z 32 na 16 bitów. Zawsze możesz również rozdzielić obliczenia pomiędzy wiele urządzeń.

Przejdzmy teraz do drugiego elementu budulcowego sieci CNN — **warstwy łączącej**.

Warstwa łącząca

Gdy już wiemy, jak działa warstwa splotowa, zrozumienie mechanizmu kryjącego się za **warstwami łączącymi** (ang. *pooling layers*) nie powinno stanowić problemu. Ich celem jest **podpróbkowanie** (ang. *subsample*; tj. zmniejszenie) obrazu wejściowego w celu zredukowania obciążenia obliczeniowego, wykorzystania pamięci i liczb parametrów (a tym samym ograniczenia ryzyka przetrenowania).

Podobnie jak w przypadku warstw splotowych, każdy neuron stanowiący część warstwy łączącej łączy się z wyjściami określonej liczby neuronów warstwy poprzedniej, mieszczącej się w obszarze niewielkiego, prostokątnego pola recepcyjnego. Podobnie jak wcześniej, musimy definiować tu rozmiar tego pola, wartość kroku, rodzaj uzupełniania zerami itd. Jednakże warstwa łącząca nie zawiera żadnych wag; jej jedynym zadaniem jest gromadzenie danych wejściowych za pomocą jakiejś funkcji agregacyjnej, np. maksymalizującej lub uśredniającej. Na rysunku 14.8 widzimy najpopularniejszy rodzaj — **maksymalizującą warstwę łączącą** (ang. *max pooling layer*). W tym przykładzie korzystamy z **jądra łączącego**⁹ (ang. *pooling kernel*) o rozmiarze 2×2 , kroku o wartości 2 i z pominięciem uzupełniania zerami. Zwróć uwagę, że jedynie maksymalna wartość z każdego jądra zostaje przekazana do następnej warstwy natomiast pozostałe wartości wejściowe zostają odrzucone. Na przykład w lewym dolnym polu recepcyjnym na rysunku 14.8 widzimy wartości wejściowe 1, 5, 3, 2, zatem tylko wartość maksymalna (5) zostanie przekazana do następnej warstwy. Z powodu kroku równego 2 obraz wejściowy ma szerokość i wysokość o połowę mniejsze w porównaniu do obrazu wejściowego (zaokrąglamy tu w dół, ponieważ nie korzystamy z uzupełniania zerami).



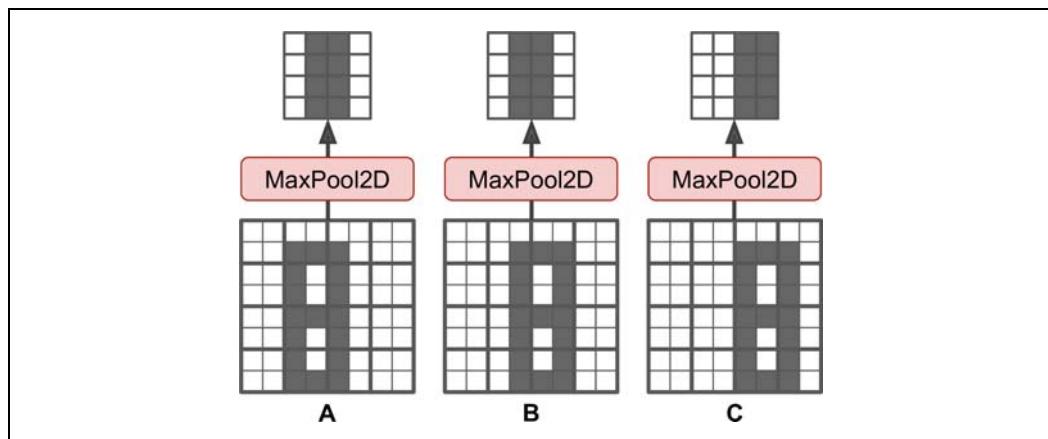
Rysunek 14.8. Maksymalizująca warstwa łącząca (jądro łączące: 2×2 , krok: 2, brak uzupełniania zerami)

⁹ Dotychczas omawiane jądra miały wagę, ale nie dotyczy to jąder łączących — stanowią one jedynie bezstanowe ramki przesuwne.



Warstwa łącząca zazwyczaj działa niezależnie na każdym kanale wejściowym, dlatego głębia wyjściowa jest taka sama jak głębia wejściowa.

Oprócz ograniczania liczby obliczeń, zużycia pamięci i liczby parametrów maksymalizująca warstwa łącząca wprowadza także pewien stopień **niezmienniczości** w stosunku do drobnych przesunięć, co widać na rysunku 14.9. Zakładamy tu, że piksele jasne mają mniejszą wartość od pikseli ciemnych, trzy obrazy (A, B i C) przechodzą przez maksymalizującą warstwę łączącą o jądrze 2×2 i kroku równym 2. Obrazy B i C wyglądają tak samo jak obraz A, ale są przesunięte o, odpowiednio, jeden i dwa piksele w prawo. Jak widać, rezultaty wygenerowane w maksymalizującej warstwie łączącej z obrazów A i B są identyczne. Na tym polega niezmienniczość przesunięć (ang. *translation invariance*). W przypadku obrazu C wynik jest odmienny: jest on przesunięty o jeden piksel w prawo (nadal jednak pozostaje niezmieniony w mniej więcej 75%). Poprzez wstawianie maksymalizującej warstwy łączącej co kilka warstw sieci CNN możliwe jest uzyskanie ograniczonej niezmienniczości przesunięć w większej skali. Ponadto warstwa ta zapewnia niewielki stopień niezmienniczości rotacyjnej i drobną niezmienniczość skalowania. Tego typu niezmienniczość (mimo że jest ograniczona) przydaje się wszędzie tam, gdzie prognozy nie powinny być zależne od tych szczegółów, na przykład w zadaniach klasyfikacji.



Rysunek 14.9. Niezmienniczość związana z drobnymi przesunięciami

Jednak maksymalizująca warstwa łącząca jest niepozbawiona również wad. Przede wszystkim jest ona bardzo niszczycielska: nawet w przypadku niewielkiego jądra o rozmiarze 2×2 i kroku o wartości 2 dane wyjściowe będą dwukrotnie mniejsze w każdym kierunku (zatem obszar obrazu będzie zmniejszony czterokrotnie), co oznacza porzucenie 75% wartości wejściowych. Z kolei w pewnych zastosowaniach niezmienniczość jest niepożądana, dajmy na to w segmentacji semantycznej (zadaniu klasyfikowania każdego piksela obrazu zgodnie z jego przynależnością do danego obiektu; później wrócimy do tego zagadnienia): jest oczywiste, że jeżeli obraz wejściowy zostanie przesunięty o jeden piksel w prawo, to wynik również powinien być przesunięty w taki sam sposób. Wówczas celem staje się **ekwiwariancja** (ang. *equivariance*), a nie niezmienniczość: mała zmiana w sygnale wejściowym powinna prowadzić do powiązanej z nią niewielkiej zmiany w sygnale wyjściowym.

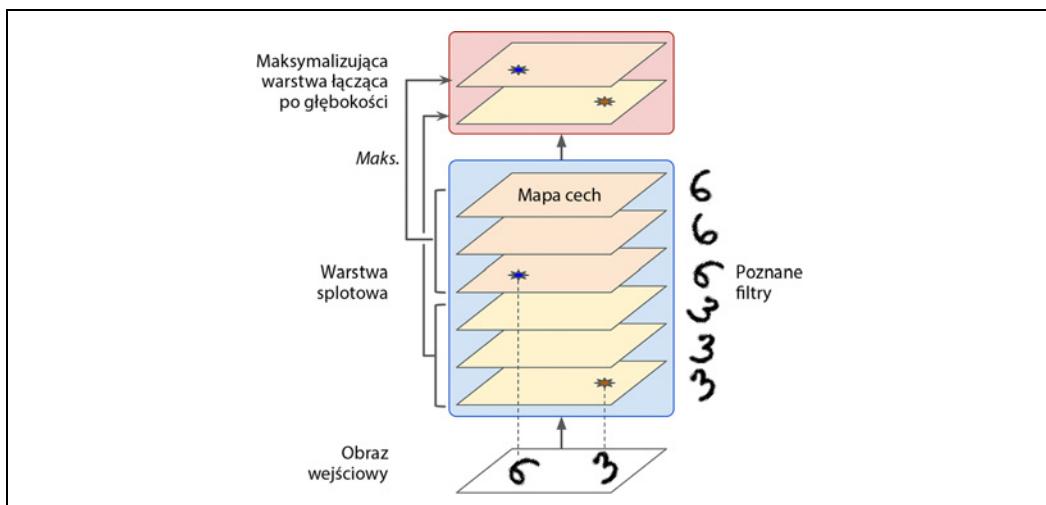
Implementacja w module TensorFlow

Implementacja maksymalizującej warstwy łączącej w module TensorFlow nie jest zbyt skomplikowana. Poniższy listing generuje maksymalizującą warstwę łączącą o jądrze 2×2 . Domyślnie wartość kroku jest taka sama jak rozmiar jądra, zatem w tej warstwie będzie stosowany krok 2 (zarówno w poziomie, jak i w pionie). W domyśle wyznaczone jest uzupełnianie zerami typu "valid" (czyli bez uzupełniania):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

Aby utworzyć **uśredniającą warstwę łączającą** (ang. *average pooling layer*), wystarczy podmienić warstwę MaxPool2D na AvgPool2D. Jak łatwo się domyślić, działa ona bardzo podobnie do maksymalizującej warstwy łączającej, tyle że tutaj jest obliczana średnia, a nie wartość maksymalna. Kiedyś warstwy uśredniające były bardzo popularne, obecnie jednak wybierane są głównie warstwy maksymalizujące z powodu lepszej wydajności ogólnej. Może wydawać się to zaskakujące, gdyż obliczanie średniej zazwyczaj powoduje utratę mniejszej ilości informacji w porównaniu do obliczania wartości maksymalnej. Z drugiej jednak strony warstwa maksymalizująca zachowuje wyłącznie najistotniejsze cechy i ignoruje wszystkie cechy nieistotne, zatem kolejne warstwy otrzymują coraz czystszy sygnał. Ponadto warstwa maksymalizująca cechuje się większą niezmienniczością przesunięcia i wymaga nieco mniej obliczeń.

Zwróci uwagę, że operacje zarówno maksymalizacji, jak i uśredniania mogą być przeprowadzane raczej wzdłuż wymiaru głębokości, a nie wymiarów przestrzennych, chociaż rozwiązywanie to nie jest spotykane tak często. Dzięki temu sieć CNN może uczyć się niezmienniczości względem różnych cech. Na przykład mogłyby „opracować” wiele filtrów, z których każdy rozpoznawałby inny rodzaj obrotu w ramach tego samego wzorca (np. w piśmie odreżnym; rysunek 14.10), natomiast maksymalizująca warstwa łącząca po głębokości sieci gwarantowałaby, że sygnał wyjściowy pozostałby niezmieniony bez względu na obrót. W ten sposób sieć splotowa może nauczyć się niezmienniczości dla dowolnego aspektu: grubości, jasności, nachylenia, barwy itd.



Rysunek 14.10. Maksymalizująca warstwa łącząca po głębokości pomaga sieci splotowej uczyć się niezmienniczości

Interfejs Keras nie zawiera maksymalizującej warstwy łączącej po głębokości, w przeciwieństwie do ogólnego interfejsu uczenia głębokiego w module Keras: wystarczy użyć funkcji `tf.nn.max_pool()` i wyznaczyć rozmiar jądra oraz kroki jako 4-krotki (tzn. krotki o rozmiarze 4). Pierwsze trzy wartości takiej krotki powinny być równe 1, co oznacza, że rozmiary jądra i kroku w wymiarach grupy, wysokości i szerokości powinny być równe 1. Za pomocą ostatniej wartości wyznaczasz rozmiary jądra i krotki dla wymiaru głębokości — może to być na przykład wartość 3 (musi być ona dzielnikiem głębokości wejściowej; nie będzie działała, jeżeli wcześniejsza warstwa będzie generowała 20 map cech, ponieważ 20 nie stanowi wielokrotności liczby 3):

```
output = tf.nn.max_pool(images,
                        kszie=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

Jeżeli chcesz dołączyć tę warstwę w modelach Keras, umieść ją wewnątrz warstw Lambda (lub utwórz niestandardową warstwę Keras):

```
depth_pool = keras.layers.Lambda(
    lambda X: tf.nn.max_pool(X, kszie=(1, 1, 1, 3), strides=(1, 1, 1, 3),
                             padding="valid"))
```

Ostatnim rodzajem warstwy łączącej często spotykanym we współczesnych architekturach jest **globalna uśredniająca warstwa łącząca** (ang. *global average pooling layer*). Mechanizm jej działania jest całkiem odmienny: oblicza ona jedynie średnią każdej mapy cech (przypomina to działanie uśredniającej warstwy łączącej, w której jądro ma takie same wymiary przestrzenne jak dane wejściowe). Oznacza to, że generuje ona na wyjściu pojedynczą wartość na każdą mapę cech i na każdy przykład. Jest to oczywiście rozwiążanie skrajnie destrukcyjne (większość informacji zawartych w mapie cech zostaje utraconych), ale, jak przekonasz się w dalszej części rozdziału, bywa przydatne na wyjściu modelu. Aby utworzyć tę warstwę, wystarczy skorzystać z klasy `keras.layers.GlobalAvgPool2D`:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

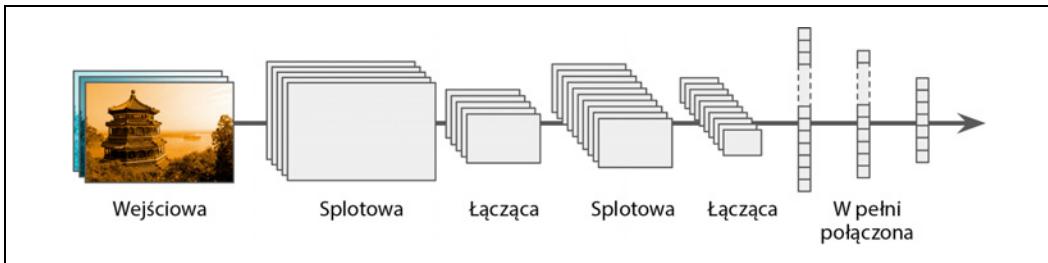
Jest ona równoważna prostej warstwie Lambda, która oblicza średnią po wymiarach przestrzennych (wysokości i szerokości):

```
global_avg_pool = keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

Znasz już wszystkie cegiełki potrzebne do zbudowania splotowej sieci neuronowej. Poskładajmy je teraz w całość.

Architektury splotowych sieci neuronowych

Typowe architektury CNN składają się zazwyczaj z kilku warstw splotowych (najczęściej naprzemienne z warstwami ReLU), warstwy łączącej, po niej znowu kilku warstw splotowych (+ReLU), kolejnej warstwy łączącej itd. Obraz stopniowo maleje, przechodząc przez kolejne warstwy sieci, ale jednocześnie zwiększa się jego głębia (np. poprzez obecność map cech) dzięki warstwom splotowym (rysunek 14.11). Na samym szczycie sieci zostaje umieszczona klasyczna sieć neuronowa, zawierająca kilka w pełni połączonych warstw (+ReLU), a ostatnia z nich wylicza prognozy (może byćnią np. warstwa softmax, oszacowująca prawdopodobieństwo przynależności próbki do danej klasy).



Rysunek 14.11. Typowa architektura sieci CNN



Często popełnianym błędem jest stosowanie zbyt dużych jąder splotowych. Na przykład zamiast korzystać z warstwy splotowej o jądrze 5×5 wprowadź dwie warstwy o jądrach 3×3 — będzie wymaganych mniej parametrów i obliczeń, a ponadto przeważnie skuteczność takiego modelu jest większa. Jedynym wyjątkiem stanowi pierwsza warstwa splotowa: może ona mieć zazwyczaj duże jądro (np. 5×5), najczęściej o kroku równym 2 lub większym. Redukujemy w ten sposób rozmiary obrazu bez ryzyka utraty dużej ilości informacji, a skoro obraz wejściowy zasadniczo składa się tylko z trzech kanałów, to operacja ta nie będzie zbyt kosztowna.

Oto implementacja prostej sieci splotowej, przetwarzającej zestaw danych Fashion MNIST (omówiony w rozdziale 10.):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

Przyjrzyjmy się temu modelowi:

- Pierwsza warstwa zawiera zdefiniowane 64 dość duże filtry (7×7), ale nie wprowadzamy tu kroków, ponieważ obrazy nie są bardzo duże. Wyznaczamy również `input_shape=[28, 28, 1]`, gdyż obrazy mają rozmiar 28×28 pikseli i jeden kanał barw (tzn. są czarno-białe).
- Następnie widzimy maksymalizującą warstwę łączącą, która zawiera jądro o rozmiarze 2, co oznacza, że obraz zostaje zmniejszony o połowę w każdym wymiarze przestrzennym.
- Powtarzamy tę strukturę dwa razy: dwie warstwy splotowe, a po nich maksymalizująca warstwa łącząca. W przypadku większych obrazów możemy powtórzyć tę strukturę kilkakrotnie (liczba powtórzeń jest hiperparametrem, który możemy stroić).

- Zwróć uwagę, że z każdą warstwą wzrasta liczba filtrów (początkowo mamy ich 64, później 128 i 256), co jest logiczne, ponieważ nierzaz liczba ogólnych cech jest dość mała (np. małych okręgów czy linii poziomych), ale istnieje mnóstwo różnych sposobów łączenia ich w bardziej szczegółowe cechy. Powszechnym rozwiązaniem jest podwajanie liczby filtrów po każdej warstwie łączającej: skoro warstwa łączająca dzieli każdy wymiar przestrzenny przez współczynnik 2, możemy sobie pozwolić na podwojenie liczby map cech w następnej warstwie bez obaw o nagły wzrost liczby parametrów, zajętego miejsca w pamięci roboczej czy obciążenia obliczeniowego.
- Na końcu widzimy warstwę w pełni połączoną, składającą się z dwóch gęstych warstw ukrytych i gęstej warstwy wyjściowej. Zwróć uwagę, że musimy spłaszczyć dane wejściowe, ponieważ sieć gęsta oczekuje jednowymiarowej tablicy cech dla każdego przykładu. W celu zmniejszenia przetrenowania dodajemy także dwie warstwy porzucania o współczynniku porzucania równym 50%.

Sieć ta osiąga 92% dokładności na zbiorze testowym. Nie jest to może mistrzostwo świata, ale i tak dobry wynik, zdecydowanie lepszy od uzyskanego za pomocą gęstych sieci neuronowych z rozdziału 10.

Przez lata wymyślono wiele wariantów tej architektury, co doprowadziło do zdumiewających postępów w dziedzinie uczenia maszynowego. Dobrym wskaźnikiem tych postępów są poziomy błędów uzyskiwane w konkursach takich jak wyzwanie ILSVRC ImageNet (<http://image-net.org/>). Wartość pięciu najlepszych poziomów błędów (ang. *top-5 error rate*) dla klasyfikacji obrazów w tych zawodach w ciągu zaledwie sześciu lat zmalała z 26% do mniej niż 2,3%. Błąd ten stanowi liczbę obrazów testowych, których pięć najlepszych prognoz danego modelu nie sklasyfikowało prawidłowo. Obrazy są duże (mają wysokość 256 pikseli) i istnieje tysiąc klas, pomiędzy którymi różnice bywają nierzaz bardzo subtelne (spróbuj rozpoznawać 120 ras psów). Obserwacja ewolucji zwycięskich modeli stanoi dobry sposób zrozumienia mechanizmu działania sieci CNN.

Przeanalizujemy najpierw klasyczną architekturę LeNet-5 (z 1998 roku), a następnie trzy zwycięskie modele zawodów ILSVRC: AlexNet (2012), GoogLeNet (2014) i ResNet (2015).

LeNet-5

Architektura LeNet-5 (<http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf>)¹⁰ stanowi prawdopodobnie najbardziej znany przykład sieci CNN. Jak już wspomniałem wcześniej, została stworzona przez Yanna LeCuna w 1998 roku i jest powszechnie stosowana w rozpoznawaniu odręcznie pisanych cyfr (MNIST). Jej budowa została zaprezentowana w tabeli 14.1.

Zwróćmy uwagę na kilka szczegółów:

- Obrazy MNIST mają rozmiar 28×28 pikseli, ale przed wysłaniem do sieci zostają uzupełnione zerami do rozmiaru 32×32 i znormalizowane. W dalszej części sieci nie jest używane uzupełnianie zerami, dlatego rozmiar maleje wraz z kolejnymi warstwami sieci.

¹⁰ Yann LeCun i in., *Gradient-Based Learning Applied to Document Recognition*, „Proceedings of the IEEE” 86, no. 11 (1998), s. 2278 – 2324.

Tabela 14.1. Architektura sieci LeNet-5

Warstwa	Typ	Mapy	Rozmiar	Rozmiar jądra	Krok	F. aktywacji
Out	W pełni połączona	—	10	—	—	RBF
F6	W pełni połączona	—	84	—	—	tanh
C5	Splotowa	120	1×1	5×5	1	tanh
S4	Uśr. łącząca	16	5×5	2×2	2	tanh
C3	Splotowa	16	10×10	5×5	1	tanh
S2	Uśr. łącząca	6	14×14	2×2	2	tanh
C1	Splotowa	6	28×28	5×5	1	tanh
In	Wejściowa	1	32×32	—	—	—

- Uśredniające warstwy łączące są nieco bardziej skomplikowane niż zazwyczaj: każdy neuron wylicza średnią z wejść, po czym mnoży wynik przez modyfikowalny współczynnik (po jednym na każdą mapę) i dodaje modyfikowalny człon obciążenia (również po jednym na mapę), po czym dopiero stosuje funkcję aktywacji.
- Większość neuronów w mapach C3 jest połączona z neuronami jedynie z trzech lub czterech map S2 (zamiast ze wszystkich sześciu). Szczegóły znajdziesz w oryginalnej dokumentacji architektury w artykule (tabela 1. na stronie 8.).
- Warstwa wyjściowa jest dość specyficzna: neurony nie wyliczają iloczynu skalarnego wejść i wektora wag, lecz kwadrat odległości euklidesowej pomiędzy wektorem wejść a wektorem wag. Każde wyjście mierzy stopień przynależności obrazu do określonej klasy cyfry. W tym wypadku jest preferowana funkcja kosztu w postaci entropii krzyżowej, a złe prognozy są karane znacznie surowiej, co prowadzi do większych gradientów, a tym samym szybszego uzyskania zbieżności.

Na stronie domowej Yanna LeCuna (<http://yann.lecun.com/exdb/lenet/index.html>) znajdują się znakomite prezentacje algorytmu LeNet-5 klasyfikującego cyfry.

AlexNet

Architektura splotowej sieci neuronowej AlexNet (<https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>)¹¹ wygrała w 2012 roku konkurs ImageNet ILSVRC z dość dużą przewagą nad konkurencją: osiągnęła wartość błędu top-5 rzędu 17%, podczas gdy zdobywca drugiego miejsca uzyskał zaledwie 26%! Jej autorami są Alex Krizhevsky (stąd nazwa), Ilya Sutskever i Geoffrey Hinton. Przypomina ona sieć LeNet-5, ale jest znacznie większa i bardziej wielowarstwowa; poza tym w niej po raz pierwszy umieszczano bezpośrednio po sobie poszczególne warstwy splotowe (do tego czasu naprzemienne wstawiano warstwy splotowe i łączące). Architektura tej sieci została ukazana w tabeli 14.2.

¹¹ Alex Krizhevsky i in., *ImageNet Classification with Deep Convolutional Neural Networks*, „Proceedings of the 25th International Conference on Neural Information Processing Systems” 1 (2012), s. 1097 –1105.

Tabela 14.2. Architektura sieci AlexNet

Warstwa	Typ	Mapy	Rozmiar	Rozmiar jądra	Krok	Uzupełnianie zerami	F. aktywacji
Out	W pełni połączona	—	1000	—	—	—	Softmax
F10	W pełni połączona	—	4096	—	—	—	ReLU
F9	W pełni połączona	—	4096	—	—	—	ReLU
S8	Maks. łącząca	256	6×6	3×3	2	valid	—
C7	Splotowa	256	13×13	3×3	1	same	ReLU
C6	Splotowa	384	13×13	3×3	1	same	ReLU
C5	Splotowa	384	13×13	3×3	1	same	ReLU
S4	Maks. łącząca	256	13×13	3×3	2	valid	—
C3	Splotowa	256	27×27	5×5	1	same	ReLU
S2	Maks. łącząca	96	27×27	3×3	2	valid	—
C1	Splotowa	96	55×55	11×11	4	same	ReLU
In	Wejściowa	3 (RGB)	224×224	—	—	—	—

W celu zmniejszenia przetrenowania autorzy wprowadzili dwie omówione wcześniej techniki regularyzacji. Najpierw w czasie uczenia użyli omówionej w rozdziale 11. metody porzucania (ze współczynnikiem porzucania o wartości 50%) wobec wyjść warstw F9 i F10. Następnie **dogenerowali dane** poprzez losowe przesuwanie obrazów o różne wartości, obracanie ich w poziomie i zmienianie jasności.

Sieć AlexNet wykorzystuje również rywalizacyjny etap normalizacji tuż po zastosowaniu funkcji ReLU w warstwach C1 i C3 — jest to tak zwana **normalizacja odpowiedzi lokalnej** (ang. *local response normalization* — LRN): neurony o największych wagach (najsiłniej uaktywniane) hamują neurony znajdujące się w tym samym położeniu, ale w sąsiadujących mapach cech (tego typu rywalizacyjna aktywacja występuje w neuronach biologicznych). W ten sposób różne mapy cech dążą do specjalizacji, przez co zwiększa się ich różnorodność i przetwarzana jest większa przestrzeń cech, co ostatecznie prowadzi do usprawnienia procesu uogólniania wyników. Równanie 14.2 ukazuje normalizację LRN.

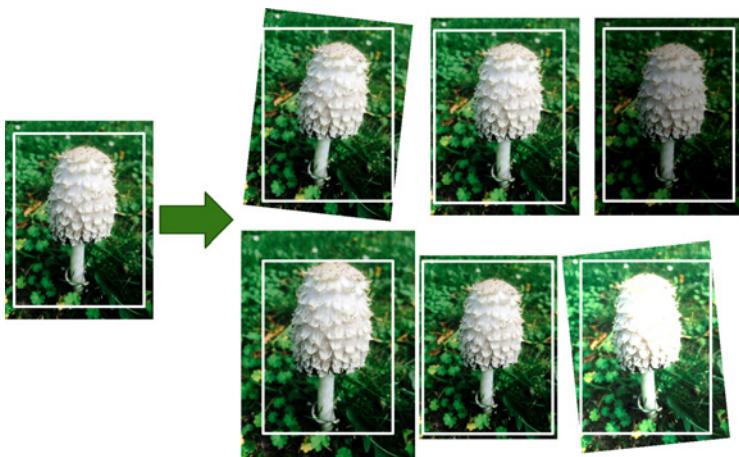
Równanie 14.2. Normalizacja odpowiedzi lokalnej (LRN)

$$b_i = a_i \left(k + \alpha \sum_{j=j_{niska}}^{j_{wysoka}} a_j^2 \right)^{-\beta} \quad \text{gdzie} \quad \begin{cases} j_{wysoka} = \min \left(i + \frac{r}{2}, f_n - 1 \right) \\ j_{niska} = \max \left(0, i - \frac{r}{2} \right) \end{cases}$$

Dogenerowanie danych

Dogenerowanie danych (ang. data augmentation) zwiększa rozmiar zbioru uczącego poprzez generowanie wielu realistycznych wariantów każdego przykładu uczącego. Metoda ta ogranicza przetrenowanie modelu, co sprawia, że jest uznawana za technikę regularyzacji. Sztuka polega na wygenerowaniu jak najrealistyczniejszych przykładów uczących — w idealnym przypadku człowiek nie powinien być w stanie odróżnić dogenerowanych próbek od oryginałów. Co więcej, zwykle dodanie szumu białego nic tu nie pomoże — model powinien być w stanie uczyć się z wprowadzanych modyfikacji (szum biały to uniemożliwia).

Przykładowo możesz w różnym stopniu nieznacznie przesuwać i obracać każde zdjęcie w zestawie uczącym oraz zmieniać jego rozmiar i tak zmodyfikowane fotografie dodawać do zbioru uczącego (rysunek 14.12). W ten sposób model musi uczyć się brać pod uwagę położenie, orientację i rozmiar obiektów na zdjęciach. Jeśli chcesz, aby model miał większą tolerancję na warunki oświetlenia zdjęcia, możesz w podobny sposób dogenerować fotografie o różnych stopniach kontrastu. Zasadniczo możesz także obracać zdjęcia w osi poziomej (oprócz tekstu i innych asymetrycznych obiektów). Poprzez łączenie poszczególnych rodzajów transformacji jesteś w stanie znacznie powiększyć rozmiar zbioru danych uczących.



Rysunek 14.12. Dogenerowanie nowych danych z istniejących przykładów

W tym równaniu:

- b_i jest znormalizowanym wynikiem neuronu znajdującego się na mapie cech i , w jakimś rzędzie u i kolumnie v (zwróć uwagę, że w tym równaniu bierzemy pod uwagę wyłącznie neurony zlokalizowane w tym rzędzie i tej kolumnie, dlatego u i v nie są ukazane);
- a_i to aktywacja tego neuronu po zastosowaniu funkcji ReLU, ale jeszcze przed normalizacją;
- k, α, β i r to hiperparametry; k jest nazywany **obciążeniem**, natomiast r — **promieniem głębokości** (ang. *depth radius*);
- f_n jest liczbą map cech.

Na przykład jeśli $r = 2$, a dany neuron cechuje duży stopień aktywacji, będzie hamował aktywację neuronów znajdujących się w mapach cech znajdujących się bezpośrednio powyżej i poniżej jego własnej mapy.

Hiperparametry w sieci AlexNet mają następujące wartości: $r = 2$, $\alpha = 0,00002$, $\beta = 0,75$ i $k = 1$. Możemy zaimplementować normalizację odpowiedzi lokalnej za pomocą operacji `tf.nn.local_response_normalization()` (jeżeli chcesz ją wykorzystać w modelu Keras, wstaw ją do warstwy Lambda).

Matthew Zeiler i Rob Fergus stworzyli modyfikację sieci AlexNet o nazwie **ZF Net** (<https://arxiv.org/abs/1311.2901>)¹², dzięki której wygrali konkurs ILSVRC w 2013 roku. Różni się ona od pierwotnej architektury jedynie wartościami niektórych hiperparametrów (liczbą map cech, rozmiarem jądra, wartością kroku itd.).

GoogLeNet

Architekturę sieci GoogLeNet (https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html) zaprojektował Christian Szegedy wraz z innymi członkami zespołu Google Research¹³ — algorytm ten wygrał konkurs ILSVRC w 2014 roku, zmniejszając poziom błędu top-5 do wartości poniżej 7%. Taka znakomita wydajność została częściowo osiągnięta dzięki temu, że sieć ta jest znacznie głębsza od poprzednich splotowych sieci neuronowych (rysunek 14.14). Stworzenie takiej architektury stało się możliwe dzięki wprowadzeniu podsieci zwanych **modułami incepcyjnymi** (ang. *inception modules*)¹⁴, za pomocą których parametry zaczęły być używane znacznie wydajniej niż u konkurencji: sieć GoogLeNet w istocie zawiera dziesięciokrotnie mniej parametrów od architektury AlexNet (w przybliżeniu 6 milionów zamiast 60 milionów).

Rysunek 14.13 przedstawia architekturę modułu incepcyjnego. Notacja $3 \times 3 + 1(S)$ oznacza, że warstwa wykorzystuje jądro o rozmiarze 3×3 , krok o wartości 1 i uzupełnianie zerami typu "same". Sygnał zostaje najpierw skopiowany i rozesłany do czterech różnych warstw. Wszystkie warstwy splotowe korzystają z funkcji aktywacji ReLU. Zwróć uwagę, że drugi zestaw warstw splotowych zawiera jądra o odmiennych rozmiarach (1×1 , 3×3 i 5×5), co pozwala im wyłapywać wzorce w różnych skalach. Zauważ także, że w każdej pojedynczej warstwie jest stosowany krok o rozmiarze 1 i uzupełnianie zerami typu "same" (dotyczy to również maksymalizującej warstwy łączącej), zatem wyjścia mają zawsze taki sam rozmiar jak wejścia. W ten sposób możliwe staje się powiązanie wszystkich wyjść w wybranym głębokości w ostatniej **warstwie łączącej w głęb** (ang. *depth concat layer*; tj. nałożenie na siebie map cech pochodzących ze wszystkich czterech górnych warstw splotowych). Ta warstwa łącząca może zostać zaimplementowana w module TensorFlow za pomocą operacji `tf.concat()` z parametrem `axis=3` (oznaczającym głębokość).

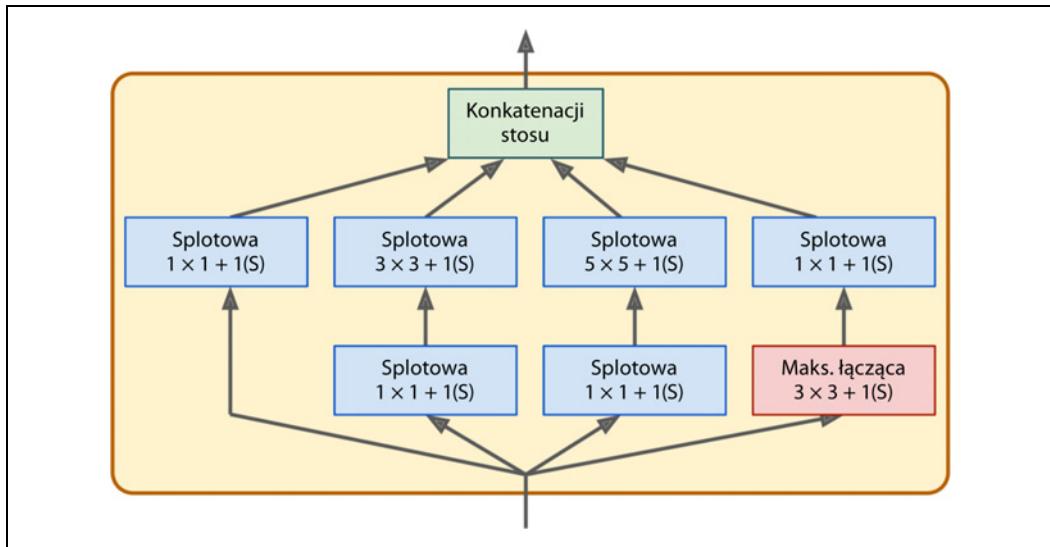
Zastanawiasz się pewnie, dlaczego moduły incepcyjne zawierają warstwy splotowe o jądrach 1×1 . Przecież nie są one w stanie wychwytywać żadnych cech, jeśli pobierają jednorazowo tylko po jednym pikselu? W rzeczywistości warstwy te spełniają trzy zadania:

- Nie są w stanie wychwytywać wzorców przestrzennych, ale są w stanie znajdować je w wybranym głębokości.

¹² Matthew D. Zeiler i Rob Fergus, *Visualizing and Understanding Convolutional Networks*, „Proceedings of the European Conference on Computer Vision” (2014), s. 818 – 833.

¹³ Christian Szegedy i in., *Going Deeper with Convolutions*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2015), s. 1 – 9.

¹⁴ W filmie *Incepcja* z 2010 roku bohaterowie schodzą coraz głębiej w kolejne warstwy snu — stąd wzięła się nazwa modułów.



Rysunek 14.13. Moduł incepcyjny

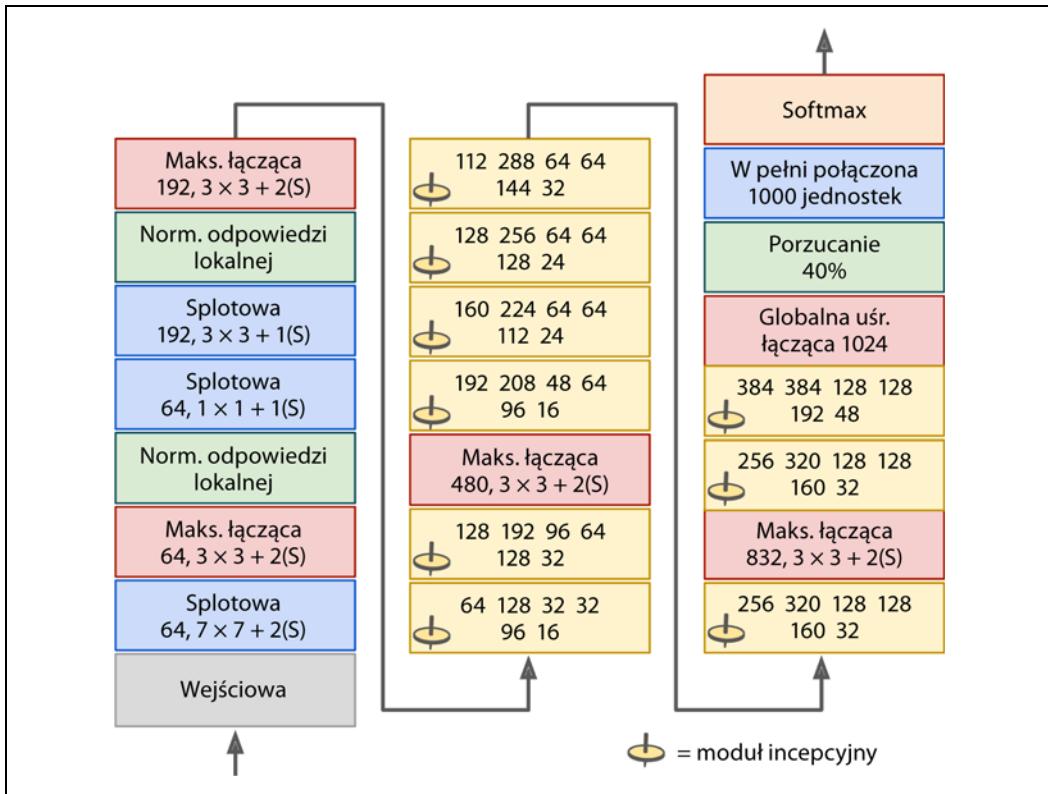
- Są one skonfigurowane do wysyłania na wyjście wielu mniejszych map cech, niż dociera na wejściu, zatem pełnią rolę **warstw ograniczających** (ang. *bottleneck layers*), czyli redukują wymiarowość. Redukujemy w ten sposób koszt obliczeniowy oraz liczbę parametrów, a także przyspieszamy proces uczenia i poprawiamy zdolność uogólniania.
- Każda para warstw splotowych ($[1 \times 1, 3 \times 3]$ i $[1 \times 1, 5 \times 5]$) zachowuje się jak pojedyncza, potężna sieć konwolucyjna, zdolna do wylapywania bardziej skomplikowanych wzorców. Rzeczywiście, zamiast przepuszczać prosty klasyfikator liniowy przez rysunek (mechanizm działania pojedynczej warstwy splotowej), para warstw konwolucyjnych wykonuje przebieg dwuwarstwowej sieci neuronowej.

Mówiąc krótko, możemy uznać moduł incepcyjny za warstwę splotową „na sterydach”, zdolną do generowania map cech wychwytyujących skomplikowane wzorce w różnych skalach.



Liczba jąder splotowych występujących w każdej warstwie konwolucyjnej stanowi hiperparametr. Oznacza to niestety, że na każdą dodawaną warstwę incepcyjną przypada sześć dodatkowych hiperparametrów, które należy dostroić.

Przyjrzyjmy się teraz architekturze samej sieci GoogLeNet (rysunek 14.14). Przed wartością oznaczającą rozmiar jądra została umieszczona liczba map cech generowanych przez każdą warstwę splotową i warstwę łączącą. Struktura sieci jest tak głęboka, że musimy ją zaprezentować w trzech kolumnach, w rzeczywistości jednak stanowi ona jeden wysoki stos, do którego zalicza się również dziewięć modułów incepcyjnych (komórki oznaczone ikoną wirującego bączka), z których każdy zawiera kolejne trzy warstwy. Sześć liczb widocznych w każdym module incepcyjnym reprezentuje liczbę map cech uzyskiwaną z każdej warstwy splotowej tworzącej ten moduł (w takiej samej kolejności, jaką jest widoczna na rysunku 14.13). Zwróć uwagę, że wszystkie warstwy splotowe korzystają z funkcji aktywacji ReLU.



Rysunek 14.14. Architektura sieci GoogLeNet

Przeanalizujmy tę sieć:

- Pierwsze dwie warstwy redukują wysokość i długość obrazu czterokrotnie (czyli jego obszar zostaje zmniejszony szesnastokrotnie) w celu ograniczenia obciążenia obliczeniowego. W pierwszej warstwie występuje jądro o dużym rozmiarze, co pozwala zachować dużą ilość informacji.
- Następnie warstwa normalizacji odpowiedzi lokalnej sprawia, że wcześniejsze warstwy uczą się rozpoznawać bardzo zróżnicowane cechy (wiemy to z wcześniejszego opisu).
- Dalej są ułożone dwie warstwy splotowe, z których pierwsza pełni funkcję warstwy ograniczającej. Jak już wiemy, możemy interpretować tę parę jako pojedynczą, „sprytniejszą” warstwę splotową.
- Kolejna warstwa normalizacji odpowiedzi lokalnej pełni taką samą rolę jak wcześniejsza analogiczna warstwa (wychwytuje szerokie spektrum wzorców).
- Następna maksymalizująca warstwa łącząca zmniejsza dwukrotnie wymiary obrazu, ponownie po to, aby przyspieszyć obliczenia.
- Teraz widzimy wysoki stos ułożony z dziewięciu modułów incepcyjnych, pomiędzy które są wplecone dwie maksymalizujące warstwy łączące, redukujące wymiarowość i przyspieszające działanie sieci.
- Widoczna dalej globalna uśredniająca warstwa łącząca daje na wyjściu średnią każdej mapy cech: pozbywamy się w ten sposób pozostałych informacji przestrzennych, ale nic nie szkodzi,

ponieważ na tym etapie i tak nie pozostało ich wiele. Istotnie, zazwyczaj oczekiwany rozmiar wejściowy obrazów dla sieci GoogLeNet to 224×224 piksele, zatem po pięciu maksymalizujących warstwach łączących, z których każda dzieli wysokość i szerokość przykładu przez 2, uzyskujemy mapy cech o rozmiarach 7×7 . Co więcej, jest to zadanie klasyfikowania, nie lokalizowania, dlatego położenie obiektu na obrazie nie ma znaczenia. Dzięki redukcji wymiarowości w tej warstwie możemy zrezygnować z kilku górnych, w pełni połączonych warstw (w przeciwieństwie do np. sieci AlexNet), co oznacza znacznie mniejszą liczbę parametrów w sieci i zmniejszenie ryzyka przetrenowania modelu.

- Ostatnie warstwy nie wymagają szczegółowego wyjaśniania: porzucanie służy do regularyzacji, a następnie w pełni połączona warstwa (składająca się z tysiąca jednostek, ponieważ mamy do czynienia z tysiącem klas) wykorzystuje funkcję aktywacji softmax do wyświetlania oszacowanych prawdopodobieństw przynależności przykładów do danej klasy.

Widoczny schemat jest nieco uproszczony: rzeczywista architektura sieci GoogLeNet zawierała również dwa dodatkowe klasyfikatory, umieszczone po trzecim i szóstym module incepcyjnym. Klasyfikatory te składały się z jednej uśredniającej warstwy łączącej, jednej warstwy splotowej, dwóch w pełni połączonych warstw i warstwy aktywacji softmax. Na etapie uczenia ich funkcja straty (zmniejszona o 70%) była dodawana do całkowitej straty systemu. Rozwiążanie to miało na celu zwalczanie problemu zanikających gradientów i regularyzację sieci. Udowodniono jednak, że wpływ tych dwóch klasyfikatorów na całą sieć był niewielki.

Naukowcy z firmy Google stworzyli kilka odmian struktury GoogLeNet, w tym takie jak Inception-v3 i Inception-v4, w których nieznacznie zmodyfikowano moduły incepcyjne, co pozwoliło uzyskać jeszcze lepszą wydajność.

VGGNet

Drugie miejsce w konkursie ILSVRC 2014 zdobyła sieć VGGNet (<https://arxiv.org/abs/1409.1556>)¹⁵, stworzona przez Karen Simonyan i Andrew Zissermana z laboratorium Visual Geometry Group (VGG) na Uniwersytecie Oksfordzkim. Miała ona bardzo klasyczną i prostą strukturę, w której po każdym dwóch czy trzech warstwach splotowych występowała warstwa łącząca (w zależności od wariantu sieci łącznie znajdowało się w niej zaledwie od 16 do 19 warstw konwolucyjnych), na końcu zaś umieszczono sieć gęstą składającą się z dwóch warstw ukrytych i warstwy wyjściowej. Wykorzystywano tu jedynie liczne filtry o rozmiarze 3×3 .

ResNet

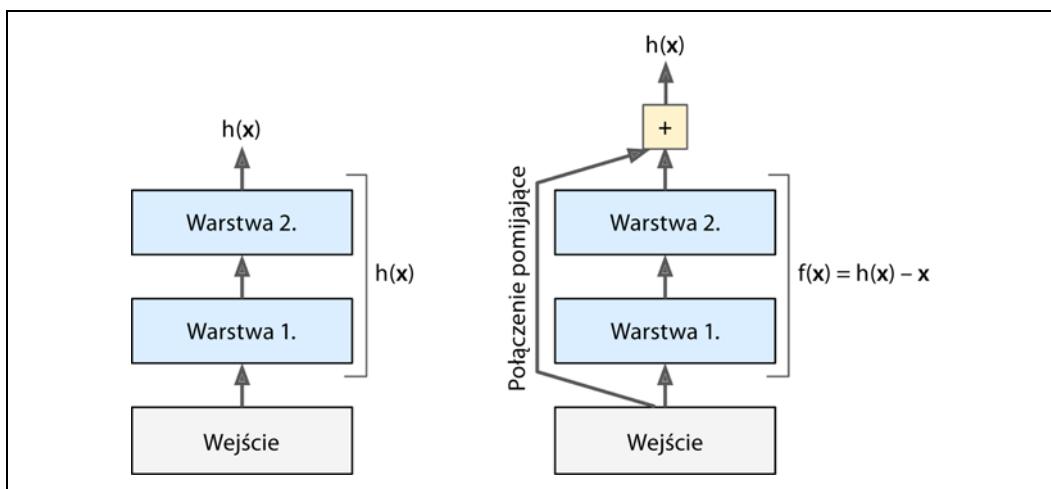
Kaiming He i in. zwyciężyli wyzwanie ILSVRC 2015 dzięki **sieci rezydualnej** (ang. *Residual Network*; w skrócie **ResNet**)¹⁶, która uzyskała zdumiewający wynik poziomu błędu top-5 poniżej 3,6%. Wykorzystano tu skrajnie głęboką architekturę CNN, składającą się ze 152 warstw (w innych wariantach

¹⁵ Karen Simonyan i Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, arXiv preprint arXiv:1409.1556 (2014).

¹⁶ Kaiming He i in., *Deep Residual Learning for Image Recognition*, arXiv preprint arXiv:1512.03385 (2015), <https://arxiv.org/abs/1512.03385>.

występuje 34, 50 lub 101 warstw). Jest to zgodne z ogólnym trendem: modele są coraz głębsze i zawierają coraz mniej parametrów. Kluczem do wyczenia tak skomplikowanej sieci okazało się wykorzystanie **połączeń pomijających** (ang. *skip connections*), zwanych również **połączonymi skrótwymi** (ang. *shortcut connections*): sygnał przekazywany do danej warstwy jest również dodawany do wyjścia warstwy znajdującej się nieco wyżej. Już wyjaśniam, dlaczego rozwiązanie to okazało się skuteczne.

Podczas uczenia sieci neuronowej naszym celem jest sprawienie, żeby dobrze odzwierciedlała ona model funkcji $h(\mathbf{x})$. Jeżeli dodamy wejście \mathbf{x} do wyjścia sieci (np. za pomocą połączenia pomijającego), to sieć zostanie zmuszona do odwzorowywania funkcji $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$, a nie samej funkcji $h(\mathbf{x})$. Jest to tak zwane **uczenie rezydualne (resztowe)** (ang. *residual learning*). Proces ten został ukazany na rysunku 14.15.

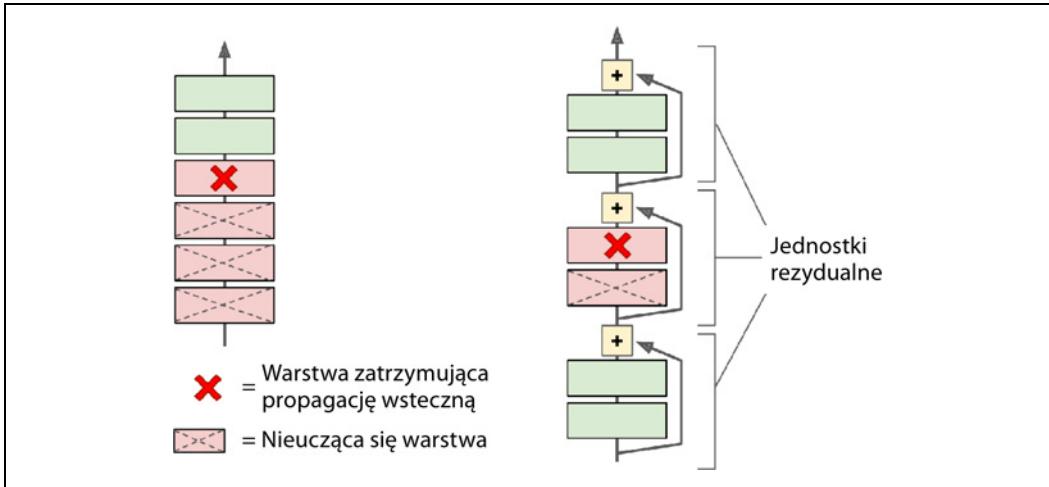


Rysunek 14.15. Uczenie resztowe

Podczas inicjalizowania standardowej sieci neuronowej jej wagi są niemal zerowe, zatem generuje ona wyniki również bliskie零. Po dodaniu połączenia pomijającego zmodyfikowana sieć przesyła na wyjście jedynie kopię danych wejściowych; inaczej mówiąc, modeluje ona początkowo funkcję tożsamościową. Jeżeli funkcja docelowa w dużym stopniu przypomina tożsamościową (często tak bywa), proces uczenia zostanie w ten sposób znacznie przyspieszony.

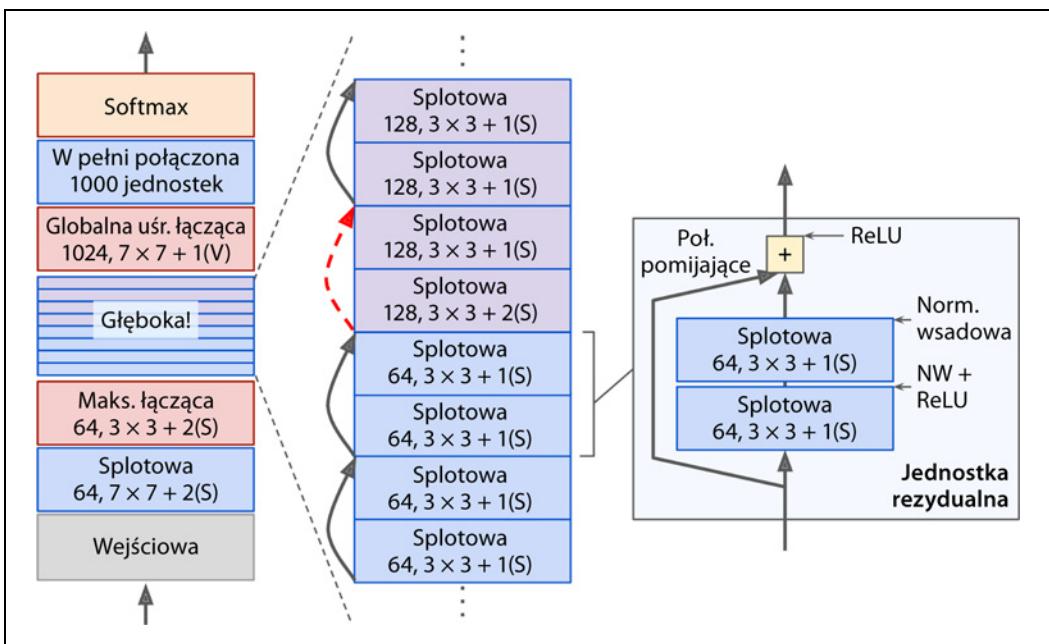
Co więcej, pod dodaniu wielu połączeń pomijających sieć zacznie robić postępy pomimo tego, że kilka warstw nie zdążyło jeszcze rozpoczęć uczenia (rysunek 14.16). Dzięki połączonym skrótwym sygnał może z łatwością rozprzestrzenić się po całej sieci. Głęboka sieć resztowa może być postrzegana jako stos **jednostek rezydualnych** (ang. *residual units* — RU), czyli niewielkich sieci neuronowych zawierających połączenie pomijające.

Przejdzmy do omówienia architektury sieci ResNet (rysunek 14.17). Jest ona zdumiewająco prosta. Jej początek i koniec są takie same, jak w sieci GoogLeNet (brakuje jedynie warstwy porzucania), środek zaś wypełnia bardzo głęboki stos prostych jednostek rezydualnych. Każda taka jednostka składa się



Rysunek 14.16. Standardowa głęboka sieć neuronowa (po lewej) i głęboka sieć rezydualna (po prawej)

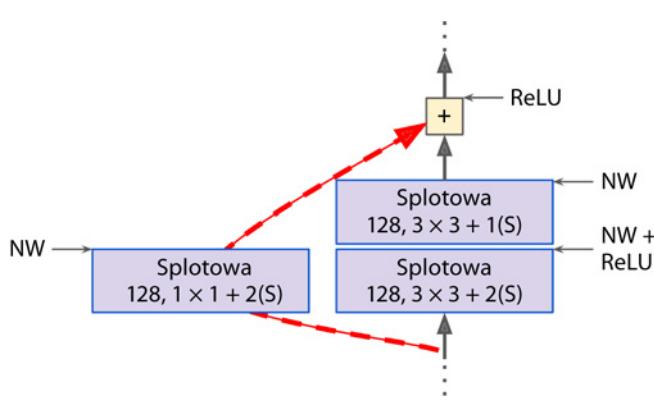
z dwóch warstw splotowych (bez warstwy łączącej!), zawierających normalizację wsadową (NW) i funkcję aktywacji ReLU; do tego ma zdefiniowane jądra o rozmiarze 3×3 oraz zachowuje pierwotne wymiary przestrzenne obrazu (krok o wartości 1, uzupełnianie zerami typu "same").



Rysunek 14.17. Architektura sieci ResNet

Zwróć uwagę, że liczba map cech jest podwajana co kilka jednostek rezydualnych, a jednocześnie ich wysokość i szerokość stają się dwukrotnie mniejsze (poprzez warstwę splotową o kroku 2). Gdy to się dzieje, dane wejściowe nie mogą być dodawane bezpośrednio do wyjść jednostki rezydualnej,

gdyż różnią się one wymiarami (problem ten dotyczy połączenia pomijającego, ukazanego na rysunku 14.17 za pomocą przerywanej strzałki). Rozwiążaniem okazuje się przepuszczanie danych wejściowych przez warstwę splotową 1×1 o kroku 2 i właściwej liczbie wyjściowych map cech (rysunek 14.18).



Rysunek 14.18. Połączenie pomijające w przypadku zmiany rozmiarów mapy cech i liczby kanałów

ResNet-34 to odmiana sieci ResNet zawierająca 34 warstwy (licząc jedynie warstwy splotowe i jedną w pełni połączoną warstwę)¹⁷, w której znajdziemy trzy jednostki rezydualne generujące 64 mapy cech, 4 RU dające 128 map cech, 6 RU tworzących 256 map cech i 3 RU z 512 mapami cech. Zaimplementujemy ten model w dalszej części rozdziału.

W głębszych sieciach resztowych (np. ResNet-152) wykorzystywane są nieco inaczej zbudowane jednostki rezydualne. Zamiast dwóch warstw splotowych 3×3 generujących (na przykład) 256 map cech zawierają one trzy warstwy konwolucyjne: pierwsza ma rozmiar 1×1 i wytwarza zaledwie 64 mapy cech (czterokrotnie mniej) i pełni funkcję warstwy ograniczającej, następna warstwa splotowa ma rozmiar 3×3 i również zawiera 64 mapy cech, natomiast wymiary ostatniej warstwy to również 1×1 i uzyskujemy dzięki niej 256 map cech (4×64), co pozwala przywrócić pierwotną głębokość. Architektura ResNet-152 zawiera trzy takie RU generujące 256 map, po nich 8 RU dających 512 map cech, następnie kolosalną liczbę 36 RU przechowujących 1024 mapy cech, a na końcu 3 RU z 2048 mapami cech.



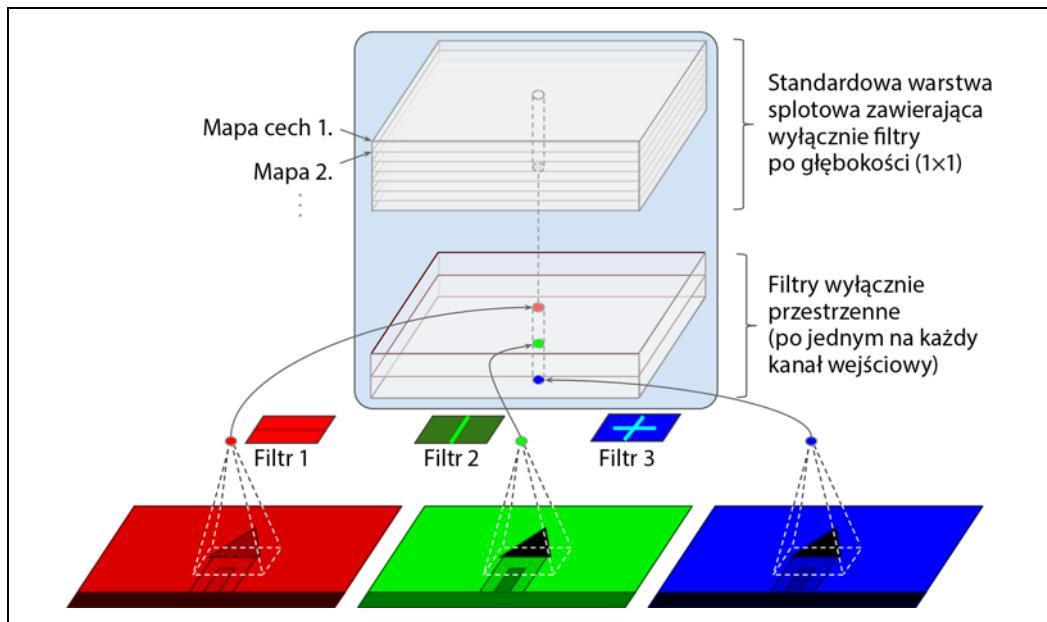
Architektura Inception-v4 (<https://arxiv.org/abs/1602.07261>)¹⁸ z firmy Google stanowi połączenie koncepcji sieci GoogLeNet i ResNet, co pozwoliło uzyskać wynik top-5 rzędu 3% dla klasyfikacji ImageNet.

¹⁷ Podczas opisywania sieci neuronowej powszechną praktyką jest zliczanie jedynie warstw z parametrami.

¹⁸ Christian Szegedy i in., *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning*, arXiv preprint arXiv:1602.07261 (2016).

Xception

Warto wspomnieć o odmianie architektury GoogLeNet: Xception (skrót od ang. *Extreme Inception* — incepja ekstremalna)¹⁹, którą zaprojektował w 2016 roku François Chollet (twórca interfejsu Keras). Uzyskała ona znacznie lepsze wyniki od modelu Inception-v3 w bardzo rozbudowanym zadaniu widzenia maszynowego (350 milionów obrazów i 17 tysięcy klas). Podobnie jak w przypadku modelu Inception-v4, zostały tu wykorzystane koncepcje wywodzące się z sieci GoogLeNet i ResNet, ale moduły incepcyjne zastąpiono specyficzny typem warstwy, mianowicie **rozdzielnią po głębokości warstwą splotową** (ang. *depthwise separable convolution layer*), w skrócie zwaną **rozdzielną warstwą splotową** (ang. *separable convolution layer*²⁰). Warstwy te były wcześniej wykorzystywane w niektórych strukturach CNN, ale nie stanowiły kluczowego elementu sieci. Klasyczna sieć splotowa wykorzystuje filtry starające się wykrywać jednocześnie wzorce przestrzenne (np. kształt ovalny) i międzykanalowe (np. usta + nos + oczy = twarz), natomiast w rozdzielnej warstwie splotowej zakładamy, że wzorce przestrzenne i międzykanalowe mogą być modelowane oddzielnie (rysunek 14.19). Zatem składa się ona z dwóch części: w pierwszej stosowany jest pojedynczy filtr przestrzenny dla każdej wejściowej mapy cech, a następnie druga część wyszukuje wyłącznie wzorce międzykanalowe (jest to tradycyjna warstwa splotowa zawierająca filtry 1×1).



Rysunek 14.19. Rozdzielna warstwa splotowa po głębokości

¹⁹ François Chollet, *Xception: Deep Learning with Depthwise Separable Convolutions*, arXiv preprint arXiv: 1610.02357 (2016), <https://arxiv.org/abs/1610.02357>.

²⁰ Nazwa ta czasami bywa niejednoznaczna, ponieważ rozdzielne przestrzennie operacje splotu bywają często nazywane rozdzielonymi operacjami splotu.

Rozdzielne warstwy splotowe zawierają wyłącznie po jednym filtrze przestrzennym na każdy kanał wejściowy, dlatego nie należy umieszczać ich po warstwach mających zbyt mało kanałów, jak na przykład po warstwie wejściowej (taką sytuację widzimy na rysunku 14.19, ale została ona zaprezentowana jedynie w celach poglądowych). Z tego powodu na początku struktury Xception mieścią się dwie standardowe warstwy splotowe, po których realizowane są wyłącznie rozdzielne operacje splotu (łącznie 34); występuje tu także kilka warstw łączących i klasyczne warstwy końcowe (globalna uśredniająca warstwa łącząca wraz z gęstą warstwą wyjściową).

Być może się zastanawiasz, dlaczego model Xception uznawany jest za wariant sieci GoogLeNet, skoro nawet nie zawiera modułów incepcyjnych. Jak już wiesz, moduł incepcyjny zawiera warstwy splotowe z filtrami 1×1 — wyszukują one wyłącznie wzorce międzykanałowe. Jednak znajdujące się ponad nimi klasyczne warstwy konwolucyjne wyszukują zarówno wzorce przestrzenne, jak i międzykanałowe. Możesz więc traktować moduł incepcyjny jako warstwę pośrednią między standardową warstwą splotową (rozpatrującą łącznie wzorce przestrzenne i międzykanałowe) a rozdzielnią warstwą splotową (traktującą te wzorce oddziennie). W praktyce okazuje się, że rozdzielne warstwy splotowe uzyskują nieco lepszą wydajność.



Rozdzielne warstwy splotowe wykorzystują mniej parametrów, pamięci i obliczeń w porównaniu do tradycyjnych warstw konwolucyjnych. Zasadniczo cechują się również lepszą skutecznością, dlatego warto traktować je jako warstwy domyślne (nie umieszczaj ich tylko za warstwami zawierającymi mało kanałów).

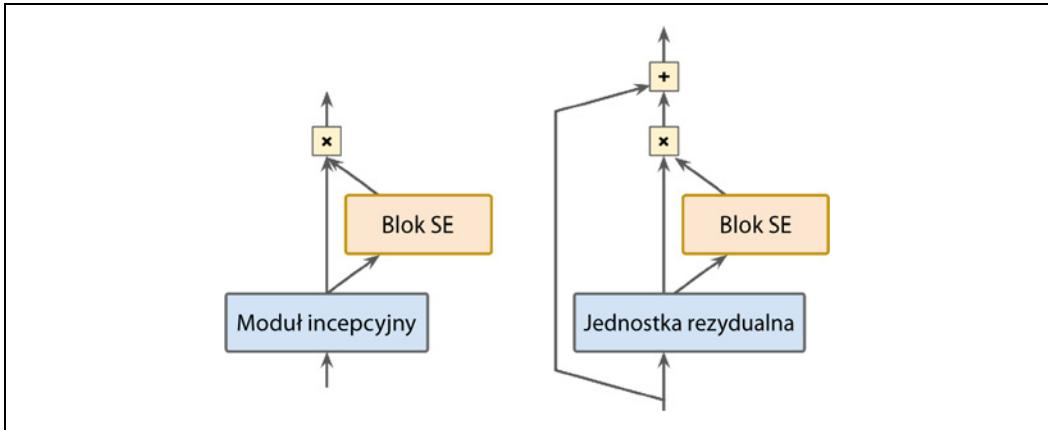
Wyzwanie ILSVRC 2016 wygrał zespół CUIImage z Uniwersytetu Chińskiego w Hongkongu. Zwycięzcy wykorzystali kombinację wielu różnych technik, w tym zaawansowany system wykrywania obiektów o nazwie GBD-Net (<https://arxiv.org/abs/1610.02579>)²¹, co pozwoliło uzyskać współczynnik błędu top-5 poniżej 3%. Wynik ten jest niewątpliwie imponujący, ale złożoność tej architektury wyraźnie kontrastuje z prostotą sieci ResNet. Ponadto przekonamy się za chwilę, że już rok później pojawiła się kolejna w miarę prosta architektura, która uzyskała jeszcze lepszy rezultat.

SENet

Zwycięska struktura w konkursie ILSVRC 2017 to sieć Squeeze-and-Excitation (SENet; <https://arxiv.org/abs/1709.01507>)²². Architektura ta stanowi rozwinięcie dotychczasowych sieci, takich jak sieci incepcyjne czy sieci ResNet, i zwiększa ich wydajność. W ten sposób sieć SENet uzyskała zdumiewający wynik top-5 rzędu 2,25%! Rozszerzone wersje sieci incepcyjnych i ResNet noszą nazwy, odpowiednio, **SE-Inception** i **SE-ResNet**. Dodatkowa skuteczność wynika z faktu, że zostaje dołączona niewielka sieć neuronowa, tzw. **blok SE**, do każdej jednostki w architekturze pierwotnej (tzn. do każdego modułu incepcyjnego lub każdej jednostki rezydualnej) (rysunek 14.20).

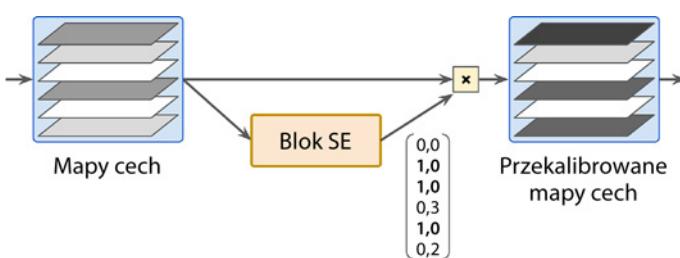
²¹ Xingyu Zeng i in., *Crafting GBD-Net for Object Detection*, „IEEE Transactions on Pattern Analysis and Machine Intelligence” 40, no. 9 (2018), s. 2109 – 2123.

²² Jie Hu i in., *Squeeze-and-Excitation Networks*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2018), s. 7132 – 7141.



Rysunek 14.20. Moduły SE-Inception (po lewej) i SE-ResNet (po prawej)

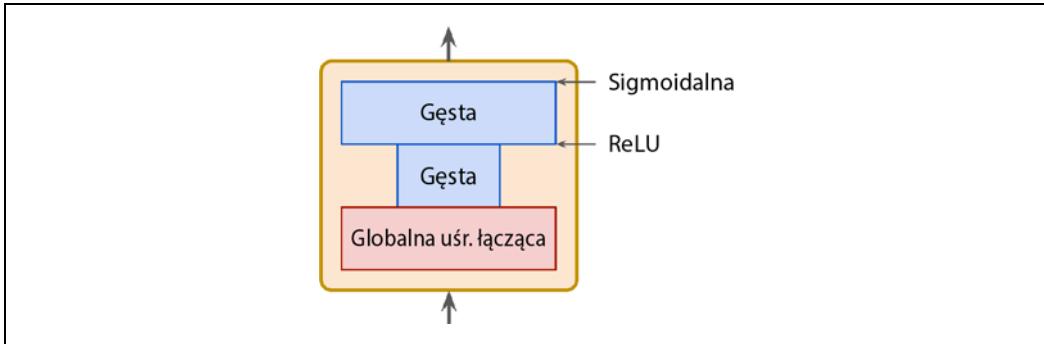
Blok SE analizuje wynik przylegającej jednostki i koncentruje się wyłącznie na wymiarze głębości (nie zwraca uwagi na wzorce przestrzenne). Sprawdza w ten sposób, które cechy są ze sobą zazwyczaj powiązane pod względem aktywności. Informacja ta zostaje wykorzystana do przekalibrowania map cech tak, jak zaprezentowano na rysunku 14.21. Na przykład blok SE może „odkryć”, że usta, nosy i oczy występują zasadniczo na tych samych obrazach: jeśli zobaczysz usta i nos, możesz spodziewać się również oczu. Zatem jeśli blok SE wykryje silne pobudzenie w mapach cech ust i nosa, ale jedynie nieznaczną aktywację w mapie cech oczu, to zostanie ona wzmacniona (mówiąc precyzyjniej, zostaną zmniejszone wagie nieistotnych map cech). Jeżeli oczy były mylone z jakimś innym obiektem, taka rekalibracja map cech pozwoli ujednoznaczyć rezultat.



Rysunek 14.21. Blok SE przeprowadza ponowną kalibrację map cech

Blok SE składa się zaledwie trzech warstw: globalnej uśredniającej warstwy łączącej, gęstej warstwy ukrytej wykorzystującej funkcję ReLU oraz gęstej warstwy wyjściowej z wyznaczoną sigmoidalną funkcją aktywacji (rysunek 14.22).

Podobnie jak wcześniej, globalna uśredniająca warstwa łącząca oblicza średnią wartość pobudzenia dla każdej mapy cech: na przykład jeżeli na jej wejściu występuje 256 map cech, to na wyjście zostanie przekazanych 256 wartości reprezentujących całkowitą odpowiedź dla każdego filtra. W następnej warstwie następuje „ścisnięcie” (ang. *squeeze*): znajduje się w niej znacznie mniej niż 256 neuronów, zazwyczaj szesnastokrotnie mniej, niż wynosi liczba map cech (czyli w tym wypadku



Rysunek 14.22. Struktura bloku SE

16 neuronów), więc 256 wartości liczbowych zostaje upakowanych w małym wektorze (tutaj szesnastowymiarowym). Jest to małowymiarowa reprezentacja wektora (tzn. wektor właściwościowy) rozkładu odpowiedzi na cechy. Taka faza ograniczająca zmusza blok SE do poznawania ogólnej reprezentacji kombinacji cech (zasada ta zostanie ponownie wykorzystana w rozdziale 17., podczas omawiania autokoderów). Na koniec warstwa wyjściowa przyjmuje taki wektor właściwościowy i daje na wyjściu wektor rekalibracji zawierający po jednej wartości liczbowej na każdą mapę cech (czyli tutaj ma rozmiar 256) w zakresie od 0 do 1. Mapy cech zostają następnie pomnożone przez ten wektor cech, co oznacza, że mało istotne cechy (mające małą wartość rekalibracji) zostają zredukowane, natomiast cechy istotne (o współczynniku rekalibracji zbliżonym do 1) są pozostawione bez zmian.

Implementacja sieci ResNet-34 za pomocą interfejsu Keras

Implementacja większości dotychczas opisanych sieci splotowych nie powinna sprawiać większego problemu (choć, jak się wkrótce przekonasz, lepiej wczytywać już wytrenowane sieci). Zilustrujemy to na przykładzie sieci ResNet-34, którą zaimplementujemy od podstaw w interfejsie Keras. Utwórzmy najpierw klasę ResidualUnit:

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                keras.layers.Conv2D(filters, 1, strides=strides,
                                   padding="same", use_bias=False),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        Z = inputs
```

```

for layer in self.main_layers:
    Z = layer(Z)
skip_Z = inputs
for layer in self.skip_layers:
    skip_Z = layer(skip_Z)
return self.activation(Z + skip_Z)

```

Jak widać, listing ten odpowiada dość ściśle strukturze pokazanej na rysunku 14.18. W konstruktorze tworzymy wszystkie potrzebne warstwy. Główne warstwy (`main.layers`) widoczne są po prawej stronie diagramu, natomiast warstwy pomijające (`skip.layers`) znajdują się po lewej (wymagane są one tylko wtedy, gdy krok jest większy od 1). Następnie w metodzie `call()` sprawiamy, że dane wejściowe przechodzą przez warstwy główne i pomijające (jeżeli występują), a potem dodajemy wyniki obydwu warstw i przetwarzamy je za pomocą funkcji aktywacji.

Następnie możemy utworzyć strukturę ResNet-34 za pomocą interfejsu `Sequential`, ponieważ zasadniczo mamy tu do czynienia jedynie z długą sekwencją warstw (po zdefiniowaniu klasy `ResidualUnit` możemy traktować każdą jednostkę rezydualną jako pojedynczą warstwę):

```

model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2, input_shape=[224, 224, 3],
                            padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="same"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))

```

Jednym nieco bardziej skomplikowanym elementem w tym listingu jest pętla dodająca warstwy `ResidualUnit` do modelu: jak już wyjaśniłem, pierwsze trzy RU zawierają 64 filtry, kolejne cztery mają 128 filtrów itd. Następnie wyznaczamy krok równy 1, gdy liczba filtrów jest taka sama jak w poprzedniej jednostce rezydualnej, w przeciwnym razie krok będzie równy 2. Potem dodajemy `ResidualUnit`, a na końcu aktualizujemy `prev_filters`.

To niesamowite, że za pomocą kodu mieszczącego się w nielicznych 40 wierszach jesteśmy w stanie zbudować model, który wygrał główną nagrodę w zawodach ILSVRC 2015! Świadczy to zarówno o elegancji sieci ResNet, jak i o ekspresyjności interfejsu Keras. Implementowanie innych struktur sieci splotowych nie jest o wiele trudniejsze. Keras zawiera jednak kilka wbudowanych architektur sieci CNN, dlaczego więc z nich nie skorzystać?

Korzystanie z gotowych modeli w interfejsie Keras

Zasadniczo nie istnieje potrzeba własnoręcznego implementowania standardowych modeli, takich jak GoogLeNet czy ResNet, ponieważ gotowe sieci są dostępne za pomocą pojedynczego wiersza kodu w pakiecie `keras.applications`. Możesz na przykład wczytać model ResNet-50, wytrenowany na zestawie danych ImageNet, za pomocą następującego listingu:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

To wszystko! Zostanie utworzony model ResNet-50, a także będą pobrane wagi wyuczone na zestawie danych ImageNet. Aby skorzystać z tego modelu, musisz najpierw zapewnić obrazy wejściowe o odpowiednim rozmiarze. Model ResNet-50 oczekuje obrazów o wymiarach 224×224 (inne modele mogą odczytywać obrazy o innych wymiarach, np. 299×299), dlatego zastosujemy funkcję `tf.image.resize()` do zmiany rozmiarów wczytanych uprzednio obrazów:

```
images_resized = tf.image.resize(images, [224, 224])
```



Funkcja `tf.image.resize()` nie zachowuje proporcji obrazu. Jeżeli stanowi to problem, spróbuj najpierw przyciąć obrazy do odpowiedniej proporcji, zanim przystąpisz do zmiany ich rozmiarów. Możesz zrealizować obydwie operacje za jednym zamachem przy użyciu funkcji `tf.image.crop_and_resize()`.

W gotowych modelach istnieje wymóg przygotowania obrazów w określony sposób. W pewnych przypadkach może zaistnieć potrzeba przeskalowania danych wejściowych do zakresu od 0 do 1, od -1 do 1 lub jeszcze inaczej. Każdy model zawiera funkcję `preprocess_input()`, dzięki której możesz wstępnie przetworzyć dane wejściowe. Funkcje te zakładają, że intensywność pikseli mieści się w zakresie od 0 do 255, zatem musimy pomnożyć je przez 255 (ponieważ wcześniej przeskalowaliśmy je do zakresu od 0 do 1):

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Możemy teraz za pomocą takiego gotowego modelu uzyskiwać prognozy:

```
Y_proba = model.predict(inputs)
```

Jak zwykle wynik `Y_proba` jest macierzą, w której każdy wiersz jest poświęcony jednemu obrazowi, a każda kolumna klasie (w tym przypadku występuje 1000 klas). Jeżeli chcesz wyświetlać K najlepszych prognoz wraz z nazwą klasy i szacowanym prawdopodobieństwem przynależności do każdej klasy, użyj funkcji `decode_predictions()`. Dla każdego obrazu zostanie zwrócona tablica zawierająca K najlepszych predykcji, gdzie każda prognoza jest przedstawiana w postaci tabeli przechowującej identyfikator klasy²³, jej nazwę i związany z nią wynik pewności:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Obraz numer {}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print(" {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

Wynik wygląda następująco:

```
Obraz numer 0
n03877845 - palace          42.87%
n02825657 - bell_cote        40.57%
n03781244 - monastery        14.56%
```

```
Obraz numer 1
n04522168 - vase             46.83%
n07930864 - cup               7.78%
n11939491 - daisy            4.87%
```

²³ W zestawie ImageNet każdy obraz jest skojarzony ze słowem tworzącym zestaw danych WordNet (<https://wordnet.princeton.edu/>): identyfikator klasy to po prostu identyfikator WordNet.

Właściwe klasy (*monastery* i *daisy*) znajdują się wśród trzech najlepszych wyników dla obydwu obrazów. Całkiem nieźle, jeśli uwzględnimy fakt, że model musiał wybierać spośród tysiąca klas.

Jak widać, stworzenie bardzo dobrego klasyfikatora obrazów z gotowego modelu jest niezmiernie proste. W pakiecie keras.applications dostępne są również inne modele widzenia maszynowego, w tym kilka odmian sieci ResNet, warianty sieci GoogLeNet, takie jak Inception-v3 i Xception, warianty sieci VGGNet czy struktury MobileNet i MobileNetV2 („lekkie” architektury przeznaczone do aplikacji mobilnych).

A gdybyśmy chcieli wykorzystać klasyfikator obrazów do klas niewystępujących w zestawie ImageNet? W takim przypadku nadal możesz korzystać z zalet gotowej sieci i przeprowadzać uczenie transferowe.

Gotowe modele w uczeniu transferowym

Jeżeli chcesz stworzyć klasyfikator obrazów, ale brakuje Ci danych uczących, to nieraz warto wykorzystać dolne warstwy wcześniej wytrenowanego modelu, o czym już wiesz z rozdziału 11. Na przykład wyuczmy model klasyfikujący zdjęcia kwiatów za pomocą gotowej sieci Xception. Najpierw wczytajmy zestaw danych za pomocą biblioteki TensorFlow Datasets (zob. rozdział 13.):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

Möżesz uzyskać informacje o zestawie danych poprzez wyznaczenie parametru `with_info=True`. W tym przypadku poznajemy rozmiar zestawu danych i nazwy klas. Niestety mamy do dyspozycji jedynie zbiór "train" (bez zbiorów walidacyjnego czy testowego), dlatego musimy go podzielić. Odpowiedni do tego celu interfejs API stanowi część projektu TF Datasets. Przeznaczmy na przykład początkowe 10% zestawu danych na testowanie, następne 15% na walidację, a pozostałe 75% na uczenie:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Musimy teraz wstępnie przetworzyć te obrazy. Nasza sieć splotowa oczekuje danych wejściowych o wymiarach 224×224 , dlatego musimy zmienić ich rozmiar. Musimy także przepuścić ją przez funkcję `preprocess_input()` sieci Xception:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Zastosujmy tę funkcję wstępnego przetwarzania na wszystkich zbiorach danych, przetasujmy zbiór uczący, a także dodajmy dzielenie na grupy oraz wstępne ładowanie wszystkich zbiorów:

```
batch_size = 32
train_set = train_set.shuffle(1000)
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

Jeżeli chcesz dogenerować dane, zmień funkcję przetwarzania wstępnego dla zbioru uczącego i dodaj jakieś losowe przekształcenia obrazów. Na przykład użyj funkcji `tf.image.random_crop()`, aby losowo przycinać obrazy, funkcja `tf.image.random_flip_left_right()` losowo przerzuca je w poziomie itd. (zajrzyj do sekcji „Gotowe modele w uczeniu transferowym” w notatniku Jupyter).



Klasa `keras.preprocessing.image.ImageDataGenerator` ułatwia wczytywanie obrazów z dysku i dogenerowanie z nich danych na różne sposoby: możesz przesuwać każdy obraz, obracać go, przeskakowywać, przerzucać w poziomie lub w pionie, przycinać lub wprowadzać dowolną funkcję transformującą. Jest to bardzo wygodne rozwiązanie w prostych projektach, jednak budowanie potoku `tf.data` ma wiele zalet: dzięki niemu możesz wydajnie (np. równolegle) wczytywać obrazy z dowolnego źródła, a nie tylko z dysku, możesz manipulować obiektem `Dataset` w dowolny sposób, a także możesz napisać funkcję przetwarzania wstępnego bazującą na operacjach `tf.image`, która może być użyta zarówno w potoku `tf.data`, jak i w modelu wdrażanym do środowiska produkcyjnego (zob. rozdział 19.).

Wczytajmy następnie model Xception, wytrenowany na zestawie danych ImageNet. Wykluczamy górną część sieci poprzez wyznaczenie parametru `include_top=False` — pomijamy w ten sposób globalną uśredniającą warstwę łączącą i gęstą warstwę wyjściową. Dodajemy teraz własną globalną uśredniającą warstwę łączącą wraz z gęstą warstwą wyjściową, zawierającą po jednej jednostce na każdą klasę, a także funkcję softmax. W końcu tworzymy obiekt Model:

```
base_model = keras.applications.Xception(weights="imagenet",
                                             include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

W rozdziale 11. wyjaśniłem, że zazwyczaj dobrym rozwiązaniem jest zamrażanie wag wytrenowanych warstw, przynajmniej na początku procesu uczenia:

```
for layer in base_model.layers:
    layer.trainable = False
```



Nasz model wykorzystuje bezpośrednio warstwy modelu bazowego, a nie sam obiekt `base_model`, dlatego wyznaczenie parametru `base_model.trainable=False` okazałoby się bezskuteczne.

Możemy w końcu skompilować model i rozpoczęć jego uczenie:

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
               metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```



Jeżeli nie korzystasz z procesora graficznego, proces ten może trwać bardzo długo. W takiej sytuacji uruchom notatnik Jupyter w środowisku Colab (jest darmowe!). Instrukcję uruchomienia trybu GPU znajdziesz w materiałach dodatkowych (<ftp://ftp.helion.pl/przykłady/uczem2.zip>).

Po wytrenowaniu modelu przez kilka epok jego dokładność dla zbioru walidacyjnego powinna osiągnąć wartość ok. 75 – 80% i zatrzymać się na tym poziomie. Oznacza to, że warstwy górne zostały całkiem dobrze wytrenowane, zatem jesteśmy gotowi odmrozić wszystkie warstwy (ewentualnie możesz sprawdzić, co się stanie po odmrożeniu tylko kilku najwyższych zamrożonych warstw) i kontynuować uczenie (nie zapomnij o kompilowaniu modelu po każdorazowym zamrażaniu lub rozmrzaniu warstw). Tym razem wprowadzimy znacznie mniejszy współczynnik uczenia, aby nie uszkodzić uprzednio wytrenowanych wag:

```
for layer in base_model.layers:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)  
model.compile(...)  
history = model.fit(...)
```

Proces ten zajmie trochę czasu, ale model powinien osiągnąć dokładność rzędu 95% dla zbioru testowego. W ten sposób możesz zacząć trenować zdumiewające klasyfikatory obrazów! Widzenie maszynowe nie kończy się jednak wyłącznie na zadaniach klasyfikacyjnych. A gdybyśmy chcieli wiedzieć na przykład, *gdzie* znajduje się kwiat na zdjęciu? Zobaczmy, co możemy osiągnąć w tym aspekcie.

Klasyfikowanie i lokalizowanie

Lokalizowanie obiektu na zdjęciu może być wyrażone w postaci zadania regresji, o czym wiesz już z rozdziału 10.: aby przewidzieć ramkę obejmującą obiekt, powszechnym rozwiązaniem jest prognozowanie współrzędnych środka obiektu, a także jego wysokości i szerokości. Oznacza to, że musimy przewidzieć cztery wartości. Nie musimy drastycznie modyfikować modelu — wystarczy dodać drugą gęstą warstwę wyjściową, zawierającą cztery jednostki (umieszczać ją zazwyczaj nad globalną uśredniającą warstwą łączącą), a następnie wyuczyć go za pomocą funkcji MSE:

```
base_model = keras.applications.Xception(weights="imagenet",  
                                             include_top=False)  
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)  
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)  
loc_output = keras.layers.Dense(4)(avg)  
model = keras.Model(inputs=base_model.input,  
                     outputs=[class_output, loc_output])  
model.compile(loss=[ "sparse_categorical_crossentropy", "mse"],  
              loss_weights=[0.8, 0.2], # Zależy, co chcesz uzyskać  
              optimizer=optimizer, metrics=[ "accuracy" ])
```

Natrafiamy tu jednak na problem: nasz zestaw danych nie zawiera ramek otaczających kwiaty. Musimy więc samodzielnie je dodać. Często jest to jedna z najtrudniejszych i najdroższych części uczenia maszynowego: oznaczanie danych. Warto poświęcić czas na znalezienie odpowiednich narzędzi. Aby umieszczać ramki ograniczające na obrazach, możesz skorzystać z narzędzi o otwartym

kodzie źródłowym, takich jak VGG Image Annotator, LabelImg, OpenLabeler czy ImgLab, albo z aplikacji komercyjnej, na przykład LabelBox lub Supervisely. Jeżeli dysponujesz olbrzymią liczbą obrazów, możesz także wziąć pod uwagę platformy oparte na źródłach społecznościowych, na przykład Amazon Mechanical Turk. Jednakże przygotowanie takiej platformy i formularza dla pracowników, nadzorowanie ich i sprawdzanie wyznaczanych przez nich ramek jest bardzo pracochnonne, dlatego zastanów się najpierw, czy warto korzystać z tego rozwiązania. Jeżeli trzeba oznakować tylko kilka tysięcy obrazów i nie zamierzasz tego robić często, być może lepiej zająć się tym samemu. Adriana Kovashka i in. napisali bardzo praktyczny artykuł (<https://arxiv.org/abs/1611.02145>)²⁴, poświęcony źródłom społecznościowym w widzeniu maszynowym. Zalecam zapoznanie się z jego treścią, nawet jeśli nie zamierzasz korzystać ze źródeł społecznościowych.

Powiedzmy, że wprowadziliśmy ramki ograniczające w każdym obrazie z naszego zestawu danych (założymy na razie, że każdy obraz zawiera tylko jedną ramkę). Musimy teraz utworzyć zestaw danych, którego elementami będą grupy wstępnie przetworzonych obrazów wraz z etykietami ich klas i ramkami. Każdy element powinien być krotką przyjmującą postać (*obrazy*, (*etykiety_klas*, *ramki_ograniczające*)). Teraz możemy wytrenować model!



Ramki ograniczające powinny być znormalizowane w taki sposób, aby współrzędne środka oraz wysokość i szerokość mieściły się w zakresie od 0 do 1. Często również przewidywane są nie wysokość i szerokość, lecz ich kwadraty: w ten sposób dziesięciopięciowy błąd w dużej ramce nie będzie karany tak bardzo jak w przypadku małej ramki.

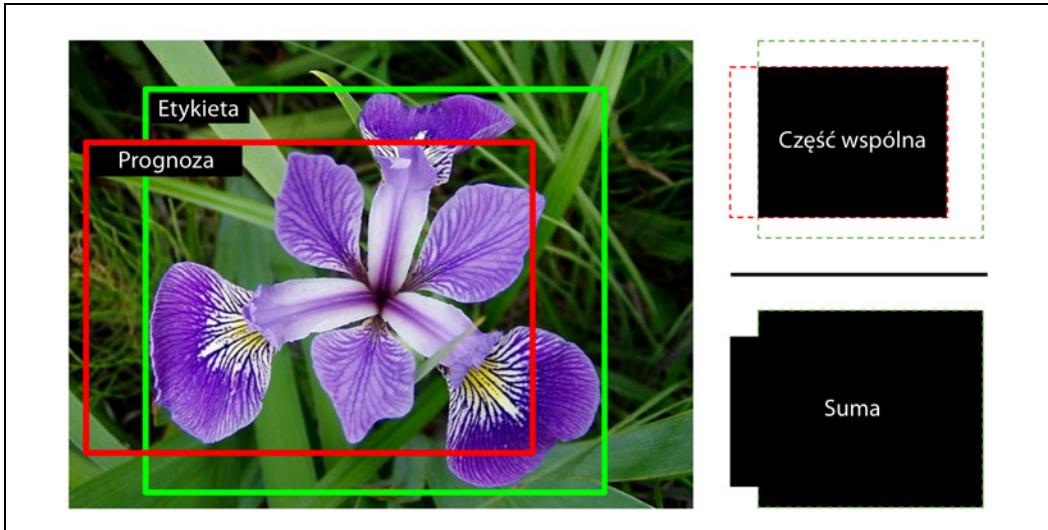
Błąd średniokwadratowy często sprawuje się całkiem dobrze jako funkcja kosztu ucząca model, ale nie stanowi on dobrego wskaźnika oceniającego skuteczność prognozowania ramek ograniczających. Najpopularniejszym wskaźnikiem tego typu jest indeks Jaccarda, znany w terminologii angielskiej jako *Intersection over Union* (IoU): jest to obszar utworzony przez część wspólną pomiędzy ramką prognozowaną i ramką docelową podzielony przez obszar sumy tych ramek (rysunek 14.23). W interfejsie tf.keras rozwiążanie to jest implementowane przez klasę tf.keras.metrics.MeanIoU.

Klasyfikowanie i lokalizowanie pojedynczego obiektu to całkiem miła funkcja, ale co w przypadku, gdy obrazy zawierają wiele obiektów (jest to często spotykana sytuacja w zestawie danych Flowers)?

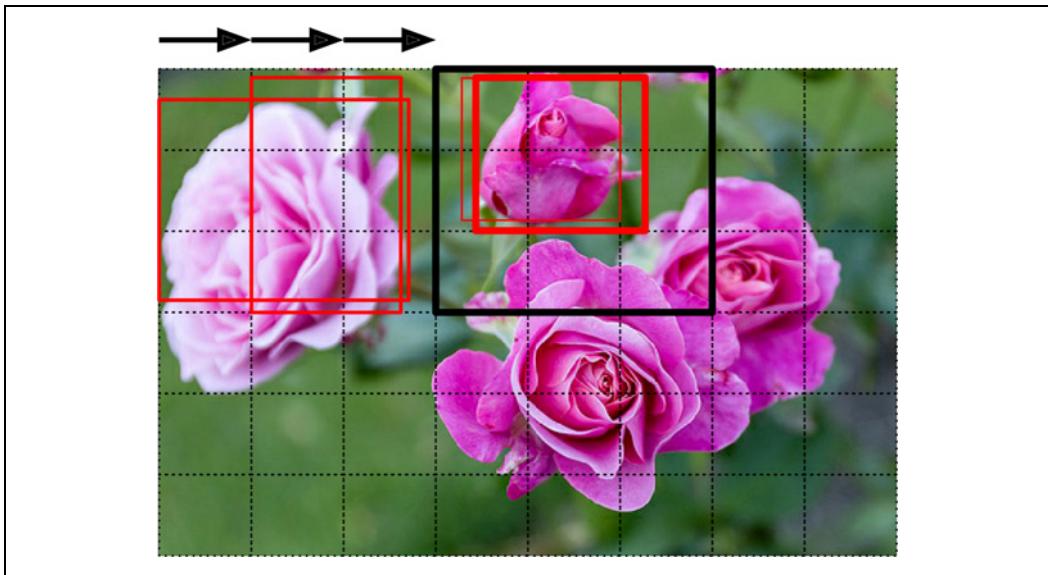
Wykrywanie obiektów

Zadanie klasyfikowania i lokalizowania wielu obiektów na zdjęciach nosi nazwę wykrywania (detekcji) obiektów. Jeszcze kilka lat temu najczęściej wykorzystywano w tym celu sieć splotową wytrenowaną w klasyfikowaniu i lokalizowaniu pojedynczych elementów, która była przesuwana wzduż obrazu (rysunek 14.24). W omawianym przykładzie obraz został podzielony na siatkę o wymiarach 6×8 , natomiast sieć CNN (gruby, czarny czworokąt) ma rozmiar 3×3 . Gdy sieć zaczynała od lewego górnego rogu, wykrywała część róży znajdującej się skrajnie po lewej stronie; tę samą

²⁴ Adriana Kovashka i in., *Crowdsourcing in Computer Vision*, „Foundations and Trends in Computer Graphics and Vision” 10, no. 3 (2014), s. 177 – 243.



Rysunek 14.23. Indeks Jaccarda jako wskaźnik ramek ograniczających



Rysunek 14.24. Wykrywanie wielu obiektów poprzez przesuwanie sieci splotowej po obrazie

różę odkrywała w momencie pierwszego przesunięcia w prawo. W kolejnym kroku została wykryty fragment górnej róży, podobnie jak po kolejnym przesunięciu. W ten sposób sieć splotowa analizowała cały obraz i za każdym razem koncentrowała się na regionach 3×3 . Ponadto obiekty mają różne rozmiary, dlatego sieć CNN powinna przesuwać się po rejonach o różnych wymiarach. Na przykład po przeanalizowaniu wszystkich rejonów 3×3 sieć splotowa powinna przeszukać również wszystkie obszary 4×4 .

Technika ta jest nieskomplikowana, ale, jak widać, jeden obiekt jest w niej wykrywany wielokrotnie w nieco innych położeniach. Konieczne jest więc przetwarzanie końcowe, służące do usuwania nadmiarowych ramek ograniczających. Powszechnie stosowanym rozwiązaniem jest **usuwanie niemaksymalnych pikseli** (ang. *non-max suppression*). Działa ono w następujący sposób:

1. Musimy najpierw dodać na wyjściu sieci CNN „obiektywość” (ang. *objectness*), która pozwala oszacować, czy na zdjęciu rzeczywiście znajduje się kwiat (ewentualnie można dodać klasę „niekwiat”, ale to rozwiązanie nie sprawdza się najlepiej). Jednostka ta musi wykorzystywać sigmoidalną funkcję aktywacji i możemy wyuczyć ją za pomocą entropii krzyżowej. Następnie należy się pozbyć wszystkich ramek ograniczających, dla których wynik „obiektywości” jest mniejszy od założonej wartości progowej — w ten sposób zostaną usunięte wszystkie ramki, które w istocie nie obejmują kwiatu.
2. Wyszukujemy ramkę ograniczającą o największym wyniku „obiektywości” i pozbywamy się wszystkich innych ramek, które nakładają się na nią w dużym stopniu (np. mających indeks Jaccarda o wartości powyżej 60%). Przykładowo na rysunku 14.24 ramka ograniczająca o maksymalnym wyniku „obiektywości” została oznaczona jako pogrubiony, czerwony kwadrat otaczający górną różę (wynik „obiektywości” jest symbolizowany grubością krawędzi ramek). Druga ramka obejmująca tę różę nakłada się w znacznym stopniu na ramkę o maksymalnym wyniku „obiektywości”, dlatego ją usuniemy.
3. Powtarzamy krok 2. aż do usunięcia wszystkich ramek ograniczających, których można się pozbyć.

Ten prosty mechanizm wykrywania obiektów jest skuteczny, ale wymaga wielu przebiegów sieci splotowej, więc nie jest zbyt szybki. Na szczęście istnieje sposób, by przyspieszyć analizowanie obrazu przez sieć CNN: jest to **w pełni połączona sieć splotowa** (ang. *Fully Convolutional Network* — FCN).

W pełni połączone sieci splotowe

Koncepcja sieci FCN została zaprezentowana przez Jonathana Longa i in. w publikacji z 2015 roku (<https://arxiv.org/abs/1411.4038>)²⁵, poświęconej segmentacji semantycznej (zadaniu klasyfikowania każdego piksela zgodnie z przynależnością obiektu do danej klasy). Autorzy zauważyl, że można zastąpić warstwy gęste znajdujące się na szcycie sieci CNN warstwami splotowymi. Przeanalizujmy to na przykładzie. Założymy, że na szcycie sieci splotowej znajduje się warstwa gęsta składająca się z 200 neuronów i dająca na wyjściu 100 map cech o wymiarach 7×7 (jest to rozmiar mapy cech, a nie jądra). Każdy neuron oblicza sumę ważoną wszystkich $100 \times 7 \times 7$ pobudeń z warstwy konwolucyjnej (plus członu obciążenia). Pomyślmy, co się stanie, jeżeli zastąpimy warstwę gęstą warstwą splotową, która wykorzystuje 200 filtrów o rozmiarze 7×7 i uzupełnianiem zerami typu "valid". Warstwa ta będzie generować 200 map cech o rozmiarze 1×1 (gdyż jądro ma rozmiar taki sam jak wejściowe mapy cech i stosujemy uzupełnianie zerami typu "valid"). Inaczej mówiąc, uzyskamy 200 liczb, podobnie jak w przypadku warstwy gęstej; a jeżeli przyjrzymy się uważniej obliczeniom przeprowadzonym przez warstwę splotową, przekonamy się, że wyniki są takie same jak uzyskane za pomocą warstwy gęstej. Jedyna różnica polega na tym, że wynik

²⁵ Jonathan Long i in., *Fully Convolutional Networks for Semantic Segmentation*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2015), s. 3431 – 3440.

w warstwie gęstej jest tensorem o wymiarach [*rozmiar grupy*, 200], natomiast warstwa splotowa generuje tensor o wymiarach [*rozmiar grupy*, 1, 1, 200].



Aby można było przekształcić warstwę gęstą w splotową, liczba filtrów w warstwie splotowej musi odpowiadać warstwie jednostek warstwy gęstej, rozmiar filtra musi być równy rozmiarom wejściowych map cech i należy korzystać z uzupełniania zerami typu "valid". Za chwilę przekonasz się, że krok może być równy 1 lub większy.

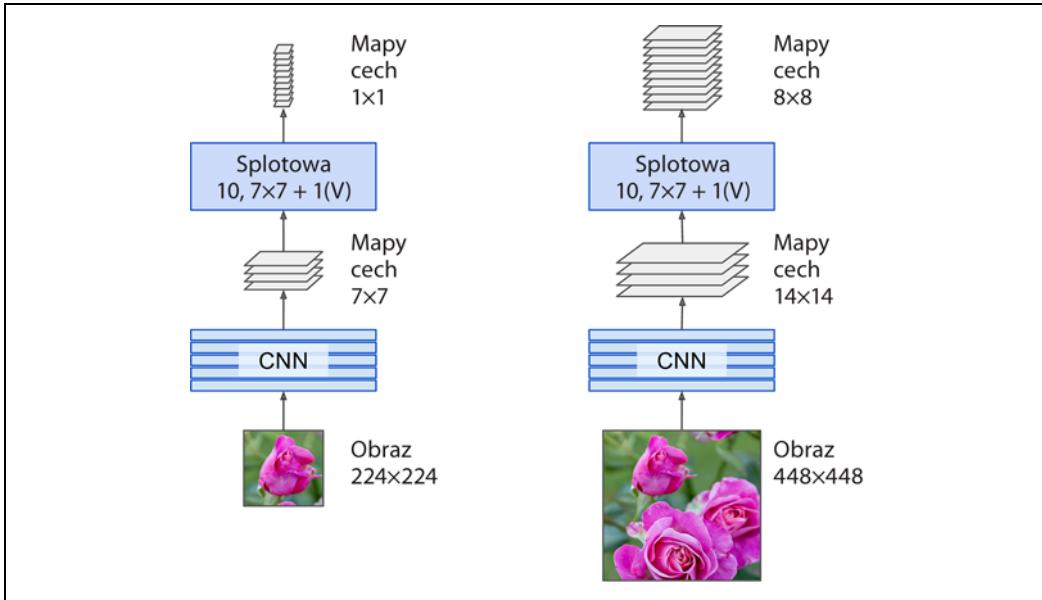
Dlaczego jest to ważne? Warstwa gęsta oczekuje danych wejściowych o określonym rozmiarze (gdyż zawiera po jednej wadze na każdą cechę wejściową), natomiast warstwa splotowa bez problemu może przetwarzanie obrazy o dowolnym rozmiarze²⁶ (oczekuje jednak, że dane wejściowe będą miały określoną liczbę kanałów, ponieważ każde jądro zawiera inny zestaw wag dla poszczególnych kanałów wejściowych). Sieć FCN składa się wyłącznie z warstw splotowych (i łączących, które mają tę samą własność), zatem można ją uczyć i wykorzystywać wobec obrazów o dowolnych rozmiarach!

Założymy na przykład, że wytrenowaliśmy już sieć splotową w klasyfikowaniu i lokalizowaniu kwiatów. Użyliśmy w tym celu obrazów o rozmiarach 224×224 , a na wyjściu otrzymywaliśmy dziesięć liczb: wyniki od 0. do 4. są przesyłane przez funkcję softmax, dzięki czemu uzyskujemy prawdopodobieństwa przynależności do klas (po jednym na klasę); wynik 5. przetwarzany jest przez logistyczną funkcję aktywacji i reprezentuje wynik „obiektości”; rezultaty od 6. do 9. nie wykorzystują żadnej funkcji aktywacji i symbolizują współrzędne środka ramki ograniczającej, a także jej wysokość i szerokość. Możemy teraz przekształcić warstwy gęste modelu w warstwy splotowe. Nie musimy nawet ponownie trenować modelu — wystarczy skopiować wagi z warstw gęstych do warstw konwolucyjnych! Ewentualnie można przekształcić sieć CNN w sieć FCN jeszcze przed uczeniem modelu.

Założymy teraz, że ostatnia warstwa splotowa przed warstwą wyjściową (warstwa ograniczająca) generuje mapy cech o rozmiarach 7×7 w czasie, gdy do sieci dostarczane są obrazy o wymiarach 224×224 (lewa strona rysunku 14.25). Jeżeli wprowadzimy do sieci FCN obrazy o rozmiarach 448×448 (prawa strona rysunku 14.25), warstwa ograniczająca zacznie generować mapy cech o wymiarach 14×14 ²⁷. Wyjściowa warstwa gęsta została zastąpiona warstwą splotową składającą się z dziesięciu filtrów o rozmiarach 7×7 , z uzupełnianiem zerami typu "valid" i krokiem równym 1, zatem na wyjściu otrzymamy 10 map cech o rozmiarach 8×8 (gdyż $14 - 7 + 1 = 8$). Inaczej mówiąc, sieć FCN będzie przetwarzać cały obraz tylko raz i stworzy siatkę o rozmiarze 8×8 , w której każda komórka będzie zawierać dziesięć wartości liczbowych (pięć prawdopodobieństw przynależności do klas, jeden wynik „obiektości” i cztery współrzędne ramki ograniczającej). Otrzymujemy taki sam efekt jak w przypadku przesuwania pierwotnej sieci splotowej po obrazie po osiem kroków w każdym wierszu i po osiem w każdej kolumnie. Aby to sobie zwizualizować, wyobraź sobie

²⁶ Istnieje jeden wyjątek: warstwa splotowa wykorzystująca uzupełnianie zerami typu "valid" nie będzie działać prawidłowo, jeżeli rozmiar danych wejściowych będzie mniejszy od rozmiaru jądra.

²⁷ Zakładamy tutaj, że w całej sieci wykorzystywane jest tylko uzupełnianie zerami typu "same" – uzupełnianie zerami typu "valid" zmniejszałoby rozmiar map cech. Co więcej, wartość 448 można podzielić kilka razy przez 2 i uzyskać wartość 7 bez żadnego błędu zaokrąglenia. Jeżeli któraś warstwa będzie zawierała krok o wartości innej niż 1 lub 2, może pojawić się jakiś błąd zaokrąglania, zatem mogłyby pojawiać się jeszcze mniejsze mapy cech.



Rysunek 14.25. Ta sama w pełni połączona sieć splotowa przetwarzająca obraz mały (po lewej) i duży (po prawej)

podzielenie pierwotnego obrazu na siatkę 14×14 , w której przesuwamy ramkę 7×7 — występują $8 \times 8 = 64$ możliwe położenia tej ramki, stąd 8×8 prognoz. Technika ta jest jednak znacznie wydajniejsza, ponieważ sieć analizuje obraz tylko jeden raz. Istnieje również bardzo popularna architektura wykrywania obiektów o nazwie YOLO (ang. You Only Look Once — patrzysz tylko raz), którą zajmiemy się w kolejnym punkcie.

Sieć YOLO

Sieć YOLO jest niezwykle szybką i dokładną architekturą wykrywania obiektów, zaproponowaną przez Josepha Redmona i in. w publikacji z 2015 roku (<https://arxiv.org/abs/1506.02640>)²⁸, a następnie udoskonaloną w 2016 (YOLOv2 — <https://arxiv.org/abs/1612.08242>)²⁹ i 2018 roku (YOLOv3 — <https://arxiv.org/abs/1804.02767>)³⁰. Jest ona tak szybka, że można obejrzeć jej działanie w czasie rzeczywistym na filmiku, co Redmon zademonstrował pod adresem <https://www.youtube.com/watch?v=MPU2HistivI>.

Architektura YOLOv3 przypomina omówioną właśnie strukturę sieci splotowej, wprowadzono w niej jednak kilka istotnych usprawnień:

- Generuje nie jedną ramkę ograniczającą na każdą komórkę siatki, lecz pięć i każda z nich zawiera wynik „obiektowości”. Zostaje również obliczonych 20 prawdopodobieństw przynależ-

²⁸ Joseph Redmon i in., *You Only Look Once: Unified, Real-Time Object Detection*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2016), s. 779 – 788.

²⁹ Joseph Redmon i Ali Farhadi, *YOLO9000: Better, Faster, Stronger*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2017), s. 6517 – 6525.

³⁰ Joseph Redmon i Ali Farhadi, *YOLOv3: An Incremental Improvement*, arXiv preprint arXiv:1804.02767 (2018).

ności do klas w każdej komórce siatki, gdyż model został wyuczony na zestawie danych PASCAL VOC, zawierającym właśnie 20 klas. Oznacza to łącznie 45 wartości liczbowych na każdą komórkę: pięć ramek ograniczających (po cztery współrzędne na każdą ramkę), pięć wyników „obiektywości” i 20 prawdopodobieństw przynależności do klas.

- Nie przewiduje współrzędnych bezwzględnych środków ramek ograniczających, lecz przesunięcie względne wobec współrzędnych komórki siatki, gdzie (0, 0) oznacza lewy górny róg komórki, a natomiast (1, 1) prawy dolny róg. Model YOLOv3 uczy się przewidywać wyłącznie ramki ograniczające, których środek mieści się w tej komórce (ale sama ramka ograniczająca zazwyczaj wykracza daleko poza tę komórkę). Współrzędne ramki ograniczającej są traktowane logistyczną funkcją aktywacji, dzięki czemu pozostają w zakresie od 0 do 1.
- Przed rozpoczęciem uczenia sieci neuronowej model YOLOv3 wyszukuje wymiary pięciu reprezentatywnych ramek ograniczających, zwanych **ramkami mocującymi** (ang. *anchor boxes*) lub **poprzednikami ramek ograniczających** (ang. *bounding box priors*). Dokonuje tego poprzez potraktowanie wysokości i szerokości ramek danych uzyskanych dla obrazów uczących algorytmem centroidów (zob. rozdział 9.). Na przykład jeżeli obrazy uczące zawierają wielu pieszych, to jedna z ramek mocujących będzie miała prawdopodobnie wymiary typowego pieszego. Gdy sieć neuronowa prognozuje pięć ramek ograniczających w każdej komórce, przewiduje ona w istocie stopień przeskalowania każdej ramki mocującej. Założymy na przykład, że jedna ramka mocująca ma 100 pikseli wysokości i 50 pikseli szerokości, natomiast sieć przewiduje współczynnik przeskalowania pionowego równy 1,5 i przeskalowania poziomego równy 0,9 (dla jednej z komórek siatki). W takiej sytuacji przewidywana ramka ograniczająca będzie miała rozmiar 150×45 . Mówiąc dokładniej, sieć przewiduje dla każdej komórki siatki i każdej ramki mocującej logarytm współczynników przeskalowania. Obecność takich poprzedników zwiększa prawdopodobieństwo wyznaczenia ramek ograniczających o odpowiednich rozmiarach, a także przyspiesza proces uczenia, ponieważ model szybciej nauczy się wyznaczać właściwe ramki ograniczające.
- Sieć jest uczona za pomocą obrazów o różnych skalach: co kilka grup zostaje losowo wyznaczony nowy rozmiar obrazów (w zakresie od 330×330 do 608×608 pikseli). W ten sposób sieć uczy się rozpoznawać obiekty w różnych skalach. Do tego można korzystać z sieci YOLOv3 w różnych skalach: w mniejszej skali będzie ona mniej dokładna, ale szybsza w stosunku do skali większej, dlatego możesz dobrze odpowiedni kompromis.

Znajdziemy tu także kilka innych interesujących innowacji, takich jak wprowadzenie połączeń pomijających w celu odzyskania części rozdzielczości przestrzennej utraconej w sieci CNN (omówię tę kwestię już niebawem, podczas opisu segmentacji semantycznej). W publikacji z 2016 roku autorzy zaprezentowali model YOLO9000, wykorzystujący klasyfikację hierarchiczną: model prognozuje prawdopodobieństwo dla każdego węzła w wizualnej strukturze hierarchicznej o nazwie **WordTree**. Dzięki temu sieć może z dużą dozą pewności przewidywać, że na przykład rozpoznaje na zdjęciu psa, chociaż nie ma pewności co do jego rasy. Gorąco zachęcam do zapoznania się ze wszystkimi trzema artykułami, tym bardziej że są napisane bardzo przystępnie oraz zawierają znakomite przykłady stopniowego usprawniania systemów uczenia głębokiego.

Wskaźnik mAP

Bardzo popularnym wskaźnikiem w zadaniach wykrywania obiektów jest **średnia wartość przeciętna precyzji** (ang. *mean Average Precision* — mAP). Wyrażenie „średnia wartość przeciętna” brzmi trochę jak „masło maślano”, prawda? Aby zrozumieć ten wskaźnik, przypomnij sobie dwa wskaźniki zadań klasyfikacji omówione w rozdziale 3.: precyzję i pełność. Pamiętasz zapewne istniejącą pomiędzy nimi zależność: im większa pełność, tym mniejsza precyzja. Zostało to zilustrowane za pomocą krzywej precyzji/pełności (rysunek 3.5). Aby przedstawić tę krzywą za pomocą jednej wartości liczbowej, możemy obliczyć znajdujący się pod nią obszar (AUC). Zwróć jednak uwagę, że krzywa ta może zawierać kilka miejsc, w których precyzja rośnie wraz ze wzrostem pełności, zwłaszcza w przypadku małych wartości pełności (widać to w lewej górnej części rysunku 3.5). Jest to jedna z przyczyn powstania wskaźnika mAP.

Załóżmy, że klasyfikator cechuje się precyzją o wartości 90% przy 10% pełności, ale precyzją 96% przy 20% pełności. Tutaj zasadniczo nie ma co myśleć o kompromisie: bardziej logiczne jest użyć klasyfikatora przy pełności 20% niż przy 10%, ponieważ uzyskujemy zarówno lepszą pełność, jak i precyzję. Zatem powinniśmy szukać nie precyzji przy 10% pełności, lecz maksymalnej precyzji, jaką klasyfikator może zagwarantować dla co najmniej 10% pełności. Znaleziona precyzja powinna wynosić 96%, a nie 90%. Jednym ze sposobów całkiem rzetelnego określenia wydajności modelu jest więc obliczenie maksymalnej precyzji uzyskiwanej przy co najmniej 0% pełności, następnie dla 10% pełności, 20% i aż do 100%, a potem wyliczenie średniej z tych maksymalnych precyzji. Jest to wskaźnik **wartości przeciętnej precyzji** (ang. *Average Precision* — AP). Jeżeli występuje większa liczba klas, możemy obliczyć ich wskaźniki AP, a następnie wy ciągnąć z nich średnią. To jest właśnie wskaźnik mAP!

W systemie wykrywania obiektów występuje dodatkowy poziom złożoności: co w przypadku, gdy system wykryje właściwą klasę, ale w złym miejscu (tzn. ramka ograniczająca jest umieszczona zupełnie gdzie indziej)? Zdecydowanie nie powinniśmy traktować tego jako prognozy pozytywnej. Jednym z rozwiązań jest wyznaczenie progu IOU: na przykład możesz stwierdzić, że prognoza będzie właściwa jedynie wtedy, gdy wartość IOU będzie większa powiedzmy od 0,5, a prognozowana klasa będzie prawidłowa. Odpowiedni wskaźnik mAP jest zazwyczaj zapisywany jako mAP@0,5 (ewentualnie mAP@50% lub, rzadziej, AP₅₀). W niektórych konkursach (np. wyzwanie PASCAL VOC) wykorzystywane jest właśnie to rozwiązanie. W innych (takich jako konkurs COCO) wskaźnik mAP jest obliczany dla różnych wartości progowych IOU (0,50, 0,55, 0,60, ..., 0,95), a ostateczny wynik stanowi średnią wszystkich tych wskaźników mAP (zapisywane jest to w postaci AP@[.50:.95] lub AP@[.50:0.05:.95]). Tak, oznacza to średnią ze średnich wartości przeciętnych.

W repozytorium GitHub znajduje się kilka implementacji modelu YOLO wykorzystujących moduł TensorFlow. Warto sprawdzić zwłaszcza implementację stworzoną w module TensorFlow (<https://github.com/zzh8829/yolov3-tf2>), zaprojektowaną przez Zihao Zanga. Inne modele wykrywania obiektów są dostępne w ramach projektu TensorFlow Models, z których wiele zawiera już wytrenowane wagi, niektóre z nich zostały zaś przeniesione do serwisu TF Hub, między innymi dość popularne modele SSD (<https://arxiv.org/abs/1512.02325>)³¹ i Faster R-CNN (<https://arxiv.org/abs/1506.01497>)³². SSD także jest modelem „jednostrzałowym”, przypominającym architekturę YOLO. Model Faster R-CNN jest bardziej skomplikowany: obraz przechodzi najpierw przez sieć CNN,

³¹ Wei Liu i in., *SSD: Single Shot Multibox Detector*, „Proceedings of the 14th European Conference on Computer Vision” 1 (2016), s. 21 – 37.

³² Shaoqing Ren i in., *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, „Proceedings of the 28th International Conference on Neural Information Processing Systems” 1 (2015), s. 91 – 99.

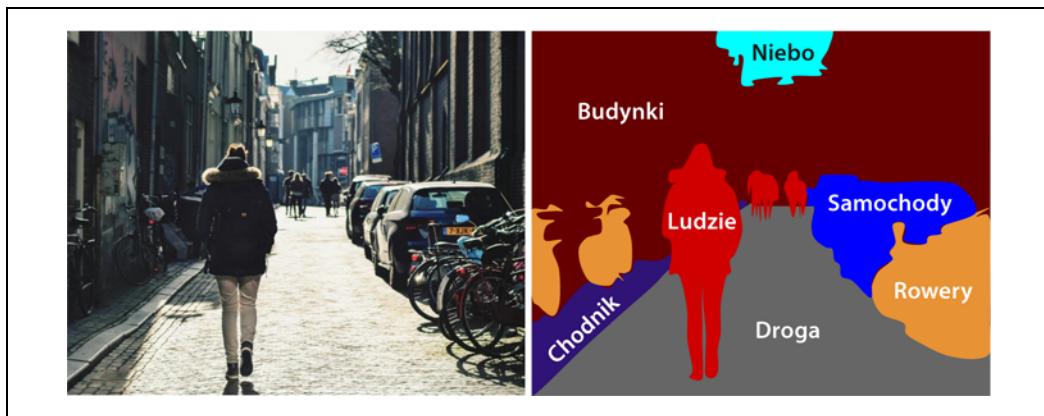
następnie wynik jest przekazywany **sieci proponowania rejonów** (ang. *Region Proposal Network* — RPN), której zadaniem jest proponowanie ramek ograniczających o największym prawdopodobieństwie przechowywania obiektu; każda z tych ramek jest przetwarzana przez klasyfikator, na bazie obciętego wyniku sieci splotowej.

Wybór systemu wykrywania obiektów zależy od wielu czynników: szybkości, dokładności, dostępnych gotowych modeli, czasu uczenia, złożoności itd. Publikacje zawierają tabele ze wskaźnikami, ale istnieje całkiem duże różnicowanie wśród środowisk testowych, a technologie rozwijają się tak szybko, że trudno przeprowadzać sprawiedliwe porównanie, które będzie przydatne dla większości osób i pozostało aktualne dłużej niż kilka miesięcy.

Zatem lokalizujemy obiekty poprzez rysowanie ramek ograniczających wokół nich. Śivetnie! Był może jednak wymagał większej precyzji. Zobaczmy, co możemy osiągnąć po zejściu do poziomu pikseli.

Segmentacja semantyczna

W **segmentacji semantycznej** (ang. *semantic segmentation*) każdy piksel jest klasyfikowany zgodnie z klasą obiektu, do którego przynależy ten piksel (np. do drogi, pieszego, samochodu, budynku itd.), co zostało zaprezentowane na rysunku 14.26. Zwróć uwagę, że *nie są* rozróżniane różne obiekty należące do tej samej klasy. Na przykład wszystkie rowery widoczne po prawej stronie segmentowanego obrazu tworzą wielkie skupisko pikseli. Głównym problemem w tym zadaniu jest fakt, że obrazy przechodzące przez kolejne warstwy sieci splotowej stopniowo tracą rozdzielcość przestrzenną (z powodu warstw zawierających kroki większe od 1), zatem tradycyjna sieć konwolucyjna może stwierdzić, że gdzieś w lewej dolnej części obrazu znajduje się osoba, ale nie możemy spodziewać się po tej sieci większej precyzji.

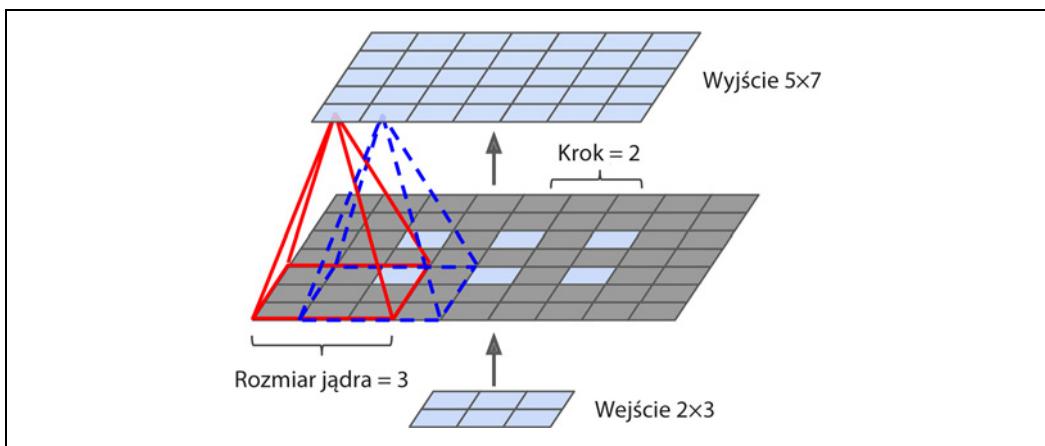


Rysunek 14.26. Segmentacja semantyczna

Podobnie jak w przypadku wykrywania obiektów, zaproponowano wiele różnych technik rozwiązywania tego problemu, wśród nich sporo dość skomplikowanych. Jednak Jonathan Long i in. zaproponowali dość prostą technikę we wspomnianej już publikacji z 2015 roku. Autorzy przekształcają gotową sieć splotową w sieć FCN. Obraz wejściowy jest traktowany ogólnym krokiem o rozmiarze 32

(tzn. po dodaniu wszystkich kroków większych od 1), co oznacza, że ostatnia warstwa generuje mapę cech 32-krotnie mniejszą od obrazu wejściowego. Jest oczywiste, że taki obraz wynikowy jest zbyt zredukowany, dlatego autorzy dodali pojedynczą warstwę **ekspansji** (ang. *upsampling layer*) mnożącą rozdzielcość przestrzenną 32-krotnie.

Istnieje kilka technik ekspansji (zwiększenia rozmiaru obrazu), np. interpolacja dwuliniowa, jednak sprawdzają się one dobrze przy mnożnikach $\times 4$ lub $\times 8$. Dlatego wprowadzamy transponowaną warstwę splotową (ang. *transposed convolutional layer*)³³ — jest ona równoważna rozciągnięciu obrazu poprzez wstawienie pustych wierszy i rzędów (wypełnionych zerami), a następnie przeprowadzeniu operacji splotu (rysunek 14.27). Ewentualnie niektórzy traktują ją jako tradycyjną warstwę splotową wykorzystującą kroki o wartościach ułamkowych (np. $\frac{1}{2}$ na rysunku 14.27). Można zainicjalizować transponowaną warstwę splotową tak, aby przeprowadzała operację przypominającą interpolację liniową, ale warstwa ta jest modyfikowalna, dlatego w trakcie uczenia będzie uzywać coraz lepsze rezultaty. W interfejsie tf.keras możesz wykorzystać warstwę Conv2DTranspose.



Rysunek 14.27. Ekspansja za pomocą transponowanej warstwy splotowej



W transponowanej warstwie splotowej krok definiuje stopień rozciągnięcia danych wejściowych, a nie rozmiar kroku filtra, zatem im większa jego wartość, tym większy obraz wyjściowy (odwrotnie niż w przypadku warstw splotowych lub łączących).

Operacje splotu w module TensorFlow

Biblioteka TensorFlow zawiera również kilka innych rodzajów warstw splotowych:

`keras.layers.Conv1D`

Tworzy warstwę splotową dla jednowymiarowych danych wejściowych, takich jak szeregi czasowe czy tekst (sekwencje liter lub słów), o czym przekonasz się w rozdziale 15.

³³ Czasami można spotkać się z terminem **warstwa rozplotowa** lub **dekonwolucyjna** (ang. *deconvolution layer*), nie przeprowadza ona jednak matematycznej operacji rozplotu, dlatego należy unikać stosowania tej nazwy.

`keras.layers.Conv3D`

Tworzy warstwę splotową dla trójwymiarowych danych wejściowych, np. trójwymiarowych skanów wykonanych tomografem PET.

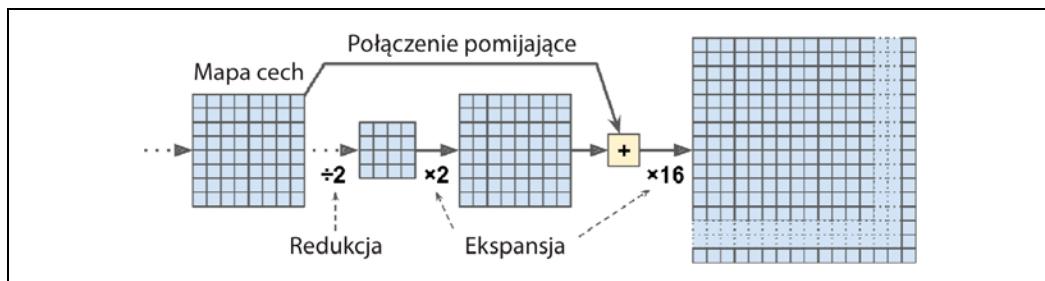
`dilation_rate`

Wyznaczenie hiperparametru `dilation_rate` o wartości większej niż 2 w dowolnej warstwie splotowej tworzy **rozszerzoną warstwę splotową** (ang. *à trous convolutional layer* — francuski wyraz *à trous* oznacza „z dziurami”). Jest to standardowa warstwa splotowa zawierająca filtr rozszerzony o dodatkowe wiersze i kolumny wypełnione zerami („dziurami”). Na przykład filtr 1×3 o wartości $[[1, 2, 3]]$ możemy rozszerzyć za pomocą **współczynnika rozszerzalności** (ang. *dilution rate*) o wartości 4, dzięki czemu uzyskamy **filtrow rozcieńczony** (ang. *diluted filter*) o postaci $[[1, 0, 0, 0, 2, 0, 0, 0, 3]]$. W ten sposób warstwa splotowa uzyska znacznie szersze pole recepcyjne bez zwiększenia obciążenia obliczeniowego i dodawania parametrów.

`tf.nn.depthwise_conv2d()`

Funkcja ta tworzy **warstwę splotową po głębokości** (ang. *depthwise convolutional layer*). musisz jednak samodzielnie stworzyć zmienne. Stosuje ona niezależnie każdy filtr wobec wszystkich kanałów wejściowych. Zatem jeśli mamy f_n filtrów i $f_{n'}$ kanałów, to uzyskamy $f_n \times f_{n'}$ map cech.

Rozwiążanie to nie jest złe, ale brakuje mu precyzji. W celu uzyskania lepszych rezultatów autorzy dodali połączenia pomijające, które wychodzą z niższych warstw: na przykład obraz wyjściowy został rozszerzony o współczynnik ekspansji 2 (zamiast 32) i dodano wynik z warstwy niższej o dwa razy większej rozdzielczości. Rezultat został następnie rozszerzony szesnastokrotnie, dzięki czemu uzyskano współczynnik ekspansji równy 32 (rysunek 14.28). Autorzy odzyskali w ten sposób część rozdzielczości przestrzennej, która została utracona we wcześniejszych warstwach łączących. W najlepszej architekturze wprowadzono jeszcze jedną taką warstwę pomijającą, która pozwala odzyskać jeszcze więcej szczegółów z wcześniejszej warstwy. Mówiąc krótko, wynik sieci splotowej przechodzi jeszcze przez następne etapy: powiększenie $\times 2$, dodanie wyniku z poprzedniej warstwy (o odpowiedniej skali), powiększenie $\times 2$, dodanie wyniku z jeszcze niższej warstwy, a na koniec powiększenie $\times 8$. Można nawet powiększyć wynik o rozmiar przekraczający obraz pierwotny — technika ta służy zwiększeniu rozdzielczości obrazu i jest nazywana **superrozdzielcością** (ang. *super-resolution*).



Rysunek 14.28. Połączenia pomijające odzyskują część rozdzielczości przestrzennej z niższych warstw

Także w tym przypadku znajdziesz w repozytorium GitHub implementacje segmentacji semantycznej utworzone za pomocą modułu TensorFlow (jeszcze nie ma implementacji dla wersji TensorFlow 2); dostępne są nawet wytrenowane modele **segmentacji instancji** (ang. *instance segmentation*) w projekcie TensorFlow Models. Segmentacja instancji przypomina segmentację semantyczną, jednak obiekty należące do tej samej klasy nie są tutaj scalane w jedną dużą grupę pikseli, lecz każdy z nich jest odróżniany od pozostałych (np. są rozpoznawane poszczególne rowery). Obecnie dostępne w projekcie TensorFlow Models modele segmentacji instancji bazują na architekturze Mask R-CNN, która została opisana w publikacji z 2017 roku (<https://arxiv.org/abs/1703.06870>)³⁴: model Faster R-CNN został tu rozwinięty o dodatkową maskę pikseli generowaną dla każdej ramki ograniczającej. Otrzymujemy więc nie tylko ramkę otaczającą każdy obiekt (wraz z wyznaczonym zestawem prawdopodobieństw przynależności do klas), lecz także maskę pikseli wykrywającą wewnątrz ramki piksele składające się na dany obiekt.

Jak widać, dziedzina głębokiego widzenia maszynowego jest rozległa i bardzo szybko poznawana; każdego roku pojawiają się nowe struktury oparte na splotowych sieciach neuronowych. Postępy dokonane w ciągu zaledwie kilku lat są zdumiewające, a naukowcy koncentrują się na coraz trudniejszych problemach, takich jak **uczenie przeciwstawne** (ang. *adversarial learning*; uczenie sieci rozpoznawania obrazów mających na celu jej oszukanie), wyjaśnialność (zrozumienie, dlaczego sieci przeprowadzają taką, a nie inną klasyfikację), **generowanie realistycznych obrazów** (wróćmy do tego zagadnienia w rozdziale 17.) czy **uczenie „jednostrzałowe”** (ang. *single-shot learning*; system zdolny do rozpoznania obiektu po zaledwie jednokrotnym ujrzeniu go). Niektórzy analizują całkowicie nowatorskie struktury, takie jak zaproponowane przez Geoffreya Hintona **sieci kapsułowe** (ang. *capsule networks*; <https://openreview.net/pdf?id=HJWLfGWRb>³⁵; dla zainteresowanych osób przygotowałem prezentację na ich temat: <https://homl.info/capsnet>, wraz z listingiem umieszczonym w notatniku Jupyter). Przejdźmy do następnego rozdziału, w którym nauczysz się przetwarzać dane sekwencyjne, takie jak szeregi czasowe, za pomocą sieci rekurencyjnych i splotowych.

Ćwiczenia

1. Jakie ma zalety sieć CNN w porównaniu do w pełni połączonej głębokiej sieci neuronowej w zadaniach klasyfikacji obrazów?
2. Wyobraź sobie sieć CNN składającą się z trzech warstw splotowych, z których każda zawiera filtr 3×3 , krok o wartości 2 i uzupełnianie zerami typu "same". Najniższa warstwa generuje 100 map cech, środkowa — 200, a góra — 400. Obrazy wejściowe mają trzy kanały (RGB) i rozmiar 200×300 pikseli.

Jaką całkowitą liczbę parametrów ma taka sieć? Jeżeli korzystamy z 32-bitowych wartości zmiennoprzecinkowych, jakiej minimalnej ilości pamięci operacyjnej będzie wymagała ta sieć do prognozowania wartości dla jednego przykładu? Ile pamięci byłoby wymagane dla minigrupy składającej się z 50 obrazów?

³⁴ Kaiming He i in., *Mask R-CNN*, arXiv preprint arXiv:1703.06870 (2017).

³⁵ Geoffrey Hinton i in., *Matrix Capsules with EM Routing*, „Proceedings of the International Conference on Learning Representations” (2018).

- 3.** Jakie pięć czynności możesz spróbować wykonać w przypadku niedoboru pamięci operacyjnej na karcie graficznej podczas uczenia głębokiej sieci splotowej?
 - 4.** Dlaczego lepiej dodać maksymalizującą warstwę łączącą zamiast warstwy splotowej o takim samym kroku?
 - 5.** Kiedy należy wstawić warstwę normalizacji odpowiedzi lokalnej?
 - 6.** Wymień najważniejsze usprawnienia architektury AlexNet względem sieci LeNet-5. W podobny sposób porównaj sieci GoogLeNet, ResNet, SENet i Xception.
 - 7.** Czym jest w pełni splotowa sieć? Jak można przekształcić warstwę gęstą w splotową?
 - 8.** Wymień największą trudność techniczną związaną z segmentacją semantyczną.
 - 9.** Utwórz od podstaw własną sieć splotową i spróbuj uzyskać jak największą dokładność dla zestawu danych MNIST.
- 10.** Uczenie transferowania w zadaniach klasyfikacji dużych obrazów:
- a.** Stwórz zbiór uczący składający się przynajmniej ze 100 obrazów na każdą klasę. Przykładowo, możesz klasyfikować własne zdjęcie pod względem lokalizacji (plaża, góry, miasto itd.) lub ewentualnie skorzystać z istniejącego zbioru danych (np. z projektu TensorFlow Datasets).
 - b.** Podziel zestaw danych na zbiory uczący, walidacyjny i testowy.
 - c.** Zbuduj potok danych wejściowych zawierający odpowiednie operacje przetwarzania wstępnego, a także dodatkowo operacje dogenerowania danych.
 - d.** Dostosuj gotowy model do tego zestawu danych.
- 11.** Zapoznaj się z samouczkiem Style Transfer modułu TensorFlow (https://www.tensorflow.org/tutorials/generative/style_transfer). Jest to interesujący sposób generowania dzieł sztuki za pomocą uczenia głębokiego.

Rozwiązania tych ćwiczeń znajdziesz w dodatku A.

Przetwarzanie sekwencji za pomocą sieci rekurencyjnych i splotowych

Trwa mecz piłki nożnej. Środkowy pomocnik odbiera piłkę przeciwnikowi w strefie środkowej. Widzisz przed sobą wolne pole i przewidując zachowanie kolegi z drużyny, zaczynasz pędzić w kierunku bramki. Pomocnik idealnie posyła piłkę pomiędzy Ciebie a obrońców drugiej drużyny, koordynujesz pracę wszystkich mięśni i strzelasz gola (czym wzbudzasz szalony entuzjazm na trybunach). Nasze mózgi przez cały czas przewidują przyszłość, czy to poprzez dokończenie zdania rozpoczętego przez rozmówcę, czy też spodziewając się zapachu kawy w kuchni. W tym rozdziale przeanalizujemy **rekurencyjne sieci neuronowe** (ang. *Recurrent Neural Networks* — RNN), odmianę sieci przewidującą przyszłość (oczywiście do pewnego stopnia). Są one w stanie analizować dane **szeregów czasowych** (ang. *time series*), np. ceny akcji giełdowych, doradzając, czy należy je skupować, czy sprzedawać. W autonomicznych systemach prowadzenia pojazdów sieci te mogą przewidywać trajektorie uczestników ruchu i zapobiegać kolizjom. Mówiąc ogólniej, mogą one pracować na sekwencjach danych o dowolnej długości, w przeciwieństwie do wcześniej omówionych sieci, w których dane mają ustalone rozmiary. Na przykład przyjmują one dane w postaci zdań, dokumentów lub próbek dźwiękowych, co okazuje się bardzo przydatne w systemach przetwarzania języka naturalnego, takich jak automatyczne tłumaczenie czy przetwarzanie mowy na tekst.

W tym rozdziale przyjrzymy się najpierw fundamentalnym koncepcjom definiującym sieci rekurencyjne oraz metodom ich uczenia za pomocą propagacji wstecznej w czasie, po czym wykorzystamy je do prognozowania szeregów czasowych. Następnie zajmiemy się dwoma głównymi wyzwaniami związanymi z sieciami RNN:

- Niestabilne gradienty (zob. rozdział 11.), z którymi można sobie radzić na wiele różnych sposobów, np. poprzez porzucanie rekurencyjne czy normalizację warstwy rekurencyjnej.
- Ograniczona (bardzo) pamięć krótkotrwała, którą można udoskonalić za pomocą komórek LSTM i GRU.

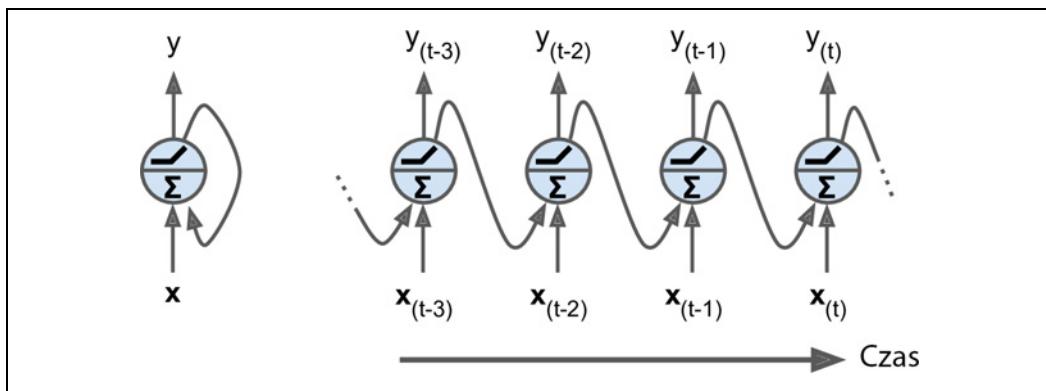
Nie tylko sieci rekurencyjne służą do przetwarzania danych sekwencji: z małymi sekwencjami radzi sobie również tradycyjna sieć gęsta, a w przypadku bardzo długich sekwencji (np. dźwiękowych czy tekstowych) okazuje się, że całkiem dobrze sprawdzają się również sieci splotowe. Omówię obydwa te rozwiązania, a rozdział podsumuję implementacją modelu **WaveNet** — jest to struktura sieci rekurencyjnej, która przetwarza sekwencje składające się z dziesiątek tysięcy

taktów. W rozdziale 16. będziemy dalej zajmować się sieciami RNN i nauczysz się wykorzystywać je w zadaniach przetwarzania języka naturalnego, a także poznasz nowsze architektury, oparte na mechanizmach uwagi. Zatem do dzieła!

Neurony i warstwy rekurencyjne

Dotychczas koncentrowaliśmy się na jednokierunkowych sieciach neuronowych, w których pobudzenia przepływają tylko w jedną stronę, od warstwy wejściowej do wyjściowej (w dodatku E opisałem kilka wyjątków). Rekurencyjna sieć neuronowa bardzo przypomina taką sieć jednokierunkową, z tym że zawiera również połączenia wsteczne. Przeanalizujmy najprostszy model sieci RNN, składający się z pojedynczego neuronu, który otrzymuje sygnały wejściowe i generuje sygnały wyjściowe, które przesyła z powrotem na swoje wejście (rysunek 15.1, po lewej). W każdym **takcie t** (ang. *time step*), zwany też **ramką** (ang. *frame*) taki neuron rekurencyjny otrzymuje sygnały wejściowe $x_{(t)}$ oraz wynik wygenerowany przez siebie w poprzednim taktie, $y_{(t-1)}$. W pierwszym taktie nie występuje wynik z taktu poprzedniego, dlatego zazwyczaj temu taktowi poprzedniemu zostaje wyznaczona wartość 0. Możemy ukazać tę malutką sieć w funkcji czasu (rysunek 15.1, po prawej).

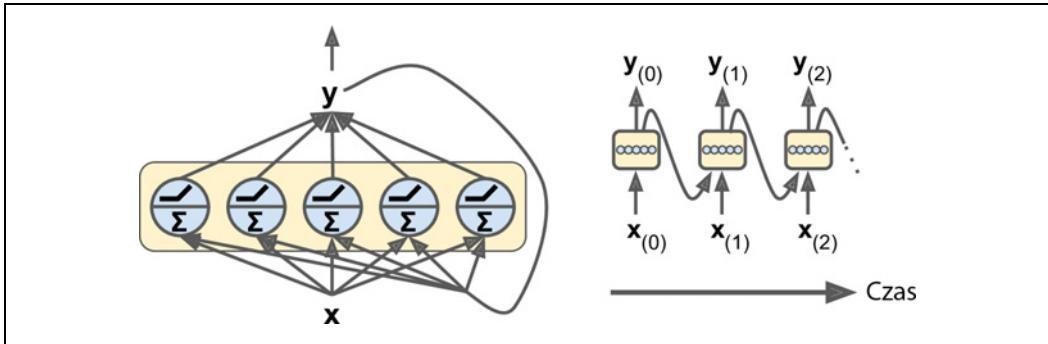
Jest to tak zwane rozwijanie sieci w czasie (widzimy ten sam neuron reprezentowany w poszczególnych taktach).



Rysunek 15.1. Neuron rekurencyjny (po lewej) rozwijany w czasie (po prawej)

Łatwo utworzyć warstwę neuronów rekurencyjnych. W każdym taktie t każdy neuron otrzymuje zarówno sygnał wejściowy $x_{(t)}$, jak i wynik wygenerowany w poprzednim taktie $y_{(t-1)}$ (rysunek 15.2). Teraz i sygnały wejściowe, i wyjściowe są wektorami (w przypadku pojedynczego neuronu mieliśmy do czynienia z wartościami skalarnymi).

Każdy neuron rekurencyjny zawiera dwa zestawy wag: jedne są przeznaczone dla danych wejściowych $x_{(t)}$, a drugie dla wyniku z poprzedniego taktu $y_{(t-1)}$. Nazwijmy te wektory wag, odpowiednio, w_x i w_y . Jeżeli weźmiemy pod uwagę całą warstwę rekurencyjną, to możemy umieścić wektory wag w dwóch macierzach, odpowiednio W_x i W_y . Wektor wynikowy całej warstwy rekurencyjnej



Rysunek 15.2. Warstwa neuronów rekurencyjnych (po lewej) rozwijana w czasie (po prawej)

może być następnie obliczany w tradycyjny sposób (równanie 15.1) (\mathbf{b} jest wektorem obciążen, a $\phi(\cdot)$ funkcja aktywacji, np. ReLU¹).

Równanie 15.1. Wynik warstwy rekurencyjnej dla jednego przykładu

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^T \mathbf{x}_{(t)} + \mathbf{W}_y^T \mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

Podobnie jak w przypadku jednokierunkowych sieci neuronowych, możemy za jednym zamachem obliczyć wynik warstwy rekurencyjnej dla całej minigrupy przykładów poprzez umieszczenie wszystkich sygnałów wejściowych z taktu t w macierzy wejściowej $\mathbf{X}_{(t)}$ (równanie 15.2).

Równanie 15.2. Wyniki warstwy neuronów rekurencyjnych dla wszystkich przykładów tworzących minigrupę

$$\begin{aligned} \mathbf{Y}_{(t)} &= \phi\left(\mathbf{X}_{(t)} \mathbf{W}_x + \mathbf{Y}_{(t-1)} \mathbf{W}_y + \mathbf{b}\right) = \\ &= \phi\left(\begin{bmatrix} \mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)} \end{bmatrix} \mathbf{W} + \mathbf{b}\right) \quad \text{gdzie } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix} \end{aligned}$$

W tym równaniu:

- $\mathbf{Y}_{(t)}$ jest macierzą $m \times n_{neurony}$ zawierającą dane wyjściowe warstwy w taktie t dla każdego elementu minigrupy (m oznacza liczbę próbek w minigrupie, a $n_{neurony}$ liczbę neuronów);
- $\mathbf{X}_{(t)}$ to macierz $m \times n_{wejścia}$ zawierająca dane wejściowe dla wszystkich przykładów ($n_{wejścia}$ określa liczbę cech wejściowych);
- \mathbf{W}_x jest macierzą $n_{wejścia} \times n_{neurony}$ zawierającą wagę połączeń dla wejść w bieżącym taktie;

¹ Zwróć uwagę, że wielu naukowców woli stosować w sieciach rekurencyjnych funkcję tangensa hiperbolicznego (tanh) zamiast funkcji ReLU. Jest tak na przykład w publikacji Vu Phama i in. pod tytułem *Dropout Improves Recurrent Neural Networks for Handwriting Recognition* z 2013 roku (<https://homl.info/91>). Można również korzystać z sieci RNN zaopatrzonych w funkcję aktywacji ReLU, co udowodnili Quoc V. Le i in. w artykule pod tytułem *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units* z 2015 roku (<https://homl.info/92>).

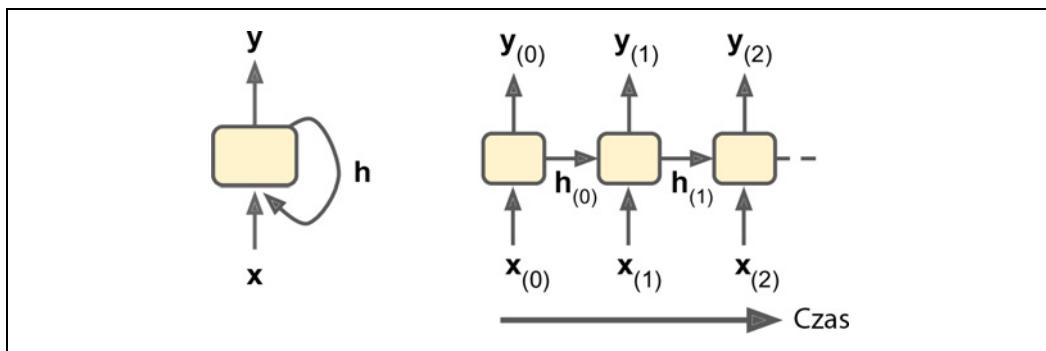
- W_y to macierz $n_{neurony} \times n_{neurony}$ zawierająca wagi połączeń dla danych wyjściowych z poprzedniego taktu;
- b jest wektorem o rozmiarze $n_{neurony}$ zawierającym człony obciążenia każdego neuronu;
- macierze wag W_x i W_y często są ze sobą łączone pionowo w pojedynczą macierz wag W o wymiarach $(n_{wejścia} + n_{neurony}) \times n_{neurony}$ (drugi wiersz w równaniu 15.2);
- notacja $[X_{(t)} \ Y_{(t-1)}]$ oznacza poziome połączenie macierzy $X_{(t)}$ i $Y_{(t-1)}$.

Zwróc uwagę, że $Y_{(t)}$ jest funkcją macierzy $X_{(t)}$ i $Y_{(t-1)}$, która z kolei stanowi funkcję macierzy $X_{(t-1)}$ i $Y_{(t-2)}$, będącej funkcją macierzy $X_{(t-2)}$ i $Y_{(t-3)}$ itd. W ten sposób $Y_{(t)}$ to funkcja wszystkich danych wejściowych od momentu $t = 0$ (tzn. $X_{(0)}, X_{(1)}, \dots, X_{(t)}$). W pierwszym taktie ($t = 0$) nie istnieją jeszcze żadne dane wyjściowe, dlatego zakładą się, że mają one wartość 0.

Komórki pamięci

Wyjście neuronu rekurencyjnego w taktie t stanowi funkcję wszystkich danych wejściowych z poprzednich ramek czasowych, dlatego możemy stwierdzić, że taka sieć zawiera coś na kształt **pamięci**. Fragment sieci neuronowej zachowujący pewne informacje o stanie w poszczególnych taktach nazywamy **komórką pamięci** (ang. *memory cell*) lub po prostu **komórką** (ang. *cell*). Pojedynczy neuron rekurencyjny lub warstwa neuronów rekurencyjnych są przykładami **komórek podstawowych** (ang. *basic cells*), które mogą zapamiętywać tylko krótkie sekwencje (zazwyczaj około 10 taktów, ale jest to zależne od zadania). W dalszej części rozdziału poznasz bardziej skomplikowane i potężne typy komórek zdolnych do zapamiętywania dłuższych sekwencji (około 10 razy dłuższych, ale tutaj także wszystko zależy od zadania).

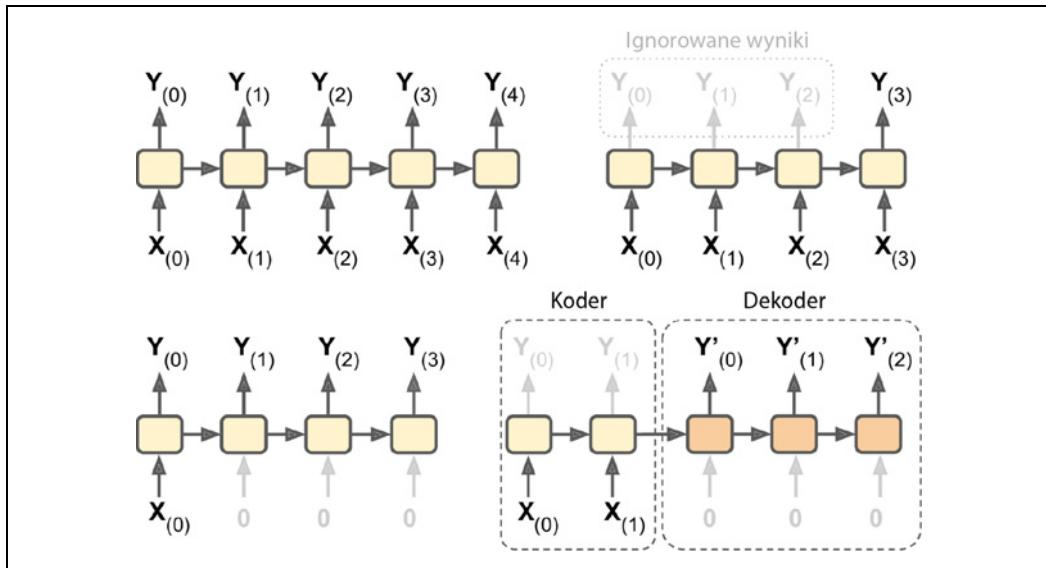
Zasadniczo stan komórki w taktie t , oznaczany symbolem $h_{(t)}$ (litera h pochodzi od angielskiego wyrazu *hidden*, czyli „ukryty”), jest funkcją danych wejściowych w tym kroku czasowym i jej stanu z poprzedniego taktu: $h_{(t)} = f(h_{(t-1)}, x_{(t)})$. Jej wyjście w kroku t , zdefiniowane jako $y_{(t)}$, oznacza również funkcję poprzedniego stanu i bieżących danych wejściowych. W przypadku dotychczas omawianych komórek podstawowych wyjście jest tożsame ze stanem, ale w bardziej zaawansowanych typach komórek nie zawsze tak się dzieje (rysunek 15.3).



Rysunek 15.3. Stan ukryty komórki może być inny od jej danych na wyjściu

Sekwencje wejść i wyjść

Sieć RNN może jednocześnie pobierać sekwencję danych wejściowych i generować za ich pomocą wyniki na wyjściach (rysunek 15.4, lewa góra część). Na przykład taka **sieć sekwencyjna** (ang. *sequence-to-sequence network*) przydaje się do prognozowania danych szeregów czasowych, takich jak ceny akcji na giełdzie: podajesz sieci ceny z N poprzednich dni, w rezultacie zaś prognozowane ceny muszą być przesunięte w czasie o jeden dzień do przodu (np. od $N-1$ dni temu do jutra).



Rysunek 15.4. Architektury sieci RSN: sekwencyjna (lewy górny róg), sekwencyjno-wektorowa (prawy górny róg), wektorowo-sekwencyjna (lewy dolny róg), koder – dekoder (prawy dolny róg)

Ewentualnie możemy przekazać sieci sekwencję danych wejściowych i ignorować wszystkie wyniki oprócz najnowszego (rysunek 15.4, prawa góra część). Innymi słowy, jest to **sieć sekwencyjno-wektorowa** (ang. *sequence-to-vector network*). Na przykład możemy przesyłać do sieci sekwencję słów pochodzących z recenzji filmu, a sieć określi wynik sentymentów (w zakresie od -1 [nienawiść] do 1 [uwielbienie]).

I odwrotnie: możemy również ciągle dostarczać ten sam wektor wejściowy w każdym taktie i generować z niego sekwencję (rysunek 15.4, lewa dolna część). Jest to **sieć wektorowo-sekwencyjna** (ang. *vector-to-sequence network*). Przykładowo na wejściu możemy wstawić obraz (lub wynik sieci spłotowej), a na wyjściu zostałby wygenerowany jego opis.

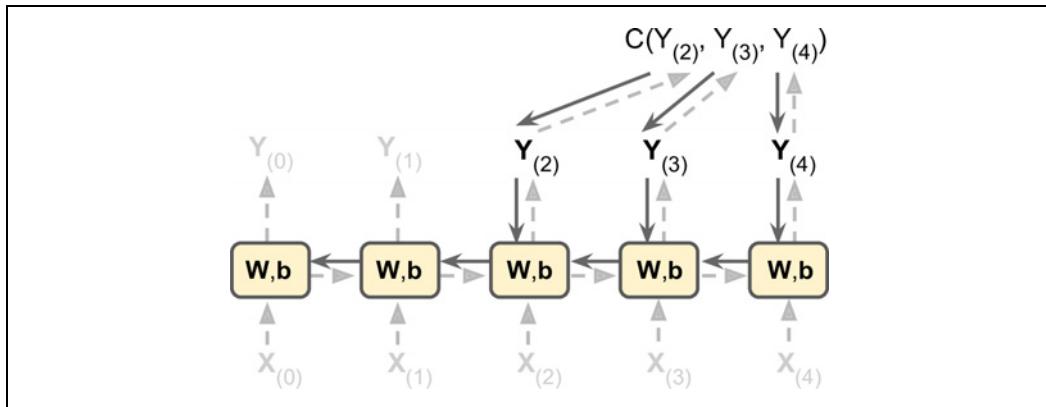
W ostatnim przypadku możemy mieć sieć sekwencyjno-wektorową, zwaną **koderem** (ang. *encoder*), do której na końcu byłaby przyłączona sieć wektorowo-sekwencyjna, tzw. **dekoder** (ang. *decoder*) (rysunek 15.4, prawa dolna część). Takie rozwiązanie może przydawać się podczas tłumaczenia zdania na inny język. Należałyby przesyłać do sieci zdanie napisane w jednym języku i koder przekształciłby je do postaci wektorowej, natomiast dekoder przeistoczyłby ten wektor w zdanie już przetłumaczone na inny język. Taki dwuetapowy model, zwany **koder – dekoder**, znacznie lepiej tłumaczy tekst na bieżąco niż zwykła sekwencyjna sieć RSN (taka jak zaprezentowana w lewym górnym rogu), ponieważ

ostatnie słowa w zdaniu mogą wpływać na pierwsze wyrazy w przetłumaczonym tekście, dlatego przed przystąpieniem do przekładu należy zaczekać na wczytanie całego zdania. W rozdziale 16. zaimplementujemy model koder – dekoder (przekonasz się, że opisywany mechanizm jest nieco bardziej skomplikowany, niż wynikałoby z rysunku 15.4).

Wygląda to całkiem obiecująco, ale jak uczymy rekurencyjną sieć neuronową?

Uczenie sieci rekurencyjnych

Aby wytrenować sieć RSN, należy ją rozwinać w czasie (dokładnie tak, jak to robiliśmy wcześniej), a następnie zastosować wobec niej klasyczny algorytm propagacji wstecznej (rysunek 15.5). Strategia ta jest nazywana **propagacją wsteczną w czasie** (ang. *backpropagation through time* — BPTT).



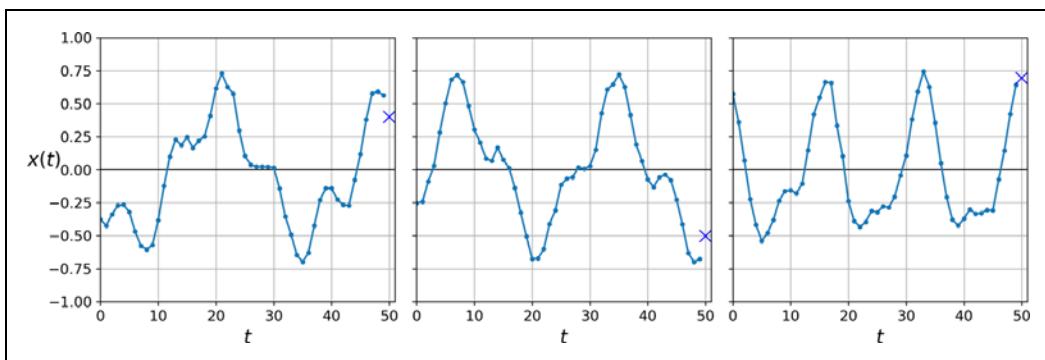
Rysunek 15.5. Propagacja wsteczna w czasie

Podobnie jak w przypadku klasycznej propagacji wstecznej, najpierw wykonywany jest przebieg do przodu poprzez rozwiniętą sieć (przerywane strzałki), po czym sekwencja wyjściowa jest oceniana za pomocą funkcji kosztu $C(Y_{(0)}, Y_{(1)}, \dots, Y_{(T)})$ (gdzie T oznacza maksymalny takt). Zwróć uwagę, że funkcja kosztu może ignorować niektóre wyjścia, co widać na rysunku 14.5 (np. w sieci sekwencyjno-wektorowej ignorowane są wszystkie sygnały wyjściowe oprócz ostatniego). Następnie gradienty tejże funkcji kosztu zostają wstecznie rozprowadzone po całej rozwiniętej sieci (ciągłe strzałki). Na koniec parametry modelu są aktualizowane za pomocą gradientów obliczonych na etapie BPTT. Zwróć uwagę, że gradienty są rozprowadzane przez wszystkie wyjścia wykorzystane przez funkcję kosztu, a nie wyłącznie przez ostateczne wyjście (na rysunku 15.5 funkcja kosztu jest obliczana przy użyciu ostatnich trzech wyjść sieci, $Y_{(2)}$, $Y_{(3)}$ i $Y_{(4)}$, zatem gradienty będą się rozprzestrzeniały przez te trzy wyjścia, ale już nie przez wyjścia $Y_{(0)}$ i $Y_{(1)}$). Co więcej, te same parametry W i b są wykorzystywane w każdym taktcie, więc propagacja wsteczna zsumuje wszystkie ramki czasowe.

Na szczęście od zajmowania się wszelką złożonością jest interfejs `tf.keras`, przejdźmy zatem do pisania kodu!

Prognozowanie szeregów czasowych

Założymy, że analizujesz liczbę odwiedzin aktywnych użytkowników na Twojej stronie w ciągu godziny, dzienne wahania temperatury w Twoim mieście lub kondycję finansową Twojej firmy mierzoną kwartalnie za pomocą wielu wskaźników. We wszystkich tych sytuacjach dane będą sekwencją co najmniej jednej wartości na każdy takt. Jest to tak zwany **szereg czasowy** (ang. *time series*). W pierwszych dwóch przypadkach występuje tylko jedna wartość w każdym taktie, zatem jest to **jednowymiarowy szereg czasowy** (ang. *univariate time series*), podczas gdy w przykładzie finansowym występowałoby w każdym taktie wiele wartości (np. przychody firmy, zadłużenie itd.), więc mamy tu do czynienia z **wielowymiarowym szeregiem czasowym** (ang. *multivariate time series*). Typowym zadaniem jest przewidywanie przyszłych wartości, czyli **prognozowanie** (ang. *forecasting*). Innym popularnym zadaniem jest wypełnianie luk — przewidywanie (ale wstecz) brakujących wartości z przeszłości. Proces ten jest nazywany **imputacją** (ang. *imputation*). Na przykład na rysunku 15.6 widzimy trzy jednowymiarowe szeregi czasowe, każdy o długości 50 taktów; celem tutaj jest prognozowanie wartości w następnym taktie (reprezentowanej przez symbol X) dla każdego z nich.



Rysunek 15.6. Prognozowanie szeregów czasowych

Dla uproszczenia wykorzystamy szereg czasowy wygenerowany przez funkcję `generate_time_series()`:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10)) #fala 1.
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) #+fala 2.
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) #+szum
    return series[..., np.newaxis].astype(np.float32)
```

Funkcja ta tworzy wyznaczaną przez nas liczbę szeregów czasowych (służy do tego argument `batch_size`) o długości `n_steps`; w każdym szeregu czasowym występuje tylko jedna wartość na każdy takt (czyli wszystkie szeregi czasowe są tu jednowymiarowe). Funkcja zwraca tablicę NumPy o wymiarach [`rozmiar grupy, takty, 1`], gdzie każdy szereg czasowy stanowi sumę dwóch sinusoid o stałych amplitudach, ale różnych częstotliwościach i fazach, a także odrobiny szumu.



Podczas przetwarzania szeregów czasowych (i innych rodzajów sekwencji, np. zdań) cechy wejściowe są zazwyczaj przedstawiane w postaci tablicy trójwymiarowej o wymiarach [rozmiar grupy, takty, wymiarowość], gdzie wartość wymiarowości wynosi 1 dla szeregów jednowymiarowych, a 2 i więcej dla szeregów wielowymiarowych.

Utwórzmy teraz za pomocą tej funkcji zbiory uczący, walidacyjny i testowy:

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

Zbiór `X_train` zawiera 7000 szeregów czasowych (jego wymiary to [7000, 50, 1]), z kolei `X_valid` ma 2000 szeregów czasowych (od szeregu 7000. do 8999.), natomiast `X_test` przechowuje 1000 szeregów czasowych (od 9000. do 9999.). Chcemy prognozować pojedynczą wartość w każdym szeregu czasowym, dlatego etykiety są wektorami kolumnowymi (`y_train` ma wymiary [7000, 1]).

Wskaźniki bazowe

Zanim zaczniemy korzystać z sieci rekurencyjnej, nieraz warto przygotować kilka wskaźników bazowych, gdyż może nam się wydawać, że model działa znakomicie, podczas gdy działa gorzej niż podstawowe modele. Na przykład najprostszym rozwiązaniem jest przewidywanie ostatniej wartości w każdym szeregu. Jest to tak zwane **prognozowanie naiwne** (ang. *naive forecasting*) i czasami osiąga ono tak dobre wyniki, że trudno je poprawić. W tym przypadku uzyskujemy błąd średniokwadratowy o wartości 0,02:

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Innym prostym rozwiązaniem jest wykorzystanie w pełni połączonej sieci. Oczekuje ona płaskiej listy cech dla każdego sygnału wejściowego, dlatego musimy dołączyć warstwę `Flatten`. Użyjmy prostego modelu regresji liniowej, dzięki czemu każda predykcja będzie stanowiła kombinację liniową wartości w szeregu czasowym:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

Po skompilowaniu tego modelu za pomocą funkcji straty MSE i domyślnego optymalizatora Adam spróbuj wytrenować go na zbiorze uczącym przez 20 epok i ocen go na zbiorze walidacyjnym. Powinnaś/powinieneś uzyskać wartość błędu MSE rzędu 0,004. To o wiele lepszy rezultat w porównaniu do podejścia naiwnego.

Implementacja prostej sieci rekurencyjnej

Zobaczmy, czy prosta sieć rekurencyjna jest w stanie osiągnąć jeszcze lepszy wynik:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Prostszej sieci rekurencyjnej nie da się stworzyć. Składa się ona z jednej jednoneuronowej warstwy, odzwierciedlającej schemat pokazany na rysunku 15.1. Nie musimy wyznaczać długości sekwencji wejściowych (czyli inaczej niż w poprzednim modelu), ponieważ rekurencyjna sieć neuronowa może przetwarzać dowolną liczbę taktów (z tego właśnie powodu wyznaczamy wartość `None` pierwszemu wymiarowi wejściowemu). Domyślnie warstwa `SimpleRNN` wykorzystuje funkcję tangensa hiperbolicznego. Mechanizm jej działania jest zgodny z naszym wcześniejszym opisem: stan początkowy $h_{(ini)}$ ma wartość 0 i jest przekazywany do pojedynczego neuronu rekurencyjnego wraz z wartością pierwszego taktu, $x_{(0)}$. Neuron oblicza sumę ważoną tych wartości i stosuje wobec niej funkcję tangensa hiperbolicznego, co pozwala uzyskać pierwszy sygnał wyjściowy, y_o . W prostej sieci rekurencyjnej ten rezultat staje się od razu nowym stanem h_o . Nowy stan zostaje przekazany na wejście tego samego neuronu wraz z następnym sygnałem wejściowym $x_{(1)}$ i proces ten jest powtarzany aż do ostatniego taktu. Na koniec zostaje wypuszczona na wyjście tylko ostatnia wartość, y_{49} . Operacje te są przeprowadzane równocześnie dla wszystkich szeregów czasowych.



Domyślnie warstwy rekurencyjne w interfejsie Keras zwracają wyłącznie ostateczny wynik. Jeżeli chcesz, aby był zwracany wynik z każdego taktu, musisz wyznaczyć parametr `return_sequences=True`.

Po skompilowaniu, wytrenowaniu i ocenieniu tego modelu (przez 20 epok i za pomocą optymalizatora Adam) okaże się, że błąd MSE wynosi 0,014, co jest wynikiem lepszym od uzyskanego w podejściu naiwnym, ale nie przebija rezultatu modelu liniowego. Zwróć uwagę, że dla każdego neuronu model liniowy ma po jednym parametrze na każdy sygnał wejściowy i takt, a także człon obciążenia (w użytym przez nas modelu liniowym mamy łącznie 51 parametrów). Z kolei w każdym neuronie rekurencyjnym tworzącym prostą sieć RNN przypada po jednym parametrze na wejście i na wymiar stanu ukrytego (w prostej sieci rekurencyjnej jest nim liczba neuronów rekurencyjnych tworzących warstwę), a dodatkowo występuje także człon obciążenia. Oznacza to, że w naszej prostej sieci występują tylko trzy parametry.

Trend i sezonowość

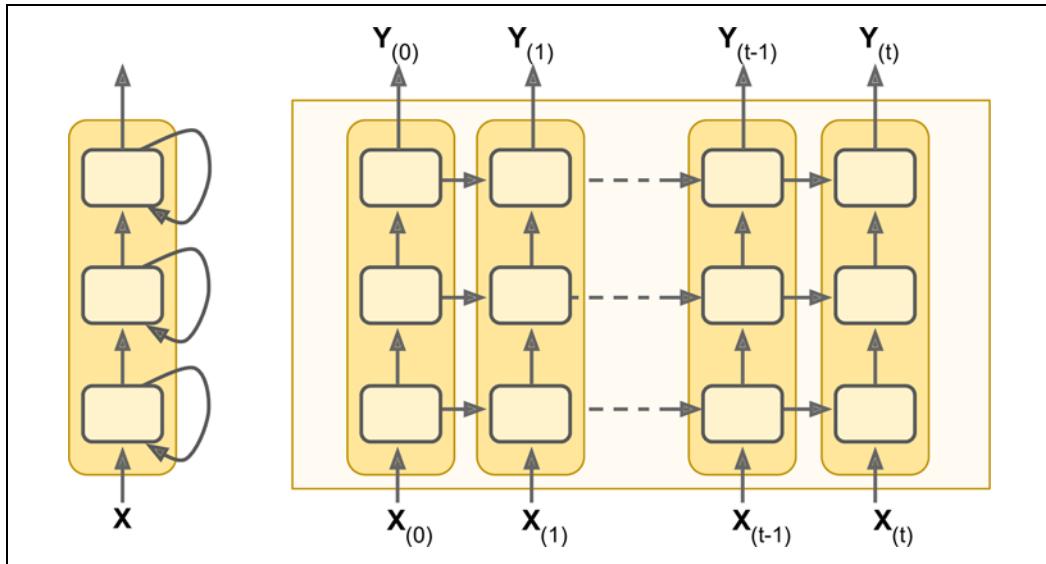
Istnieje wiele innych modeli prognozujących szeregi czasowe, jak choćby modele oparte na **ważonej średniej kroczącej** czy modele **ARIMA** (ang. *Autoregressive Integrated Moving Average* — autoregresyjna zintegrowana średnia krocząca). Przed użyciem niektórych z nich konieczne jest usunięcie trendu i sezonowości z danych. Jeżeli na przykład analizujesz liczbę użytkowników odwiedzających Twoją stronę i liczba ta wzrasta o 10% co miesiąc, to należy usunąć ten trend z szeregu czasowego. Po wykuczeniu modelu możesz z powrotem dodać ten trend, aby uzyskać prognozy ostateczne. Analogicznie, gdybyśmy chcieli przewidzieć miesięczną sprzedaż kremu przeciwskłonecznego, prawdopodobnie zauważlibyśmy silną sezonowość: największa sprzedaż przypada na lato, więc tego typu wzorzec będzie występował co roku. Należałyby usunąć taką sezonowość z szeregu czasowego, na przykład poprzez obliczenie różnicy pomiędzy wartością w każdym taktie a wartością z jednego roku wstecz (jest to tzw. **różnicowanie**). Również w tym przypadku po wykuczeniu modelu możesz z powrotem dodać składową sezonowości.

Zasadniczo nie musisz przeprowadzać tych czynności, gdy korzystasz z sieci rekurencyjnych, ale w pewnych przypadkach mogą one poprawić wydajność, ponieważ model nie będzie musiał uczyć się rozpoznawać trendu czy sezonowości.

Najwidoczniej nasza prosta sieć RNN okazała się zbyt prostą, aby mogła osiągnąć dobrą wydajność. Spróbujmy więc dodać więcej warstw rekurencyjnych!

Głębokie sieci rekurencyjne

Tworzenie wielowarstwowych sieci rekurencyjnych jest powszechnie spotykane (rysunek 15.7). Otrzymujemy w ten sposób głęboką sieć rekurencyjną.



Rysunek 15.7. Głęboka sieć rekurencyjna (po lewej) rozwijana w czasie (po prawej)

Implementowanie głębokiej sieci rekurencyjnej w interfejsie `tf.keras` jest stosunkowo proste: wystarczy ułożyć stos warstw rekurencyjnych. W poniższym przykładzie wykorzystamy trzy warstwy `SimpleRNN` (ale możemy wprowadzać dowolny typ warstw rekurencyjnych, taki jak warstwa LSTM czy GRU, które omówię już niebawem):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```



Pamiętaj o wyznaczeniu atrybutu `return_sequences=True` w każdej warstwie rekurencyjnej (oprócz ostatniej, jeżeli zależy Ci wyłącznie na sygnale wyjściowym). W przeciwnym wypadku będą one generowały tablicę dwuwymiarową (zawierającą wynik z ostatniego taktu) zamiast trójwymiarowej (przechowującą wyniki wszystkich taktów), a następna warstwa rekurencyjna będzie wyświetlać informację, że nie dostarczamy sekwencji w oczekiwany formacie trójwymiarowym.

Jeżeli skompilujesz, dopasujesz i ocenisz ten model, okaże się, że błąd średniokwadratowy osiąga wartość rzędu 0,003. W końcu udało nam się pokonać model liniowy!

Zwróć uwagę, że ostatnia warstwa nie jest idealna: zawiera tylko jedną jednostkę, ponieważ chcemy, aby był prognozowany jednowymiarowy szereg czasowy, a to oznacza konieczność wyznaczania pojedynczej wartości wyjściowej na każdy takt. Oznacza to jednak, że stan ukryty jest jedną wartością liczbową. Nie jest to wiele, a do tego okazuje się raczej niezbyt przydatne; prawdopodobnie sieć rekurencyjna będzie wykorzystywać stany ukryte z pozostałych warstw do przenoszenia wymaganych informacji pomiędzy poszczególnymi taktami i nie będzie często używać stanu ukrytego z warstwy końcowej. Do tego domyślną funkcją aktywacji w warstwie SimpleRNN jest tanh, dlatego prognozowane wartości muszą mieścić się w zakresie od -1 do 1. A gdybyśmy chcieli wprowadzić inną funkcję aktywacji? Z obydwu powyższych powodów zalecanym rozwiązaniem jest umieszczenie warstwy Dense na wyjściu sieci: model działałby nieco szybciej, dokładność pozostałaby mniej więcej taka sama i moglibyśmy wybrać dowolną funkcję aktywacji. Po wprowadzeniu takiej zmiany nie zapomnij usunąć atrybut return_sequences=True z drugiej (teraz ostatniej) warstwy rekurencyjnej:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

Po wytrenowaniu tego modelu przekonasz się, że uzyskuje on zbieżność nieco szybciej i osiąga bardzo zbliżone rezultaty w stosunku do poprzedniego modelu, a ponadto w razie potrzeby możesz zmienić funkcję aktywacji.

Prognozowanie kilka taktów w przód

Dotychczas prognozowaliśmy tylko wartość w następnym taktcie, ale również moglibyśmy przewidzieć wartość w jakimś dalszym taktcie poprzez zmodyfikowanie zmiennych docelowych w odpowiedni sposób (np. gdybyśmy chcieli uzyskać prognozy dla wartości w dziesiątym taktcie, wystarczyłoby tak zmienić zmienne docelowe, aby zawierały wartość dla dziesiątego taktu, a nie pierwszego). Ale może chcielibyśmy przewidzieć dziesięć kolejnych wartości?

Pierwszą możliwością jest wykorzystanie uprzednio przez nas wyuczonego modelu: prognozowanie za jego pomocą następnej wartości, przekazanie jej na wejście (tak jakby ta prognozowana wartość rzeczywiście wystąpiła) i na jej podstawie prognozowanie kolejnej wartości itd. Ilustruje to poniższy listing:

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

Jak można się spodziewać, prognozy dla następnego taktu są dokładniejsze dla dalszych predykcji z powodu nagromadzenia błędów (rysunek 15.8). Jeżeli ocenisz ten model na zestawie walidacyjnym, okaże się, że wartość MSE wynosi około 0,029. Jest to wartość znacznie większa niż w poprzednich modelach, ale również zadanie okazuje się o wiele trudniejsze, zatem porównywanie modeli nie ma



Rysunek 15.8. Prognozowanie dziesięciu taktów w czasie po jednym

tu większego sensu. Znacznie rozsądniej porównać wydajność modelu z prognozami naiwnymi (prognozowaniem, że szereg czasowy pozostanie niezmienny przez dziesięć taktów) lub z prostym modelem liniowym. Prognozy naiwne są okropne (mają wartość MSE zbliżone do 0,223), ale w modelu liniowym błąd średniokwadratowy jest równy 0,0188, co jest wartością o wiele lepszą od uzyskanej przez sieć rekurencyjną przewidującą przyszłość po jednym taktie, a jednocześnie model liniowy jest zdecydowanie szybszy, zarówno podczas uczenia, jak i wnioskowania. Mimo to, jeżeli zależy Ci wyłącznie na prognozowaniu kilku taktów w przód w bardziej skomplikowanych zadaniach, rozwiązanie to może spisywać się całkiem nieźle.

Drugim rozwiązaniem jest wyuczenie sieci rekurencyjnej do prognozowania dziesięciu następnych wartości naraz. Możemy nadal korzystać z modelu sekwencyjno-wektorowego, ale będzie on teraz generował na wyjściu dziesięć wartości zamiast jednej. Musimy jednak zmodyfikować najpierw zmienne docelowe tak, aby były wektorami przechowującymi dziesięć następnych wartości:

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Teraz wystarczy utworzyć warstwę wyjściową zawierającą dziesięć jednostek:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

Po wytrenowaniu modelu możemy z łatwością prognozować dziesięć kolejnych wartości naraz:

```
Y_pred = model.predict(X_new)
```

Model ten spisuje się całkiem dobrze: błąd MSE dla następnych dziesięciu taktów wynosi około 0,008. Jest to wartość o wiele lepsza od uzyskanej przez model liniowy. Nadal jednak możemy uzyskać jeszcze lepsze rezultaty — zamiast uczyć model prognozowania dziesięciu następnych wartości wyłącznie w ostatnim taktie możemy zrobić to samo dla wszystkich taktów. Inaczej mówiąc, możemy przekształcić tę sieć sekwencyjno-wektorową w sieć sekwencyjną. Zaleta tej techniki jest taka, że funkcja straty będzie zawierała człon dla wyniku sieci w każdym taktie, nie tylko w ostatnim. Oznacza to, że przez model będzie przepływać znacznie więcej gradientów błędów, które nie będą rozprowadzane wyłącznie w czasie — ich propagacja będzie następować również od wyjścia w każdym taktie. Mechanizm ten jednocześnie ustabilizuje i przyspieszy proces uczenia.

Gwoli jasności, w taktie 0. model będzie generował wektor zawierający prognozy dla taktów od 1. do 10., następnie w taktie 1. będą prognozowane taki od 2. do 11. itd. Każda zmienna docelowa musi więc być sekwencją o takiej samej długości jak sekwencja wejściowa, będąca w każdym taktie dziesięciowymiarowym wektorem. Przygotujmy takie sekwencje docelowe:

```
Y = np.empty((10000, n_steps, 10)) # Każda zmienna docelowa jest sekwencją wektorów
# dziesięciowymiarowych
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] = series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```



Może być zaskakujące, że zmienne docelowe będą przechowywały wartości pojawiające się na wejściach (zmienne X_train i Y_train mają wiele elementów wspólnych). Czy nie jest to oszustwem? Na szczęście nie. W każdym taktie obiekt zna tylko wcześniejsze taki, zatem nie może „zerkać” w przód. Jest to tak zwany **model przyczynowy** (ang. *causal model*).

Aby przekształcić ten model do postaci sekwencyjnej, musimy wyznaczyć atrybut return_sequences=True we wszystkich warstwach rekurencyjnych (oprócz ostatniej) i wprowadzić wyjściową warstwę Dense w każdym taktie. Interfejs Keras zawiera zdefiniowaną w tym właśnie celu warstwę TimeDistributed: obejmuje ona dowolną warstwę (np. warstwę Dense) i stosuje ją w każdym taktie sekwencji wejściowej. Skuteczność tego rozwiązania zostaje uzyskana poprzez taką modyfikację wymiarów sygnałów wejściowych, że każdy takt jest traktowany jako osobny przykład (tzn. dane wejściowe znajdują przekształcone z [rozmiar grupy, taki, wymiary wejściowe] w [rozmiar grupy × taki, wymiary wejściowe]; w naszym przykładzie liczba wymiarów wejściowych wynosi 20, ponieważ poprzednia warstwa SimpleRNN zawierała 20 jednostek), następnie zostaje zastosowana warstwa Dense, a na koniec dane wyjściowe znajdują przekształcone z powrotem w sekwencje (tzn. wymiary [rozmiar grupy × taki, wymiary wyjściowe] znajdują zmodyfikowane na postać [rozmiar grupy, taki, wymiary wyjściowe]; w tym przykładzie liczba wymiarów wyjściowych wynosi 10, gdyż warstwa Dense zawiera 10 jednostek)². Oto zaktualizowany model:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

² Zwróć uwagę, że warstwa TimeDistributed(Dense(n)) jest równoważna warstwie Conv1D(n, filter_size=1).

Warstwa Dense w rzeczywistości obsługuje sekwencje jako dane wejściowe (a nawet bardziej wielowymiarowe sygnały wejściowe) — traktuje je tak samo jak warstwa TimeDistributed(Dense(...)), co oznacza, że jest stosowana wyłącznie do ostatniego wymiaru wejściowego (niezależnie we wszystkich taktach). Możemy więc zastąpić ostatnią warstwę warstwą Dense(10), ale dla zachowania przejrzystości pozostawimy warstwę TimeDistributed(Dense(10)), ponieważ w ten sposób widać wyraźnie, że warstwa Dense jest stosowana niezależnie w każdym taktie, a model będzie generował sekwencję, a nie pojedynczy wektor.

Wszystkie dane wyjściowe są potrzebne w czasie uczenia, ale tylko wynik z ostatniego taktu przydaje się do uzyskiwania prognoz i oceny modelu. Zatem mimo tego, że podczas trenowania sieci będziemy polegać na funkcji MSE ze wszystkich wyjść, do oceny modelu wykorzystamy niestandardowy wskaźnik, obliczający wyłącznie błąd średniokwadratowy dla wyjścia w ostatnim taktie:

```
def last_time_step_mse(Y_true, Y_pred):  
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])  
  
optimizer = keras.optimizers.Adam(lr=0.01)  
model.compile(loss="mse", optimizer=optimizer, metrics=[last_time_step_mse])
```

Osiągamy wartość błędu średniokwadratowego dla zbioru walidacyjnego rzędu 0,006, co oznacza poprawę o $\frac{1}{4}$ w stosunku do poprzedniego modelu. Możesz połączyć to rozwiązanie z poprzednim: prognozuj dziesięć pierwszych wartości za pomocą tej sieci rekurencyjnej, następnie dodaj te wartości do wejściowego szeregu czasowego i wykorzystaj ponownie model do prognozowania dziesięciu kolejnych wartości — powtarzaj ten proces dowolną liczbę razy. W ten sposób możesz generować sekwencje o dowolnej długości. Rozwiązanie to może nie jest bardzo dokładne w przypadku długich sekwencji, ale powinno w zupełności wystarczyć, jeżeli Twoim zadaniem jest generowanie muzyki lub tekstu, o czym przekonasz się w rozdziale 16.



Podczas prognozowania szeregów czasowych warto nieraz wprowadzić jakieś wskaźniki błędów. Dobrym rozwiązaniem jest wprowadzenie warstwy porzucania MC, która została omówiona w rozdziale 11.: dodaj warstwę porzucania MC w każdej komórce pamięci, dzięki czemu porzucane będą niektóre wejścia i stany ukryte. Po wyuczeniu modelu wykorzystaj wielokrotnie model do prognozowania nowego szeregu czasowego, a następnie oblicz średnią i odchylenie standardowe prognoz w każdym taktie.

Proste sieci rekurencyjne mogą radzić sobie całkiem nieźle z prognozowaniem szeregów czasowych lub przetwarzaniem innych sekwencji pod warunkiem, że te szeregi czasowe czy sekwencje są krótkie. Zastanówmy się, dlaczego tak jest i co możemy z tym zrobić.

Obsługa długich sekwencji

Aby wytrenować sieć rekurencyjną na długich sekwencjach, musimy przetwarzać wiele taktów, przez co rozwinięta sieć RNN staje się bardzo głęboka. Podobnie jak w przypadku innych struktur uczenia głębokiego jest ona narażona na omówiony w rozdziale 11. problem niestabilnych gradientów: uczenie sieci może zająć wieczność albo być bardzo niestabilne. Do tego w miarę przetwarzania długich sekwencji sieć rekurencyjna stopniowo „zapomina” pierwsze sygnały. Przyjrzymy się obydwu tym problemom, począwszy od kwestii niestabilnych gradientów.

Zwalczanie problemu niestabilnych gradientów

W sieciach rekurencyjnych spotykamy wiele sztuczek wykorzystywanych w innych sieciach głębkich do zredukowania problemu niestabilnych gradientów: odpowiednie inicjalizowanie parametrów, szybsze optymalizatory, porzucanie itd. Jednak nienasycające funkcje aktywacji (np. ReLU) raczej nie zdadzą się tu na wiele — w rzeczywistości mogą one wręcz zmniejszać stabilność sieci RNN w czasie uczenia. Dlaczego? Założymy, że algorytm gradientu prostego aktualizuje wagi w sposób zwiększający nieznacznie wartość sygnałów wyjściowych w pierwszym taktcie. W każdym taktcie są wykorzystywane te same wagi, zatem sygnały wyjściowe w drugim taktcie również zostaną nieco wzmacnione, podobnie jak w taktie trzecim itd., aż do „eksplozji” danych wyjściowych. Funkcja nienasycająca nie jest niestety w stanie temu zapobiec. Możesz zredukować to zagrożenie poprzez wprowadzenie mniejszego współczynnika uczenia, ale równie dobrze możesz wprowadzić nasycającą funkcję aktywacji, jak na przykład tanh (dlatego właśnie jest ona funkcją domyślną). Analogicznie mogą eksplodować również gradienty. Jeżeli stwierdzisz, że proces uczenia jest niestabilny, sprawdź wartości gradientów (np. za pomocą narzędzia TensorBoard) i w razie potrzeby spróbuj zastosować technikę obcinania gradientów.

Do tego technika normalizacji wsadowej nie jest tak skuteczna w sieciach rekurencyjnych jak w głębiokich strukturach jednokierunkowych. W rzeczywistości nie możesz jej używać pomiędzy taktami, lecz wyłącznie pomiędzy warstwami rekurencyjnymi. Mówiąc dokładniej, istnieje możliwość dołączenia warstwy BN do komórki pamięci (o czym przekonasz się już wkrótce), dzięki czemu może być stosowana w każdym taktcie (zarówno wobec sygnałów wejściowych w tym taktcie, jak i stanu ukrytego z taktu poprzedniego). Jednak w każdym taktcie będzie używana ta sama warstwa normalizacji wsadowej, z tymi samymi parametrami, bez względu na rzeczywistą skalę i przesunięcie danych wejściowych oraz stanu ukrytego. W praktyce rozwiązywanie to nie daje zbyt dobrych rezultatów, co udowodnili César Laurent i in. w publikacji z 2015 roku (<https://arxiv.org/abs/1510.01378>)³: autorzy zauważyl, że normalizacja wsadowa daje niewielkie korzyści jedynie wtedy, gdy są nią traktowane jedynie wejścia, a nie stany ukryte. Inaczej mówiąc, nieznaczne usprawnienie modelu następuje wówczas, kiedy normalizacja wsadowa jest przeprowadzana pomiędzy warstwami rekurencyjnymi (pionowo na rysunku 15.7), ale już nie w ich wnętrzu (poziomo na rysunku 15.7). W interfejsie Keras możemy to z łatwością zrobić poprzez dodanie warstwy `BatchNormalization` przed każdą warstwą rekurencyjną, nie oczekuj jednak żadnych cudów.

Lepsze rezultaty w sieciach rekurencyjnych uzyskujemy za pomocą innego typu normalizacji: **normalizacji warstwowej** (ang. *layer normalization*). Koncepcję tę zaproponowali Lei Ba i in. w artykule z 2016 roku (<https://arxiv.org/abs/1607.06450>)⁴: rozwiązywanie to bardzo przypomina normalizację wsadową, tutaj jednak normalizacja nie jest przeprowadzana wzduż osi grup, lecz wzduż osi cech. Jedną z zalet tej techniki jest możliwość obliczania wymaganych statystyk na bieżąco w każdym taktcie, niezależnie od przykładów. Oznacza to także, że w przeciwieństwie do normalizacji wsadowej rozwiązywanie to działa tak samo w czasie nauki i testowania oraz nie jest tu potrzebne obliczanie wykładniczych średnich kroczących do oszacowania statystyk cechy dla wszystkich przykładów

³ César Laurent i in., *Batch Normalized Recurrent Neural Networks*, „Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing” (2016), s. 2657 – 2661.

⁴ Jimmy Lei Ba i in., *Layer Normalization*, arXiv preprint arXiv:1607.06450 (2016).

w zbiorze uczącym. Z kolei podobnie jak w normalizacji wsadowej, normalizacja warstwowa określa parametry skali i przesunięcia każdego wejścia. W sieciach rekurencyjnych technika ta jest najczęściej wykorzystywana tu po kombinacji liniowej wejść i stanów ukrytych.

Wykorzystajmy interfejs tf.keras do zaimplementowania normalizacji warstwowej w prostej komórce pamięci. W tym celu musimy zdefiniować niestandardową komórkę pamięci. Będzie ona przypominać standardową warstwę, z tym że jej metoda call() będzie przyjmować dwa argumenty: dane wejściowe (inputs) w bieżącym takcie i stany ukryte (hidden) z taktu poprzedniego. Zwróć uwagę, że argument states jest listą zawierającą co najmniej jeden tensor. W przypadku prostej komórki rekurencyjnej mieści się w niej jeden tensor równoważny wyjściom z poprzedniego taktu, ale inne komórki mogą zawierać wiele tensorów stanów (np. wkrótce przekonasz się, że komórka LSTMCell zawiera stany długotrwały i krótkotrwały). Komórka musi zawierać również atrybuty state_size i output_size. W prostej sieci rekurencyjnej są one równe liczbie jednostek. Poniższy listing implementuje niestandardową komórkę pamięci, działającą tak samo jak warstwa SimpleRNNCell, z tym że w każdym taktie będzie realizowała również normalizację warstwową:

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)

    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Kod ten nie jest zbyt skomplikowany⁵. Klasa LNSimpleRNNCell dziedziczy z klasy keras.layers.Layer, podobnie jak wszystkie niestandardowe warstwy. Konstruktor przyjmuje liczbę jednostek i wymaganą funkcję aktywacji oraz wyznacza atrybuty state_size i output_size, po czym tworzy komórkę SimpleRNNCell niezawierającą funkcji aktywacji (gdyż chcemy przeprowadzić normalizację warstwową po operacji liniowej, ale przed zastosowaniem funkcji aktywacji). Następnie konstruktor ten tworzy warstwę LayerNormalization i na koniec dostarcza wybraną funkcję aktywacji. Metoda call() najpierw wprowadza prostą komórkę rekurencyjną, która oblicza kombinację liniową bieżących wejść i poprzednich stanów ukrytych, a potem dwukrotnie zwraca wynik (w komórce SimpleRNNCell dane wyjściowe są równe stanom ukrytym — inaczej mówiąc, new_states[0] jest równe outputs, zatem możemy spokojnie ignorować new_states w dalszej części metody call()). Następnie metoda call() stosuje normalizację warstwową, a po niej funkcję aktywacji. Na koniec dwukrotnie są zwierane wyniki (raz jako dane wyjściowe i raz jako stany ukryte). Aby wykorzystać tę niestandardową komórkę, wystarczy teraz utworzyć warstwę keras.layers.RNN i przekazać jej instancję komórki:

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                     input_shape=[None, 1]),
```

⁵ Byłoby łatwiej dziedziczyć z klasy SimpleRNNCell, dzięki czemu nie musielibyśmy tworzyć wewnętrznej komórki SimpleRNNCell lub zajmować się atrybutami state_size i output_size, jednak w tym przykładzie celem jest pokazanie mechanizmu tworzenia niestandardowej komórki od podstaw.

```
        keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
        keras.layers.TimeDistributed(keras.layers.Dense(10))
    ])
```

Analogicznie moglibyśmy utworzyć niestandardową komórkę, która przeprowadza porzucanie pomiędzy poszczególnymi taktami. Istnieje jednak prostszy sposób: wszystkie warstwy rekurencyjne (oprócz `keras.layers.RNN`) i wszystkie komórki występujące w interfejsie Keras zawierają hiperparametry `dropout` i `recurrent_dropout` — pierwszy definiuje współczynnik porzucania stosowany wobec wejść (w każdym taktie), natomiast drugi hiperparametr wyznacza współczynnik porzucania dla stanów ukrytych (także w każdym taktie). Nie musimy tworzyć niestandardowej komórki, aby realizować porzucanie w każdym taktie sieci rekurencyjnej.

Dzięki tym technikom możesz zmniejszyć ryzyko niestabilnych gradientów i znacznie skuteczniej uczyć się rekurencyjną. Zobaczmy teraz, jak możemy poradzić sobie z problemem pamięci krótkotrwałej.

Zwalczanie problemu pamięci krótkotrwałej

W wyniku przekształceń danych zachodzących podczas ich przepływu przez sieć w każdym taktie następuje utrata części informacji. Po pewnym czasie stan sieci RNN nie zawiera praktycznie żadnych śladów danych wejściowych. Może to oznaczać fiasko całego modelu. Wyobraź sobie rybkę Dory⁶, która próbuje przetłumaczyć długie zdanie — zanim doczytałaby je do końca, zapomniałaby, jaki był jego początek. Receptą na ten problem są zaprojektowane różne typy komórek pamięci długotrwałej. Okazały się one tak skuteczne, że od czasu ich wprowadzenia podstawowe komórki rekurencyjne praktycznie przestały być używane. Zajmijmy się najpierw najpopularniejszą odmianą komórek pamięci: komórkami LSTM.

Komórki LSTM

Komórka **LSTM** (ang. *Long Short-Term Memory* — długa pamięć krótkotrwała) została zaproponowana w 1997 roku (<https://hml.info/93>)⁷ przez Seppa Hochreitera i Jürgena Schmidhubera, a od tamtego czasu ulepszało jej konstrukcję wielu badaczy, takich jak Alex Graves (<https://www.cs.toronto.edu/~graves/>), Haşim Sak (<https://arxiv.org/abs/1402.1128>)⁸ i Wojciech Zaremba (<https://arxiv.org/abs/1409.2329>)⁹. Jeżeli wyobrażymy sobie komórkę LSTM jako czarną skrzynkę, okaże się, że przypomina ona bardzo komórkę podstawową, z tym że znacznie wydajniejszą; model szybciej uzyskuje zbieżność oraz jest w stanie wykrywać w danych długoterminowe zależności. W interfejsie Keras wystarczy zastąpić warstwę `SimpleRNNCell` warstwą LSTM:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

⁶ Postać z filmów animowanych „Gdzie jest Nemo?” i „Gdzie jest Dory?” cierpiąca na zaniki pamięci krótkotrwałej.

⁷ Sepp Hochreiter i Jurgen Schmidhuber, *Long Short-Term Memory*, „Neural Computation” 9, no. 8 (1997), s. 1735 – 1780.

⁸ Haşim Sak i in., *Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*, arXiv preprint arXiv:1402.1128 (2014).

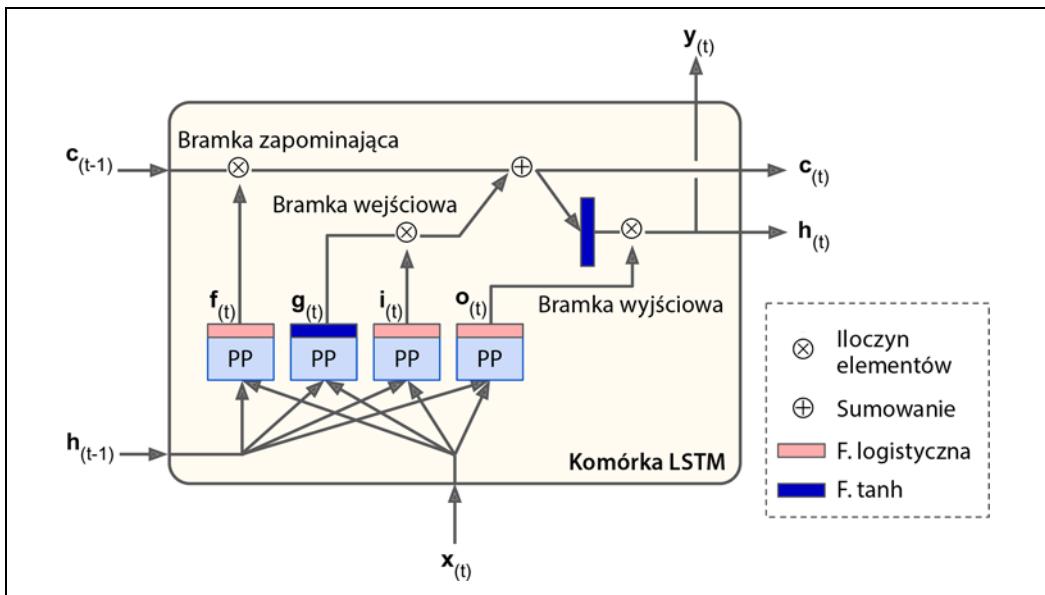
⁹ Wojciech Zaremba i in., *Recurrent Neural Network Regularization*, arXiv preprint arXiv:1409.2329 (2014).

Ewentualnie możesz skorzystać z ogólnej warstwy `keras.layers.RNN`, której podajemy `LSTMCell` jako argument:

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True,
                     input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Warstwa LSTM wykorzystuje jednak zoptymalizowaną implementację, gdy jest przetwarzana przez procesory graficzne (zob. rozdział 19.), dlatego zasadniczo jest ona zalecana (warstwa RNN przydaje się najczęściej podczas definiowania niestandardowych komórek).

Zatem jak działa komórka LSTM? Jej struktura została pokazana na rysunku 15.9.



Rysunek 15.9. Komórka LSTM

Jeżeli nie zajrzymy do skrzynki, komórka LSTM będzie wyglądać tak samo jak komórka podstawowa. Jedyną różnicą jest rozdzielenie jej pomiędzy dwa wektory: $h_{(t)}$ i $c_{(t)}$ (litera „c” pochodzi od wyrazu *cell*, czyli „komórka”). Wektor $h_{(t)}$ możemy interpretować jako stan krótkotrwały, a $c_{(t)}$ — długotrwały.

Otwórzmy w końcu skrzynkę! Podstawowa koncepcja polega na tym, że sieć uczy się, co ma zapamiętywać w stanie długotrwałym, co odrzucać oraz co z niego odczytywać. Stan długotrwały $c_{(t-1)}$ porusza się w sieci w prawo, widzimy zatem, że najpierw przechodzi przez **bramkę zapominającą** (ang. *forget gate*), porzuca trochę wspomnień, a następnie dodaje nowe za pomocą operacji sumowania (dodawane są wspomnienia wyselekcjonowane poprzez **bramkę wejściową** — ang. *input gate*). Wynikowy stan $c_{(t)}$ jest przesyłany dalej bez wprowadzania jakichkolwiek przekształceń. Zatem w każdym taktie pewne wspomnienia są usuwane, a inne dodawane. Ponadto po operacji sumowania stan długotrwały zostaje skopiowany i przepuszczony przez funkcję tangensa hiperbolicznego, po czym uzyskany rezultat jest filtrowany przez **bramkę wyjściową** (ang. *output gate*). Uzyskujemy

w ten sposób stan krótkotrwały $\mathbf{h}_{(t)}$ (który jest równy wynikowi na wyjściu danej komórki w takcie $y_{(t)}$). Przyjrzyjmy się teraz, skąd się biorą nowe wspomnienia oraz w jaki sposób działają bramki.

Najpierw do czterech różnych, w pełni połączonych warstw (na rysunku zostały oznaczone jako PP — pełne połączenie) zostają przekazane bieżący wektor wejściowy $\mathbf{x}_{(t)}$ i poprzedni krótkotrwały wektor stanu $\mathbf{h}_{(t-1)}$. Każda z tych warstw pełni inną funkcję:

- Główna warstwa generuje wartości wektora $\mathbf{g}_{(t)}$. Jej zadaniem jest analizowanie bieżących danych wejściowych $\mathbf{x}_{(t)}$ i poprzedniego (krótkotrwałego) stanu $\mathbf{h}_{(t-1)}$. W komórce podstawowej występuje tylko ta warstwa, a jej wynik jest przekazywany do wektorów $\mathbf{y}_{(t)}$ i $\mathbf{h}_{(t)}$. Z kolei w komórce LSTM wynik uzyskiwany w tej warstwie nie jest wypuszczany dalej, lecz jego najistotniejsze elementy są przechowywane w stanie długotrwałym (pozostałe elementy zostają porzucone).
- Trzy pozostałe warstwy są **kontrolerami bramek** (ang. *gate controllers*). Korzystają one z logistycznej funkcji aktywacji, dlatego ich wyniki mieszczą się w zakresie od 0 do 1. Jak widać, ich wyniki przechodzą przez operacje iloczynu elementowego, więc w przypadku wartości 0 bramka zostaje zamknięta, a wartość 1 ją otwiera. A dokładniej:
 - **Bramka zapominająca** (sterowana przez wektor $f_{(t)}$) określa, które składniki pamięci długotrwałej powinny zostać usunięte.
 - **Bramka wejściowa** (sterowana przez wektor $i_{(t)}$) określa, które elementy wektora $\mathbf{g}_{(t)}$ powinny być dodawane do stanu długotrwałego.
 - Na koniec **bramka wyjściowa** (sterowana przez wektor $o_{(t)}$) określa, które składowe stanu długotrwałego powinny być odczytywane i wysyłane na wyjście w danym takcie (zarówno do wektora $\mathbf{h}_{(t)}$, jak i do $\mathbf{y}_{(t)}$).

Mówiąc krótko, komórka LSTM jest w stanie nauczyć się rozpoznawać ważne dane na wejściu (zadanie bramki wejściowej), przechowywać je w stanie długotrwałym, uczyć się przechowywać je dopóty, dopóki są potrzebne (rola bramki zapominającej), a także wydobywać je wtedy, gdy będzie trzeba. Dlatego właśnie komórki tego typu są tak zdumiewająco skuteczne w wychwytywaniu długoterminowych wzorców w szeregach czasowych, długich tekstach, nagraniach dźwiękowych itd.

Równanie 15.3 stanowi podsumowanie operacji wyliczania stanu długotrwałego, stanu krótkotrwałego oraz wyjść w każdym takcie dla danej próbki (wzory dla całej minigrupy są bardzo podobne).

Równanie 15.3. Obliczenia komórki LSTM

$$\begin{aligned}\mathbf{i}_{(t)} &= \sigma\left(\mathbf{W}_{xi}^T \mathbf{x}_{(t)} + \mathbf{W}_{hi}^T \mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \\ \mathbf{f}_{(t)} &= \sigma\left(\mathbf{W}_{xf}^T \mathbf{x}_{(t)} + \mathbf{W}_{hf}^T \mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \\ \mathbf{o}_{(t)} &= \sigma\left(\mathbf{W}_{xo}^T \mathbf{x}_{(t)} + \mathbf{W}_{ho}^T \mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \\ \mathbf{c}_{(t)} &= \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \\ \mathbf{y}_{(t)} &= \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)})\end{aligned}$$

W tym równaniu:

- W_{xi} , W_{xf} , W_{xo} , W_{xg} to macierze wag każdej z czterech warstw dla ich połączenia z wektorem wejściowym $\mathbf{x}_{(t)}$.
- W_{hi} , W_{hf} , W_{ho} , W_{hg} są macierzami wag każdej z czterech warstw dla ich połączenia z poprzednim stanem krótkotrwałym $\mathbf{h}_{(t-1)}$.
- b_b , b_f , b_o , b_g to członły obciążenia każdej z czterech warstw. Zwróć uwagę, że moduł TensorFlow inicjalizuje wektor \mathbf{b}_f wypełniony jedynkami, a nie zerami. Zapobiegamy w ten sposób zapominaniu wszystkiego na początku uczenia.

Połączenia przeziorne

W podstawowej komórce LSTM kontrolery bramek mogą obserwować jedynie tensor wejściowy $\mathbf{x}_{(t)}$ i poprzedni stan krótkotrwały $\mathbf{h}_{(t-1)}$. Korzystne mogłyby jednak być także umożliwienie sprawdzania stanu długotrwałego w celu uzyskania szerszego kontekstu. Pomyśl ten został zaproponowany przez Feliksa Gersa i Jürgena Schmidhubera w 2000 roku (<https://homl.info/96>)¹⁰. Przedstawili oni koncepcję komórki zawierającej tak zwane **połączenia przeziorne** (ang. *peephole connections*): do wejść kontrolerów bramki zapominającej i bramki wejściowej dołączany jest poprzedni stan długotrwały $\mathbf{c}_{(t-1)}$, natomiast na wejściu bramki wyjściowej zostaje dodany bieżący stan długotrwały $\mathbf{c}_{(t)}$. Rozwiążanie to często (ale nie zawsze) poprawia wydajność i nie potrafimy przewidzieć, w których zadaniach może się ono przydać, a w których nie, musisz więc samodzielnie przetestować jego działanie w danym zadaniu.

W module Keras warstwa LSTM bazuje na komórce keras.layers.LSTMCell, która nie obsługuje połączeń przeziorowych. Ich obsługę zawiera natomiast eksperymentalna komórka tf.keras.experimental.PeepholeLSTMCell, możesz więc utworzyć warstwę keras.layers.RNN i przekazać jej konstruktorowi komórkę PeepholeLSTMCell.

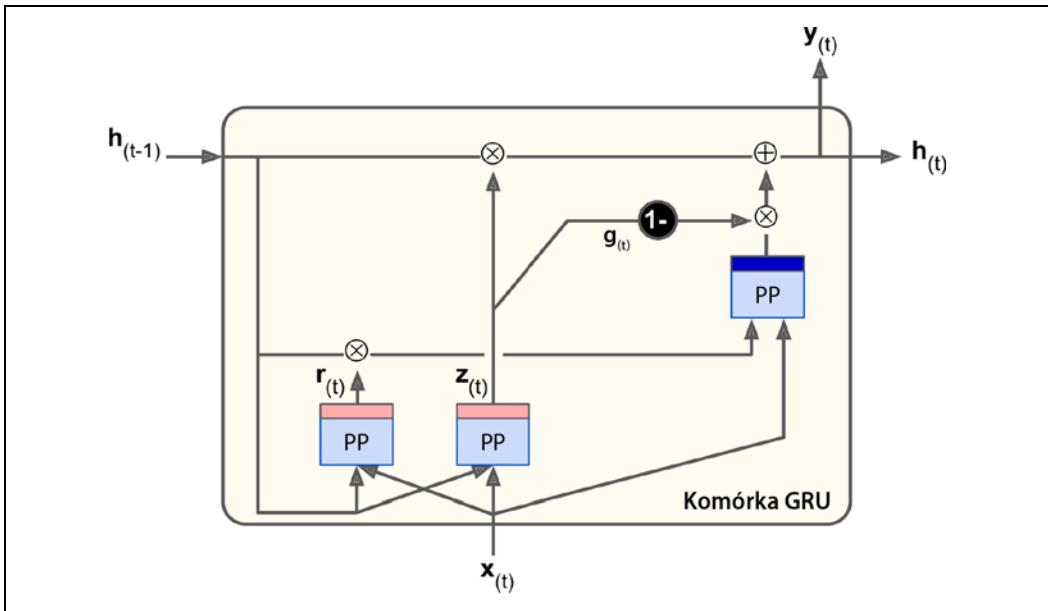
Istnieje wiele innych odmian komórki LSTM. Szczególną popularnością cieszy się komórka GRU, której przyjrzymy się już za chwilę.

Komórki GRU

Komórka GRU (ang. *Gated Recurrent Unit* — bramkowana jednostka rekurencyjna) (rysunek 15.10) została po raz pierwszy opisana w 2014 roku przez Kyunghuna Cho i in. w dokumencie (<https://arxiv.org/abs/1406.1078>)¹¹ opisującym również wspomnianą wcześniej sieć typu koder – dekoder.

¹⁰ F. A. Gers i J. Schmidhuber, *Recurrent Nets That Time and Count*, „Proceedings of the IEEE INNS-ENNS International Joint Conference on Neural Networks” (2000), s. 189 – 194.

¹¹ Kyunghyun Cho i in., *Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation*, „Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing” (2014), s. 1724 – 1734.



Rysunek 15.10. Komórka GRU

GRU stanowi uproszczoną wersję komórki LSTM i osiąga zbliżoną wydajność¹² (stąd jej rosnąca popularność). Główne uproszczenia architektury są następujące:

- Obydwa wektory stanów zostają scalone w jeden wektor $h_{(t)}$.
- Bramki zapominająca i wejściowa są sterowane przez wspólny kontroler $z_{(t)}$. Jeżeli na wyjściu kontrolera pojawia się wartość 1, to bramka wejściowa zostaje otwarta (= 1), a zapominająca — zamknięta ($1 - 1 = 0$). W przypadku wartości 0 występuje przeciwna sytuacja. Innymi słowy, za każdym razem, gdy wspomnienia mają zostać zachowane, miejsce na nie przeznaczone zostaje najpierw wyczyszczone. W rzeczywistości to rozwiązywanie samo w sobie stanowi osobny wariant komórki LSTM.
- Nie ma bramki wyjściowej; w każdym takcie na wyjściu pojawia się pełny wektor stanu. Dla równowagi jednak zostaje umieszczony dodatkowy kontroler $r_{(t)}$, określający, która część poprzedniego stanu zostanie ukazana w warstwie głównej ($g_{(t)}$).

W równaniu 15.4 przedstawiam wzory służące do obliczania stanu komórki w każdym taktie dla pojedynczego przykładu.

¹² Klaus Greff i in. w publikacji z 2015 roku, *LSTM: A Search Space Odyssey* (<https://arxiv.org/abs/1503.04069>), udowadniają, że wydajność wszystkich wariantów komórki LSTM jest mniej więcej taka sama.

Równanie 15.4. Obliczenia komórki GRU

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\ \mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}\end{aligned}$$

Interfejs Keras zawiera warstwę keras.layers.GRU (opartą na komórce pamięci keras.layers.GRUCell) — jej wstawienie to kwestia podmienienia warstwy SimpleRNN na warstwę LSTM zawierającą komórki GRU.

Komórki LSTM i GRU stanowią jedne z głównych powodów sukcesu sieci rekurencyjnych. Jednak mimo że umożliwiają przetwarzanie znacznie dłuższych sekwencji w porównaniu do prostej sieci RNN, ich pamięć krótkotrwała ciągle pozostaje dość ograniczona i duży problem sprawia im wykrywanie długoterminowych wzorców w co najmniej stutaktowych sekwencjach, takich jak próbki dźwiękowe, długie szeregi czasowe lub długie zdania. Jednym z rozwiązań okazuje się skrócenie sekwencji wejściowych, na przykład za pomocą jednowymiarowych warstw splotowych.

Wykorzystywanie jednowymiarowych warstw splotowych do przetwarzania sekwencji

Z rozdziału 14. wiesz, że mechanizm działania dwuwymiarowej warstwy splotowej polega na przesuwaniu kilku dość małych jąder (filtrów) wzdłuż obrazu, co pozwala uzyskać dwuwymiarowe mapy cech (po jednej na jądro). W podobny sposób jednowymiarowa warstwa splotowa przesywa kilka jąder wzdłuż sekwencji i generuje po jednej jednowymiarowej mapie cech na każde jądro. Każdy filtr uczy się wykrywać jeden bardzo krótki wzorzec sekwencyjny (nie dłuższy od rozmiaru jądra). Gdybyśmy korzystali z dziesięciu jąder, to wynik warstwy składałby się z dziesięciu sekwencji jednowymiarowych (wszystkie miałyby taką samą długość); ewentualnie można byłoby traktować taki wynik jako pojedynczą dziesięciowymiarową sekwencję. Oznacza to, że możemy zbudować sieć neuronową składającą się z mieszanki warstw rekurencyjnych i jednowymiarowych warstw splotowych (a nawet jednowymiarowych warstw łączących). Jeżeli korzystasz z warstwy splotowej o kroku 1 i uzupełnianiu zerami typu "same", to sekwencja wyjściowa będzie miała taki sam rozmiar jak sekwencja wejściowa. Jeżeli jednak wyznaczysz uzupełnianie zerami typu "valid" lub krok większy niż 1, to sekwencja wyjściowa będzie krótsza od sekwencji wejściowej, dlatego musisz odpowiednio dostosować zmienne docelowe. Na przykład poniższy model jest niemal taki sam jak poprzedni, tutaj jednak na początku występuje jednowymiarowa warstwa splotowa, która zmniejsza sekwencję wejściową o 100% za pomocą kroku równego 2. Rozmiar jądra jest większy od kroku, więc do obliczenia wyniku warstwy wyjściowej posłużą wszystkie dane wejściowe, a model nauczy się przechowywać użyteczne informacje oraz porzucać nieprzydatne szczegóły. Dzięki skracaniu sekwencji warstwa splotowa pomaga niejako warstwom GRU wykrywać dłuższe wzorce. Zwróć uwagę, że musimy także usunąć trzy pierwsze takty w zmiennych docelowych (rozmiar jądra wynosi 4, dlatego pierwszy wynik warstwy splotowej będzie bazował na wejściowych taktach od zerowego do trzeciego), a także zmniejszyć dwukrotnie zmienne docelowe:

```

model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

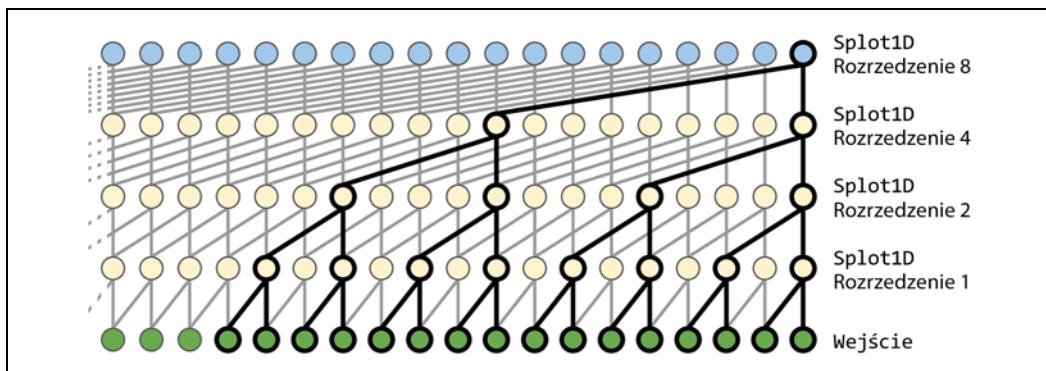
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                      validation_data=(X_valid, Y_valid[:, 3::2]))

```

Po wyuczeniu i ocenieniu tego modelu okaże się, że jest on jak na razie najlepszy. Warstwa splotowa naprawdę dużo tu daje. W rzeczywistości można nawet korzystać wyłącznie z jednowymiarowych warstw splotowych i całkowicie zrezygnować z warstw rekurencyjnych!

WaveNet

W publikacji z 2016 roku (<https://arxiv.org/abs/1609.03499>)¹³ Aaron van den Oord wraz z innymi członkami zespołu DeepMind zaproponował architekturę **WaveNet**. Ułożyli stos jednowymiarowych warstw splotowych, gdzie w każdej warstwie podwajali współczynnik rozrzędzenia (stopień rozprzestrzenienia wejść każdego neuronu): pierwsza warstwa splotowa umożliwia zajrzenie do zaledwie dwóch taktów naraz, natomiast w następnej liczba ta zwiększa się do czterech taktów (pole recepcyjne ma rozmiar czterech taktów), kolejna warstwa „widzi” osiem taktów itd. (rysunek 15.11). W ten sposób niższe warstwy poznają wzory krótkoterminowe, natomiast warstwy wyższe uczą się wzorców długoterminowych. Dzięki podwajaniu współczynnika rozrzędzenia sieć może bardzo skutecznie przetwarzać skrajnie długie sekwencje.



Rysunek 15.11. Struktura modelu *WaveNet*

We wspomnianym artykule autorzy w rzeczywistości wprowadzili trzy bloki, z których każdy zawiera dziesięć warstw splotowych o rosnących współczynnikach rozrzędzenia (1, 2, 4, 8, ..., 256, 512). Wyjaśnili, że taki pojedynczy blok zawierający stos dziesięciu warstw splotowych o podanych współczynnikach rozrzędzenia działa jak niezwykle wydajna warstwa splotowa o rozmiarze jądra równym 1024 (która działa jednak znacznie szybciej, skuteczniej i wykorzystuje o wiele mniej pa-

¹³ Aaron van den Oord i in., *WaveNet: A Generative Model for Raw Audio*, arXiv preprint arXiv:1609.03499 (2016).

rametrów), stąd pomysł na wprowadzenie trzech takich bloków. Przed każdą warstwą następuje również uzupełnianie zerami po lewej stronie danych wejściowych o długość równą współczynnikowi rozrzedzenia po to, aby zachować w sieci taką samą długość sekwencji. Oto implementacja uproszczonej sieci WaveNet służąca do przetwarzania takich samych sekwencji, jakie wykorzystywaliśmy wcześniej¹⁴:

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20, kernel_size=2, padding="causal",
                                 activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam", metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                      validation_data=(X_valid, Y_valid))
```

Ten model Sequential jawnie definiuje na początku warstwę wejściową (jest to prostsze od próby wyznaczenia `input_shape` wyłącznie dla pierwszej warstwy), po której zostaje wstawiona jednowymiarowa warstwa splotowa wykorzystująca uzupełnianie zerami typu "causal" — w ten sposób warstwa ta nie będzie „zagładała” w przyszłość podczas prognozowania (jest to równoważne uzupełnianiu lewej strony danych wejściowych o odpowiednią liczbę zer za pomocą techniki "valid"). Następnie dodajemy podobne pary warstw o rosnących współczynnikach rozrzedzenia: 1, 2, 4, 8 i ponownie 1, 2, 4, 8. Na koniec wstawiamy warstwę wyjściową — jest to warstwa splotowa zawierająca dziesięć filtrów o rozmiarze 1 pozbawiona funkcji aktywacji. Dzięki warstwom uzupełniania zerami każda warstwa splotowa generuje sekwencję o długości takiej samej jak sekwencja wejściowa, zatem wykorzystywane w trakcie uczenia zmienne docelowe mogą być pełnymi sekwencjami — nie trzeba ich ani obcinać, ani zmniejszać.

Ostatnie dwa z omówionych modeli osiągają największą skuteczność w prognozowaniu szeregow czasowych. Twórcy modelu WaveNet uzyskali niespotykaną wcześniej wydajność w różnorodnych zadaniach dźwiękowych (stąd nazwa tej architektury), takich jak przetwarzanie tekstu na mowę, w których udało się uzyskać niewiarygodnie realistyczne głosy w kilku językach. Model ten posłużył także do generowania muzyki, po jednej próbce dźwiękowej. Osiągnięcie to jest tym bardziej imponujące, gdy uświadomimy sobie, że jedna sekunda dźwięku może zawierać dziesiątki tysięcy taktów i nawet komórki LSTM i GRU nie są w stanie przetwarzać tak długich sekwencji.

W rozdziale 16. będziemy dalej zajmować się sieciami rekurencyjnymi i dowiesz się, jak sobie radzą z zadaniami przetwarzania języka naturalnego.

¹⁴ Pełna struktura WaveNet wykorzystuje kilka dodatkowych sztuczek, takich jak połączenia pomijające (spotykane również w sieci ResNet), a także **bramkowane jednostki aktywacji** (ang. *Gated Activation Units*), przypominające struktury spotykane w komórkach GRU. Więcej szczegółów znajdziesz w notatniku Jupyter.

Ćwiczenia

1. Podaj kilka zastosowań rekurencyjnej sieci sekwencyjnej, sekwencyjno-wektorowej i wektorowo-sekwencyjnej.
2. Ile wymiarów muszą zawierać dane wejściowe warstwy rekurencyjnej? Co symbolizuje każdy wymiar? Jak to wygląda w przypadku danych wyjściowych?
3. Gdybyśmy chcieli utworzyć głęboką rekurencyjną sieć sekwencyjną, które warstwy powinny zawierać argument `return_sequences=True`? Porównaj to z głęboką siecią sekwencyjno-wektorową.
4. Powiedzmy, że masz dostęp do codziennego jednowymiarowego szeregu czasowego i chcesz uzyskać prognozy na siedem kolejnych dni. Która architektura rekurencyjną należy wybrać?
5. Jakie są największe problemy związane z uczeniem sieci rekurencyjnych? Jak można sobie z nimi radzić?
6. Narysuj schemat komórki LSTM.
7. Dlaczego warto wprowadzać jednowymiarowe warstwy splotowe do sieci rekurencyjnej?
8. Która architektura sieci neuronowej nadaje się do klasyfikowania filmów?
9. Wytrenuj model klasyfikacyjny za pomocą zestawu danych SketchRNN, dostępnego w ramach projektu TensorFlow Datasets.
10. Pobierz zestaw chorałów Bacha (<https://homl.info/bach>) i rozpakuj go. Ten zestaw danych składa się z 382 chorałów skomponowanych przez Jana Sebastiana Bacha. Każdy chorał zawiera od 100 do 640 taktów, natomiast w każdym takcie mieszczą się cztery liczby całkowite odpowiadające zapisowi nutowemu (oprócz wartości 0, która oznacza pauzę). Wyucz model (rekurencyjny, splotowy lub mieszany), zdolny do przewidywania kolejnego taktu (czterech nut) na podstawie taktów tworzących chorał. Następnie za pomocą tego modelu wygeneruj muzykę „bachopodobną”, po jednej nuce — możesz to zrobić poprzez podanie modelowi początku chorału, prognozowanie kolejnego taktu, dołączenie wygenerowanych taktów do sekwencji wejściowej, która tak zmodyfikowana posłuży do wygenerowania kolejnych nut itd. Sprawdź także model Coconet firmy Google (<https://magenta.tensorflow.org/coconet>), który posłużył do stworzenia ciekawego tymczasowego logo ku pamięci Bacha.

Rozwiązania ćwiczeń znajdziesz w dodatku A.

Przetwarzanie języka naturalnego za pomocą sieci rekurencyjnych i mechanizmów uwagi

Gdy Alan Turing w 1950 roku zaproponował swój słynny test (<http://cogprints.org/499/1/turing.html>)¹, za cel wyznaczył sobie ocenę możliwości maszyny w naśladowaniu inteligencji człowieka. Co ciekawe, mógł przeprowadzać testy w różnych dziedzinach, takich jak rozpoznawanie kotów na zdjęciach, gra w szachy, komponowanie muzyki czy szukanie wyjścia z labiryntu, ale wybrał zadanie lingwistyczne. Mówiąc dokładniej, zaprojektował **czatbota**, który potrafił przekonać rozmówcę, że jest człowiekiem². Test ten nie jest pozbawiony wad: zbiór odgórnie wyznaczonych reguł może oszukać naiwne lub niczego nie podejrzewające osoby (np. maszyna może dawać ogólnikowe odpowiedzi w reakcji na jakieś słowo kluczowe, może udawać, że żartuje lub jest pijana, albo może unikać trudnych pytań, odpowiadając na nie własnymi pytaniami), a wiele aspektów ludzkiej inteligencji zostaje zupełnie zignorowanych (np. umiejętność komunikacji niewerbalnej, takiej jak interpretowanie mimiki, albo uczenie się zadań manualnych). Test ten jednak odzwierciedla fakt, że opinanowanie języka jest prawdopodobnie największym osiągnięciem poznawczym naszego gatunku. Czy jesteśmy w stanie zbudować maszynę potrafiącą zarówno odczytywać, jak i zapisywać język naturalny?

W zadaniach **przetwarzania języka naturalnego** (ang. *Natural Language Processing* — NLP) często są wykorzystywane rekurencyjne sieci neuronowe. Z tego powodu będziemy kontynuować analizę sieci neuronowych (rozpoczętą w rozdziale 15.), począwszy od **znakowych sieci RNN** (ang. *character RNN*), trenowanych w prognozowaniu następnego znaku w zdaniu. W ten sposób wygenerujemy tekst, a jednocześnie nauczymy się tworzyć zestaw danych TensorFlow Dataset za pomocą bardzo długich sekwencji. Najpierw wykorzystamy **bezstanową sieć rekurencyjną** (ang. *stateless RNN*; w każdej iteracji poznaje ona losowe fragmenty tekstu, bez dostępu do informacji na temat reszty tekstu), po czym zbudujemy **stanową sieć RNN** (ang. *stateful RNN*; przechowuje ona stan ukryty po-

¹ Alan Turing, *Computing Machinery and Intelligence*, „Mind” 49 (1950), s. 433 – 460.

² Termin **czatbot** pojawił się oczywiście znacznie później. Turing nazwał swój test **grą tajemnic** (ang. *imitation game*): maszyna A i człowiek B porozumiewają się z ludzkim „śledczym” C za pomocą wiadomości tekstowych, a zadaniem „śledzkiego” jest takie poprowadzenie rozmowy, aby był w stanie odróżnić maszynę od człowieka. Maszyna zdaje test, gdy uda jej się oszukać „śledzkiego”, natomiast człowiek B ma mu pomagać.

między przebiegami uczącymi i kontynuuje czytanie od miejsca, w którym zostało przerwane, co pozwala wykrywać dłuższe wzorce). Następnie przygotujemy sieć przeprowadzającą analizę sentymentów (tzn. czytającą recenzje filmów i na ich podstawie określającą odczucia recenzentów), w tym przypadku traktując zdania jako sekwencje wyrazów, a nie znaków. W dalszej części rozdziału przekonasz się, że za pomocą sieci rekurencyjnych możemy budować strukturę kodera – dekodera realizującą zadania **neuronowego tłumaczenia maszynowego** (ang. *Neural Machine Translation* — NMT). Posłuży nam do tego interfejs seq2seq, stanowiący część projektu TensorFlow Addons.

Druga część niniejszego rozdziału została poświęcona **mechanizmom uwagi** (ang. *attention mechanisms*). Jak wskazuje sama nazwa, stanowią one elementy sieci neuronowej uczącej się wyznaczania określonych składowych sygnału wejściowego, na których będzie koncentrować się reszta modelu w kolejnych taktach. Najpierw dowiemy się, jak można usprawnić za pomocą mechanizmów uwagi wydajność struktury kodera – dekodera opartej na sieci rekurencyjnej, a następnie zrezygnujemy całkowicie z architektury RNN i przyjrzymy się niezwykle udanej architekturze bazującej wyłącznie na mechanizmach uwagi, zwanej **transformatorem** (ang. *transformer*). Na koniec zajmiemy się najważniejszymi postępami w dziedzinie NLP dokonanymi w latach 2018 i 2019, w tym bardzo wydajnymi modelami językowymi GPT-2 i BERT, opartymi na transformatorach.

Zacznijmy od prostego i zabawnego modelu generującego teksty szekspirowskie (poniekąd).

Generowanie tekstów szekspirowskich za pomocą znakowej sieci rekurencyjnej

W sławetnym wpisie z 2015 roku zatytułowanym *The Unreasonable Effectiveness of Recurrent Neural Networks* (<https://karpathy.github.io/2015/05/21/rnn-effectiveness/>) Andrej Karpathy zademonstrował sposób uczenia sieci rekurencyjnej w zadaniu przewidywania następnego znaku w zdaniu. Taka sieć **Char-RNN** może posłużyć do tworzenia zupełnie nowego tekstu, znak po znaku. Oto mała próbka tekstu wygenerowanego przez model Char-RNN po wytrenowaniu go na dziełach Szekspira:

PANDARUS:

Alas, I think he shall be come approached and the day

When little strain would be attain'd into being never fed,

And who is but a chain and subjects of his death,

I should not sleep.

Nie jest to może jakieś wiekopomne dzieło, ale i tak wzbudza uznanie fakt, że model był w stanie poznać słowa, gramatykę, interpunkcję itd. wyłącznie poprzez prognozowanie kolejnych znaków w zdaniu. Zbudujmy teraz od podstaw taką sieć Char-RNN, począwszy od utworzenia zestawu danych.

Tworzenie zestawu danych uczących

Pobierzmy najpierw wszystkie dzieła Szekspira za pomocą funkcji `get_file()` i dane z projektu Char-RNN (<https://github.com/karpathy/char-rnn>):

```
shakespeare_url = "https://homl.info/shakespeare" # Skrótny adres URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Musimy następnie zakodować każdy znak jako liczbę całkowitą. Jednym z rozwiązań jest utworzenie niestandardowej warstwy wstępniego przetwarzania zgodnie z informacjami zawartymi w rozdziale 13. W tym jednak przypadku łatwiej będzie użyć klasy `Tokenizer`. Najpierw musimy dopasować tokenizator do tekstu — wyszuka on wszystkie znaki użyte w tekście i przypisze każdy z nich do osobnego identyfikatora znaków, od 1 do liczby poszczególnych znaków (do wartości 0 nie zostaje przypisany żaden znak, możemy więc wykorzystać ją do maskowania, o czym przekonasz się w następnym rozdziale):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

Wyznaczamy `char_level=True`, aby uzyskać kodowanie znaków, a nie wyrazów. Zwróć uwagę, że domyślnie tokenizator przekształca znaki w całym tekście na małe litery (jeśli jednak tego nie chcesz, możesz wyznaczyć atrybut `lower=False`). Tokenizator jest teraz gotowy do zakodowania zdania (lub listy zdań) do postaci listy identyfikatorów znaków (i odwrotnie); poinformuje nas także o liczbie różnych znaków w tekście i o całkowitej liczbie znaków:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index) # Liczba poszczególnych rodzajów znaków
>>> dataset_size = tokenizer.document_count # Całkowita liczba znaków
```

Zakodujmy cały tekst tak, aby każdy znak był reprezentowany za pomocą identyfikatora (odejmujemy 1, aby otrzymać identyfikatory w zakresie wartości od 0 do 38 zamiast od 1 do 39):

```
[encoded] = np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Zanim przejdziemy dalej, musimy podzielić zestaw danych na zbiory uczący, walidacyjny i testowy. Nie możemy tak po prostu potasować wszystkich znaków w tekście, jak więc rozdzielamy zestaw danych sekwencyjnych?

Rozdzielanie zestawu danych sekwencyjnych

Jest bardzo ważne, aby unikać nakładania się danych ze zbiorów uczącego, walidacyjnego i testowego. Możemy na przykład przeznaczyć pierwsze 90% tekstu na zbiór uczący, następnie 5% na zbiór walidacyjny i pozostałe 5% na zbiór testowy. Niegłupim pomysłem byłoby również pozostawienie przerwy pomiędzy poszczególnymi zbiorami, co pozwoli uniknąć nakładania się skrajnych akapitów w dwóch zbiorach.

Podczas pracy na szeregach czasowych zazwyczaj dokonujemy podziału wzdłuż osi czasu: na przykład na zbiór uczący możemy przeznaczyć dane pochodzące z lat 2000 – 2012, na zbiór walidacyjny dane z lat 2013 – 2015, a na zbiór testowy dane z lat 2016 – 2018. Jednak w pewnych sytuacjach istnieje możliwość rozdzielania danych wzdłuż innych wymiarów, dzięki czemu model będzie uczyony na dłuższym przedziale czasu. Jeżeli na przykład dysponujesz danymi o kondycji finansowej 10 000 firm z lat od 2000 do 2018, to możesz rozdzielić te dane na podstawie różnych firm. Istnieje jednak spore prawdopodobieństwo, że wiele z tych przedsiębiorstw jest ze sobą silnie skorelowanych (na przykład całe sektory gospodarki mogą wspólnie maleć lub wzrastać), a jeżeli takie skorelowane firmy zostały rozdzielone pomiędzy zbiory uczący i testowy, ten drugi okaże się niezbyt przydatny, gdyż wskaźnik błędu uogólniania będzie ukierunkowany optymistycznie.

Dlatego nieraz bezpieczniej jest rozdzielać dane wzdłuż osi czasu, ale w ten sposób zakładamy niejawnie, że wzorce poznawane przez sieć rekurencyjną w przeszłości (w zbiorze uczącym) występują również w przyszłości. Inaczej mówiąc, zakładamy, że szereg czasowy jest **stacjonarny** (przynajmniej w ogólnym znaczeniu)³. W przypadku wielu szeregów czasowych założenie to jest rozsądne (np. nie powinno być problemu z danymi reakcji chemicznych, ponieważ prawa fizyki pozostają niezmienne), ale istnieje też duża grupa szeregów czasowych, których to nie dotyczy (przykładowo rynki finansowe prawie nigdy nie są stacjonarne, ponieważ wzorce zanikają niemal dokładnie wtedy, gdy zostaną odkryte i będą eksploatowane przez spekulantów). Aby mieć pewność, że dany szereg czasowy jest wystarczająco stacjonarny, możesz sprawdzić przebieg wykresu błędów modelu dla zbioru walidacyjnego w czasie: jeżeli model uzyskuje znacznie lepszą wydajność dla pierwszej części zbioru niż dla drugiej, to szereg czasowy może nie być wystarczająco stacjonarny i lepszym rozwiązaniem może okazać się uczenie modelu w krótszym przedziale czasu.

Mówiąc krótko, rozdzielenie szeregu czasowego na zbiory uczący, walidacyjny i testowy nie jest trywialnym wyzwaniem, a sposób jego realizacji zależy od wykonywanego przez model zadania.

Wróćmy do Szekspira! Przeznaczmy pierwsze 90% tekstu na zbiór uczący (reszta posłuży nam do utworzenia zbiorów walidacyjnego i testowego) i utwórzmy obiekt klasy `tf.data.Dataset`, który będzie zwracał pojedynczo znaki z tego zbioru:

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

Dzielenie zestawu danych sekwencyjnych na wiele ramek

Zbiór uczący składa się teraz z pojedynczej sekwencji przechowującej ponad milion znaków, zatem nie możemy bezpośrednio wytrenować sieci za jego pomocą — sieć rekurencyjna musiałaby stanowić odpowiednik sieci głębokiej zawierającej ponad milion warstw, a my dysponowałibyśmy jednym (bardzo dużym) przykładem uczącym. Zamiast tego skorzystamy z metody `window()` zestawu danych, dzięki której przekształcimy tę długą sekwencję znaków w wiele mniejszych ramek tekstowych. Każdy przykład tworzący zestaw danych będzie stanowił w miarę krótki podzbiór całego tekstu, a sieć

³ Zgodnie z definicją średnia, wariancja i **autokorelacje** (tzn. korelacje pomiędzy wartościami szeregu czasowego rozdzielonymi określonym odstępem) stacjonarnego szeregu czasowego nie zmieniają się w miarę upływu czasu. Jest to dość restrykcyjny warunek — wyklucza on na przykład szeregi czasowe zawierające trendy lub wzorce cykliczne. Sieci rekurencyjne są bardziej tolerancyjne, gdyż rozpoznają trendy i wzorce cykliczne.

rekurencyjna zostanie rozwinięta jedynie na długość tych podzbiorów. Jest to tak zwana **przycięta propagacja wsteczna w czasie** (ang. *truncated backpropagation through time*). Wywołajmy teraz metodę `window()` i utwórzmy zbiór danych składający się z krótkich ramek tekstowych:

```
n_steps = 100
window_length = n_steps + 1 # zmienna docelowa = sygnał wejściowy przesunięty o jeden znak do przodu
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



Możesz spróbować dostroić parametr `n_steps` — łatwiej jest uczyć sieć rekurencyjną na krótszych sekwencjach wejściowych, ale oczywiście model nie będzie w stanie rozpoznawać żadnych wzorców dłuższych od `n_steps`, dlatego wyznacz odpowiednio dużą wartość.

Domyślnie metoda `window()` tworzy nienakładające się ramki, aby jednak uzyskać jak największy zbiór uczący, wyznaczamy argument `shift=1`, dzięki czemu pierwsza ramka zawiera znaki od 0. do 100., druga przechowuje znaki od 1. do 101. itd. Aby mieć pewność, że wszystkie ramki będą miały długość 101 znaków (w ten sposób utworzymy grupy bez konieczności uzupełniania zerami), wprowadzamy parametr `drop_remainder=True` (w przeciwnym wypadku 100 ostatnich ramek będzie zawierało, kolejno, 100 znaków, 99 znaków i tak dalej aż do ramki zawierającej jeden znak).

Metoda `window()` tworzy zestaw danych zawierający ramki, z których każda jest reprezentowana jako zbiór danych. Jest to **zagnieżdżony zestaw danych** (ang. *nested dataset*), przypominający listę list. Rozwiążanie to przydaje się wtedy, gdy chcesz przekształcać każdą ramkę poprzez wywołanie odpowiednich metod (np. tasujących lub grupujących je). Nie możemy jednak użyć zagnieżdzonego zestawu danych bezpośrednio w procesie uczenia, ponieważ model oczekuje na wejściu tensorów, nie zestawów danych. Musimy zatem wywołać metodę `flat_map()`, która przekształca zagnieżdżony zestaw danych w **płaski zestaw danych** (ang. *flat dataset*; nie zawiera on podzbiorów danych). Założymy na przykład, że zapis `{1, 2, 3}` symbolizuje zestaw danych zawierający sekwencję tensorów 1, 2 i 3. Jeżeli spłaszczysz zagnieżdżony zestaw danych `{[1, 2], [3, 4, 5, 6]}`, to otrzymasz płaski zestaw danych `[1, 2, 3, 4, 5, 6]`. Ponadto metoda `flat_map()` przyjmuje funkcję jako argument, dzięki czemu masz możliwość przekształcania każdego podzbioru w zagnieżdżonym zestawie danych przed jego spłaszczeniem. Przykładowo jeżeli przekażesz funkcję `lambda ds: ds.batch(2)` do metody `flat_map()`, to zagnieżdżony zestaw danych `{[1, 2], [3, 4, 5, 6]}` zostanie przekształcony w płaski zestaw danych `[[1, 2], [3, 4], [5, 6]]`: zostanie on przeobrażony w płaski zbiór tensorów o rozmiarze 2. Zapamiętujmy to i jesteśmy już gotowi do spłaszczenia naszego zestawu danych:

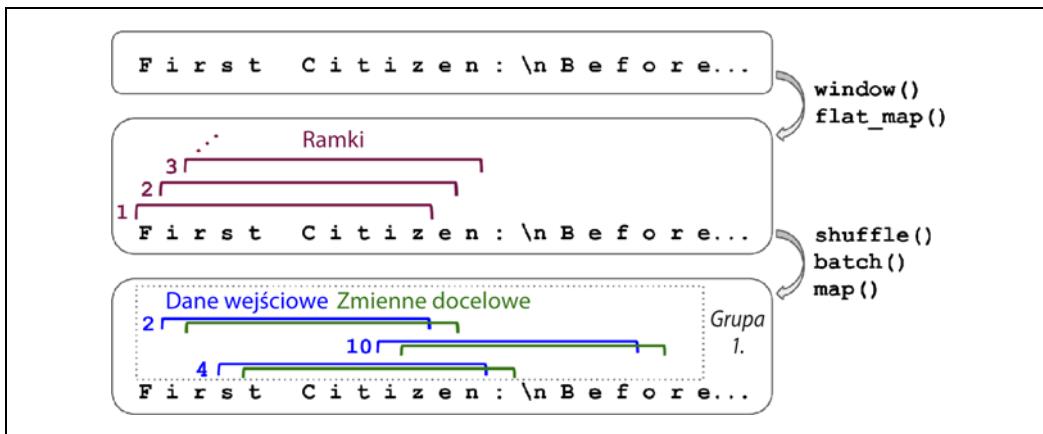
```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Zauważ, że dla każdej ramki wywołujemy metodę `batch(window_length)`. Wszystkie ramki mają taką samą długość, dlatego dla każdej z nich uzyskamy pojedynczy tensor. Teraz zestaw danych zawiera występujące po sobie ramki przechowujące po 101 znaków. Algorytm gradientu prostego sprawdza się najlepiej, gdy przykłady tworzące zbiór uczący są zmiennymi niezależnymi o takim samym rozkładzie (zob. rozdział 4.), dlatego musimy przetasować te ramki. Następnie możemy je pogrupować i oddzielić sygnały wejściowe (100 pierwszych znaków) od zmiennej docelowej (ostatni znak):

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))


```

Rysunek 16.1 pokazuje omówione etapy przygotowywania zestawu danych (ramkę o długości 101 zastąpiłem ramką o długości 11, a rozmiar grupy 32 rozmiarem 3).



Rysunek 16.1. Przygotowywanie zestawu danych składającego się z potasowanych ramek

Jak już wiesz z rozdziału 13., kategorialne cechy wejściowe powinny być zasadniczo kodowane albo jako wektory gorącojedynkowe, albo jako wektory właściwościowe. Zakodujemy każdy znak w formę gorącojedynkową, ponieważ nie występuje tu tak wiele osobnych znaków (zaledwie 39):

```
dataset = dataset.map(  
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

Teraz wystarczy dodać ładowanie wstępne:

```
dataset = dataset.prefetch(1)
```

To wszystko! Przygotowanie zestawu danych było najtrudniejszym elementem. Przejdzmy do budowania modelu.

Budowanie i uczenie modelu Char-RNN

Do przewidywania następnego znaku na podstawie stu poprzednich możemy wykorzystać sieć rekurencyjną zawierającą dwie warstwy GRU, po 128 jednostek każda i 20% porzucania, zarówno danych wejściowych (dropout), jak i stanów ukrytych (recurrent_dropout). W razie potrzeby możemy później zmodyfikować te hiperparametry. Warstwa wyjściowa to zbliżona do omówionej w rozdziale 15. rozwijanej w czasie warstwy Dense. W tym przypadku musi ona składać się z 39 jednostek (max_id), ponieważ w tekście występuje 39 różnych znaków, a my chcemy uzyskać prawdopodobieństwo występowania każdego znaku (w każdym taktcie). W każdym taktcie prawdopodobieństwa wynikowe powinny po zsumowaniu być równe 1, dlatego wprowadzamy funkcję aktywacji softmax na wyjściach warstwy Dense. Teraz możemy skompilować model przy użyciu funkcji straty "sparse_categorical_crossentropy" i optymalizatora Adam. Na koniec wystarczy wyuczyć model przez kilka epok (w zależności od komputera proces ten może zajść kilka godzin):

```
model = keras.models.Sequential([  
    keras.layers.GRU(128, return_sequences=True, input_shape=[None, max_id],  
    dropout=0.2, recurrent_dropout=0.2),
```

```

        keras.layers.GRU(128, return_sequences=True,
                           dropout=0.2, recurrent_dropout=0.2),
        keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                       activation="softmax")))
    ])
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
history = model.fit(dataset, epochs=20)

```

Korzystanie z modelu Char-RNN

Dysponujemy teraz modelem, który może przewidywać następny znak w tekście szekspirowskim. Aby dostarczyć modelowi tekst, musimy najpierw wstępnie przetworzyć dane, utwórzmy więc przeznaczoną do tego celu niewielką funkcję:

```

def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)

```

Użyjmy teraz modelu do prognozowania następnej litery w jakimś przykładowym tekście:

```

>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1] # Pierwsze zdanie, ostatni znak
'u'

```

Sukces! Model odgadł właściwą literę. Wygenerujmy teraz nowy tekst za pomocą tego modelu.

Generowanie sztucznego tekstu szekspirowskiego

Aby wygenerować nowy tekst za pomocą modelu Char-RNN, moglibyśmy dostarczyć mu jakiś tekst, sprawić, aby przewidział najbardziej prawdopodobną następną literę, dodać ją na koniec tekstu, następnie przekazać modelowi tak rozszerzony tekst, aby odgadł kolejną literę, itd. W praktyce jednak rozwiązanie to sprawia, że w kółko powtarzane są te same słowa. Zamiast tego możemy wybrać losowo następny znak, o prawdopodobieństwie równym oszacowanemu prawdopodobieństwu za pomocą funkcji `tf.random.categorical()`. Dzięki temu uzyskamy bardziej zróżnicowany i interesujący tekst. Funkcja `categorical()` losuje indeksy klas na podstawie logarytmów prawdopodobieństwa (logitów) przynależności do klasy. Możemy uzyskać większą kontrolę nad zróżnicowaniem generowanego tekstu poprzez dzielenie logitów przez wartość zwaną **temperaturą**, którą możemy dowolnie modyfikować: temperatura zbliżona do zera będzie faworyzowała znaki o dużym prawdopodobieństwie, natomiast bardzo wysoka temperatura wyrównuje szanse wszystkich znaków. Poniższa funkcja `next_char()` wykorzystuje omówione rozwiązanie do wybierania następnego znaku, który zostanie dodany do tekstu wejściowego:

```

def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id = tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]

```

Możemy teraz rozpisać małą funkcję, która będzie systematycznie wywoływać funkcję `next_char()`:

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

Możemy wygenerować w końcu jakiś tekst! Poeksperymentujmy trochę z wartościami temperatury:

```
>>> print(complete_text("t", temperature=0.2))
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rush
```

Najwyraźniej nasz model szekspirowski działa najlepiej w temperaturze bliskiej 1. Gdybyśmy chcieli wygenerować bardziej przekonujący tekst, moglibyśmy spróbować dodać więcej warstw GRU, a także zwiększyć liczbę neuronów w poszczególnych warstwach, wydłużyć czas uczenia oraz wprowadzić pewien stopień regularyzacji (np. wyznaczyć `recurrent_dropout=0,3` w warstwach GRU). Ponadto obecnie model nie jest w stanie rozpoznawać wzorców dłuższych od `n_steps`, czyli w naszym przypadku przekraczających 100 znaków. Możesz spróbować powiększyć tę ramkę, ale jednocześnie proces uczenia stanie się trudniejszy, a nawet komórki GRU i LSTM nie są w stanie przetwarzać bardzo długich sekwencji. Ewentualnie możesz skorzystać ze stanowej sieci rekurencyjnej.

Stanowe sieci rekurencyjne

Do tej pory korzystaliśmy wyłącznie z **bezstanowych sieci rekurencyjnych**: w każdej iteracji stan ukryty modelu jest wypełniony zerami, po czym w każdym takcie wartości te są aktualizowane, a po ostatnim taktie zostają znowu wyzerowane, gdyż przestają być potrzebne. A gdybyśmy kazali sieci RNN zachować ten stan końcowy po przetworzeniu grupy danych i wykorzystać go jako stan początkowy podczas przetwarzania kolejnej grupy? W ten sposób model mógłby rozpoznawać długoterminowe wzorce nawet pomimo propagacji wstępnej poprzez krótkie sekwencje. Jest to **stanowa sieć rekurencyjna**. Zobaczmy, jak można ją stworzyć.

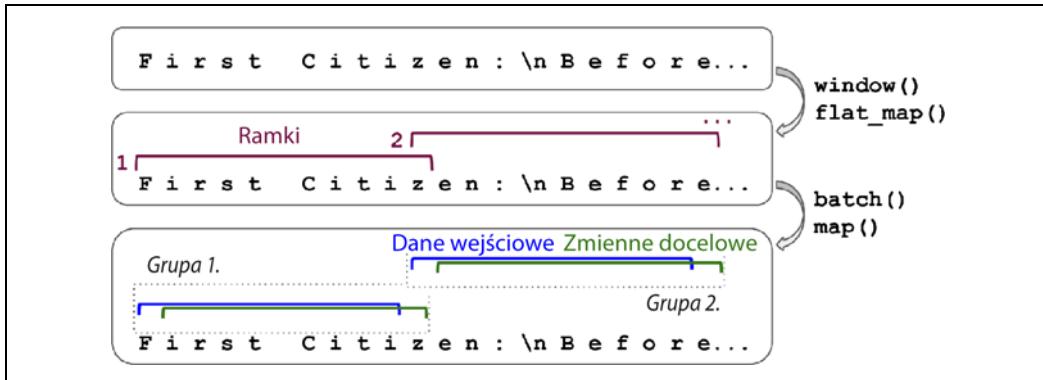
Należy najpierw zauważyc, że stanowa sieć rekurencyjna ma sens tylko wtedy, gdyż każda sekwencja wejściowa w grupie rozpoczyna się dokładnie tam, gdzie zakończyła się sekwencja w poprzedniej grupie. Zatem pierwszą czynnością podczas budowania stanowej sieci rekurencyjnej jest wykorzystanie sekwencyjnych i nienakładających się sekwencji wejściowych (czyli nie mogą być one ani po-tasowane, ani nakładające się, jak w poprzednim przykładzie). Podczas tworzenia obiektu `Dataset` musimy wyznaczyć `shift=n_steps` (zamiast `shift=1`) w metodzie `window()`. Do tego staje się jasne, że nie możemy wywołać metody `shuffle()`. Niestety grupowanie sekwencji w sieci stanowej jest znacznie trudniejsze niż w sieci bezstanowej. Rzeczywiście, gdybyśmy wywołali funkcję `batch(32)`, to w tej samej grupie zostałyby umieszczone 32 kolejne ramki, a ramki w kolejnej grupie nie rozpoczęłyby się tam, gdzie zakończyły się poprzednie. Pierwsza grupa zawierałaby ramki, powiedzmy, od 1. do 32., druga grupa miałaby ramki od 33. do 64., więc jeżeli będziemy rozpatrywać pierwszą ramkę w każdej grupie (czyli np. ramki 1. i 33.), to wyraźnie widać, że nie następują one bezpośrednio po sobie. Najprostszym rozwiązaniem jest wykorzystanie „grup” składających się z jednej ramki:

```

dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps, drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)

```

Te pierwsze etapy zostały zaprezentowane na rysunku 16.2.



Rysunek 16.2. Przygotowywanie zestawu danych składającego się z kolejno występujących po sobie ramek w stanowej sieci rekurencyjnej

Grupowanie jest utrudnione, ale wykonalne. Moglibyśmy na przykład podzielić tekst szekspirowski na 32 fragmenty o równej długości, stworzyć jeden zestaw danych, zawierający kolejno występujące sekwencje wejściowe dla każdego fragmentu, a następnie wykorzystać funkcję `tf.train.Dataset.zip(datasets).map(lambda*windows: tf.stack(windows))`, tworzącą właściwe grupy, w których n -ta sekwencja wejściowa bieżącej grupy rozpoczyna się dokładnie tam, gdzie kończy się n -ta sekwencja wejściowa poprzedniej grupy (pełny listing znajdziesz w notatniku Jupyter).

Zbudujmy teraz stanową sieć rekurencyjną. Najpierw musimy wyznaczyć parametr `stateful=True` podczas tworzenia każdej warstwy rekurencyjnej. Następnie stanowa sieć RNN musi znać rozmiar grupy (ponieważ będzie przechowywać stan każdej sekwencji wejściowej w grupie), dlatego w pierwszej warstwie musimy wyznaczyć argument `batch_input_shape`. Zwróć uwagę, że możemy pozostawić nieokreślony drugi wymiar, gdyż dane wejściowe mogą mieć dowolną długość:

```

model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                    dropout=0.2, recurrent_dropout=0.2,
                    batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                    dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                activation="softmax"))
])

```

Na końcu każdej epoki musimy wyzerować stany, zanim będziemy mogli wrócić na początek tekstu. W tym celu wykorzystamy małe wywołanie zwrotne:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

Teraz wystarczy skompilować i dopasować model (wykorzystamy więcej epok, ponieważ każda epoka jest teraz znacznie krótsza niż w poprzednim przykładzie, a każda grupa zawiera tylko jeden przykład):

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```



Po wyuczeniu tego modelu będziemy mogli uzyskiwać prognozy jedynie dla grup mających taki sam rozmiar jak użyte podczas trenowania. Aby uniknąć tego problemu, utwórz identyczny, ale **bezstanowy** model i skopij do niego wagi modelu stanowego.

Skoro potrafisz już utworzyć model oparty na znakach, przejdźmy do modeli bazujących na słowach i zajmijmy się popularnym zadaniem przetwarzania języka naturalnego: **analizą sentymentów**. Nauczysz się również obsługiwać sekwencje o zmiennej długości za pomocą maskowania.

Analiza sentymentów

Jeżeli zestaw danych MNIST stanowi sztandarowy element widzenia maszynowego, to jego odpowiednikiem w dziedzinie przetwarzania języka naturalnego jest zestaw danych IMDb reviews, który składa się z 50 000 anglojęzycznych recenzji filmów (25 000 przykładów służy do uczenia modeli, a 25 000 do ich testowania) pozyskanych ze słynnego serwisu Internet Movie Database (<https://www.imdb.com/>), a także z przypisanej do każdej recenzji prostej zmiennej binarnej, która wskazuje, czy recenzja jest negatywna (0) czy pozytywna (1). Podobnie jak w przypadku zestawu MNIST, zestaw IMDb reviews zyskał popularność z dobrych powodów: jest wystarczająco prosty, aby można go było przetwarzać na laptopie w rozsądny czasie, ale jednocześnie okazuje się wystarczająco wymagający, aby zapewniać ubaw i satysfakcję. Możemy go wczytać za pomocą prostej funkcji interfejsu Keras:

```
>>> (X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data()
>>> X_train[0][:10]
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

Gdzie są te recenzje? Jak widać, omawiany zestaw danych został już wcześniej przetworzony: X_train składa się z listy recenzji, z których każda jest reprezentowana w postaci tablicy NumPy, a każda wartość liczbowa w tej tablicy oznacza słowo. Wszystkie znaki interpunkcyjne zostały usunięte, zapis ujednolicono do małych liter, poszczególne słowa rozdzielono spacjami, a na koniec zostały oznaczone indeksami określającymi częstość występowania (mała wartość oznacza dużą częstość występowania danego słowa). Wartości 0, 1 i 2 mają specjalne znaczenie: oznaczają, odpowiednio, znak uzupełniający, początek sekwencji (ang. *start-of-sequence* — sos) i nieznane słowo. Jeżeli chcesz zwizualizować daną recenzję, musisz ją rozkodować:

```
>>> word_index = keras.datasets.imdb.get_word_index()
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}
>>> for id_, token in enumerate(("<pad>", "<sos>", "<unk>")):
...     id_to_word[id_] = token
... 
```

```
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])  
'<sos> this film was just brilliant casting location scenery story'
```

W rzeczywistym projekcie trzeba będzie samodzielnie przetwarzać wstępnie tekst. Możesz to zrobić za pomocą wspomnianej już klasy Tokenizer, tym razem jednak należy pozostawić bez zmian domyślny argument `char_level=False`. Podczas kodowania słów zostaje odfiltrowanych wiele znaków, w tym znaki interpunkcyjne, znaki nowego wiersza czy tabulatory (ale możesz to zmienić za pomocą argumentu `filters`). Najważniejsze jednak, że do określania granic słów wykorzystywane są odstępy. Rozwiązywanie to sprawdza się w tekstach angielskich i innych skryptach (językach), w których wyrazy rozdzielane są spacjami, ale nie we wszystkich skryptach tak jest. W języku chińskim nie ma odstępów pomiędzy wyrazami, w języku wietnamskim są one z kolei wstawiane nawet wewnątrz słów, a takie języki jak niemiecki cechują się łączeniem wielu wyrazów. Nawet w języku angielskim (a także polskim) odstępy nie zawsze stanowią najlepszy sposób tokenizowania tekstu — dobrymi przykładami są „San Francisco” czy „#KochamUczenieMaszynowe”.

Na szczęście istnieją lepsze rozwiązania! Taku Kudo w publikacji z 2018 roku (<https://arxiv.org/abs/1804.10959>)⁴ zaprezentował technikę uczenia nienadzorowanego umożliwiającą tokenizowanie i detokenizowanie tekstu na poziomie elementów słowotwórczych w sposób niezależny od języka, gdzie odstępy są traktowane jak inne znaki. W ten sposób, nawet jeśli model natrafi na nieznany wyraz, jest w stanie w miarę skutecznie wychwycić jego sens. Na przykład być może model nigdy nie natrafił na wyraz „najmądrzejszy” podczas uczenia, ale spotkał się ze słowem „mądrzejszy”, a także dowiedział się, że „naj-” oznacza najwyższy stopień przymiotnika, zatem może wywnioskować znaczenie słowa „najmądrzejszy”. Projekt SentencePiece (<https://github.com/google/sentencepiece>) firmy Google zawiera implementację o otwartym kodzie, opisaną w artykule (<https://arxiv.org/abs/1808.06226>)⁵ przez Taku Kudo i Johna Richardsona.

Nieco wcześniej (<https://arxiv.org/abs/1508.07909>)⁶ Rico Sennrich i in. zaproponowali odmienne rozwiązanie, oparte na tworzeniu kodu słowotwórczego (np. za pomocą kodowania parami bajtów — ang. *byte pair encoding*). Również zespół TensorFlow w czerwcu 2019 roku wydał bibliotekę TF.Text (<https://medium.com/tensorflow/introducing-tf-text-438c8552bd5e>), która implementuje wiele strategii tokenizacji, w tym taką jak WordPiece (<https://arxiv.org/abs/1609.08144>)⁷, czyli wariant kodowania parami bajtów.

Jeżeli chcesz wdrożyć model na urządzeniu mobilnym lub w przeglądarce, a nie masz ochoty pisać za każdym razem osobnej funkcji wstępnego przetwarzania, możesz zlecić wstępne przetwarzanie wyłącznie operacjom TensorFlow, dzięki czemu będzie można je wstawić bezpośrednio w modelu. Zobaczmy, jak można to zrobić. Najpierw wczytajmy pierwotny zestaw recenzji IMDb jako tekst (łańcuchy bajtowe) za pomocą projektu TensorFlow Datasets (zob. rozdział 13.):

⁴ Taku Kudo, *Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates*, arXiv preprint arXiv:1804.10959 (2018).

⁵ Taku Kudo i John Richardson, *SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing*, arXiv preprint arXiv:1808.06226 (2018).

⁶ Rico Sennrich i in., *Neural Machine Translation of Rare Words with Subword Units*, „Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics” 1 (2016), s. 1715 – 1725.

⁷ Yonghui Wu i in., *Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation*, arXiv preprint arXiv:1609.08144 (2016).

```
import tensorflow_datasets as tfds

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
```

Napiszmy teraz funkcję wstępnego przetwarzania:

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z']", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

Najpierw skracamy za jej pomocą wszystkie recenzje i pozostawiamy jedynie po 300 znaków w każdej z nich — przyspieszymy w ten sposób proces uczenia, a jednocześnie nie wpłyniemy zbytnio na wydajność modelu, ponieważ zazwyczaj już po pierwszym lub drugim zdaniu wiadomo, czy opinia o filmie jest pozytywna czy negatywna. Następnie wprowadzamy wyrażenia regularne, za pomocą których zastępujemy odstępami znaczniki `
`, a także wszelkie inne znaki inne niż litery i cudzysłowy. Na przykład zdanie `"Well, I can't
"` zostanie przekształcone w postać `"Well I can't"`. Na koniec funkcja `preprocess()` rozdziela recenzje odstępami, przez co zostaje zwrocony tensor nierówny, który zostaje przekształcony w tensor gęsty; wszystkie recenzje zostają uzupełnione tokenem `"<pad>"`, dzięki czemu będą miały taką samą długość.

Następnie musimy skonstruować wokabularz. Oznacza to wykonanie jednego pełnego przebiegu po całym zbiorze uczącym, użycie funkcji `preprocess()` i zastosowanie licznika `Counter` do policzenia wystąpień każdego słowa:

```
from collections import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Sprawdźmy, które słowa pojawiają się najczęściej:

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Znakomicie! Prawdopodobnie nasz model nie musi znać wszystkich wyrazów wokabularza, aby uzyskać dobrą wydajność, dlatego ograniczymy nasz wokabularz do 10 000 najczęściej pojawiających się słów:

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Musimy dodać teraz operację zastępowania wyrazów ich identyfikatorami (tzn. indeksami w wokabularzu). Podobnie jak to robiliśmy w rozdziale 13., utworzymy tablicę przeglądową i wprowadzimy 1000 bloków pozawokabularzowych:

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

Możemy teraz za pomocą tej tablicy przeglądowej sprawdzić kilka identyfikatorów wyrazów:

```
>>> table.lookup(tf.constant([b"This movie was faaaaantastic".split()]))  
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22,  12,  11, 10054]])>
```

Wyrazy „this,” „movie” i „was” zostały znalezione w wokabularzu, więc ich identyfikatory są mniejsze od 10 000, natomiast nie udało się znaleźć słowa „faaaaaantastic”, zostało więc ono umieszczone w nowym bloku oov o identyfikatorze równym lub większym od 10 000.



Omówione w rozdziale 13. narzędzie TF Transform zawiera przydatne funkcje obsługujące tego typu wokabularze. Sprawdź na przykład funkcję `tft.compute_and_apply_vocabulary()`: przeanalizuje ona cały zestaw danych, wykryje wszystkie oddzielne słowa, z których zbuduje wokabularz, a następnie wygeneruje operacje TF wymagane do kodowania każdego słowa za pomocą tego wokabularza.

Jesteśmy gotowi utworzyć końcowy zbiór uczący. Grupujemy recenzje, przekształcamy je w krótkie sekwencje słów za pomocą funkcji `preprocess()`, kodujemy te słowa przy użyciu prostej funkcji `encode_words()`, które wykorzystują przygotowaną przez nas tablicę, po czym zostaje wstępnie załadowana kolejna grupa danych:

```
def encode_words(X_batch, y_batch):  
    return table.lookup(X_batch), y_batch  
  
train_set = datasets["train"].batch(32).map(preprocess)  
train_set = train_set.map(encode_words).prefetch(1)
```

W końcu możemy zbudować i wyuczyć model:

```
embed_size = 128  
model = keras.models.Sequential([  
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,  
                           input_shape=[None]),  
    keras.layers.GRU(128, return_sequences=True),  
    keras.layers.GRU(128),  
    keras.layers.Dense(1, activation="sigmoid")  
])  
model.compile(loss="binary_crossentropy", optimizer="adam",  
              metrics=["accuracy"])  
history = model.fit(train_set, epochs=5)
```

Jako pierwsza występuje tu warstwa Embedding, która będzie przekształcać identyfikatory w wektory właściwościowe (zob. rozdział 13.). Macierz właściwościowa musi zawierać po jednym rzędzie na każdy identyfikator słowa (`vocab_size + num_oov_buckets`) i po każdej kolumnie na każdy wymiar wektora właściwościowego (w naszym przykładzie wykorzystujemy 128 wymiarów, ale ten hiperparametr można dostroić). Dane wejściowe modelu to dwuwymiarowe tensory o wymiarach [*rozmiar grupy, takty*], natomiast na wyjściu warstwy Embedding jest tworzony trójwymiarowy tensor o wymiarach [*rozmiar grupy, takty, rozmiar wektora właściwościowego*].

Pozostała część modelu nie wymaga wyjaśnienia: składa się z dwóch warstw GRU, gdzie druga zwraca jedynie wynik z ostatniego taktu. Warstwę wyjściową tworzy jeden neuron ze zdefiniowaną sigmoidalną funkcją aktywacji, co pozwala uzyskać szacowane prawdopodobieństwo pozytywnego ładunku emocjonalnego wykrytego w recenzji filmu. Bez problemu komplujemy model i uczymy go przez kilka epok na odpowiednio przygotowanym zestawie danych.

Maskowanie

W obecnej formie model musi się nauczyć, że należy ignorować tokeny uzupełniające. My to jednak już wiemy! Może po prostu każmy mu ignorować te tokeny, dzięki czemu będzie mógł skoncentrować się na istotnych danych? Zadanie to jest w istocie banalne: wystarczy dodać argument `mask_zero=True` podczas tworzenia warstwy `Embedding`. Oznacza to, że tokeny uzupełniające (o identyfikatorze równym 0)⁸ będą ignorowane przez wszystkie następne warstwy. To wszystko!

Mechanizm działania jest następujący: warstwa `Embedding` tworzy **tensor maski** (ang. *mask tensor*) równy `K.not_equal(inputs, 0)` (gdzie `K = keras.backend`). Jest to tensor logiczny o wymiarach takich samych jak dane wejściowe, w którym występuje wartość `False` wszędzie tam, gdzie identyfikatory wyrazów są równe 0, a w przeciwnym razie zostaje umieszczona wartość `True`. Tensor maski zostaje następnie automatycznie rozprowadzony do następnych warstw, jeśli tylko wymiar czasu jest zachowany. Zatem w naszym przykładzie obydwie warstwy `GRU` uzyskają automatycznie tę maskę, ale druga z tych warstw nie zwraca sekwencji (jedynie wynik z ostatniego taktu), dlatego maska nie trafi do warstwy `Dense`. Każda warstwa może obsługiwać maskę w odmienny sposób, zasadniczo jednak zamaskowane takty są ignorowane (tzn. te takty, dla których wartość maski jest równa `False`). Na przykład gdy warstwa rekurencyjna natrafi na zamaskowany takt, to skopiuje rezultaty z poprzedniego taktu. Jeżeli maska zostanie rozprzestrzeniona aż do wyjścia (w modelach generujących na wyjściu sekwencje, czyli nie w naszym przykładzie), to zostanie zastosowana również wobec funkcji straty, zatem zamaskowane takty nie będą brane pod uwagę podczas obliczania funkcji straty (ich wartość funkcji straty będzie równa 0).



Warstwy `LSTM` i `GRU` zawierają zoptymalizowaną implementację, dostosowaną do procesorów graficznych i wykorzystującą technologię cuDNN firmy Nvidia. Implementacja ta nie obsługuje jednak maskowania. Jeżeli Twój model zawiera maskę, to warstwy te cofną się do (znacznie wolniejszej) implementacji domyślnej. Zwróć także uwagę, że taka zoptymalizowana implementacja wymaga również stosowania domyślnych wartości w kilku hiperparametrach: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias` i `reset_after`.

Wszystkie warstwy otrzymujące maskę muszą obsługiwać maskowanie (w przeciwnym razie zostanie wyświetlony komunikat o wyjątku). Dotyczy to wszystkich warstw rekurencyjnych, a także warstwy `TimeDistributed` i kilku innych ich rodzajów. W każdej warstwie obsługującej maskowanie należy wstawić atrybut `support_masking` z wartością `True`. Jeżeli chcesz zaimplementować własną, niestandardową warstwę obsługującą maskowanie, musisz dodać argument `mask` do metody `call()` (oraz oczywiście sprawić, żeby metoda ta w jakiś sposób wykorzystywała maskę). Ponadto należy umieścić zapis `self.supports_masking = True` w konstruktorze. Jeżeli niestandardowa warstwa nie ma na początku warstwy `Embedding`, możesz wykorzystać warstwę `keras.layers.Masking` — wyznacza ona maskę `K.any(K.not_equal(inputs, 0), axis=-1)`, co oznacza, że takty, w których ostatni wymiar jest wypełniony zerami, zostaną zamaskowane w kolejnych warstwach (pod warunkiem, że występuje wymiar czasu).

⁸ Został im wyznaczony identyfikator o wartości 0 tylko dlatego, że stanowią one najczęściej występujące „słowa” w zestawie danych. Prawdopodobnie warto zagwarantować, że tokeny uzupełniające będą zawsze reprezentowane przez identyfikator 0, nawet wtedy, gdy nie będą występować najczęściej.

Mechanizmy warstw maskujących i automatycznej propagacji masek najlepiej działają w prostych modelach Sequential. Nie zawsze sprawdzają się one w bardziej skomplikowanych modelach, na przykład takich, w których trzeba mieszać warstwy Conv1D z warstwami rekurencyjnymi. W takich sytuacjach musisz obliczyć maskę jawnie i przekazać ją do odpowiednich warstw za pomocą interfejsu funkcyjnego albo podklasowego. Na przykład poniżej definiuję model identyczny z poprzednim, tym razem jednak został zbudowany za pomocą interfejsu funkcyjnego i pozwala na ręczną obsługę maskowania:

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

Po wytrenowaniu modelu przez kilka epok zacznie on sobie całkiem nieźle radzić z rozpoznawaniem sentymentów w recenzjach. Jeżeli skorzystasz z wywołania zwrotnego TensorBoard(), możesz zwizualizować wektory właściwościowe uzyskiwane w trakcie uczenia: fascynująco wygląda proces stopniowego grupowania takich wyrazów jak „awesome” czy „amazing” (mających wydłużek pozytywny) po jednej stronie przestrzeni właściwościowej i „awful” czy „terrible” (o wydłużku negatywnym) po jej drugiej stronie. Niektóre wyrazy nie są uznawane za tak pozytywne, jak na to zasługują (przynajmniej w tym modelu) — np. „good”, prawdopodobnie dlatego, że w wielu recenzjach negatywnych pojawia się w wyrażeniu „not good”. To zdumiewające, że model był w stanie opanować przydatne reprezentacje właściwościowe słów na podstawie zaledwie 25 000 recenzji. Wyobraź sobie, jaki byłby skuteczny, gdybyśmy dostarczyli mu miliard recenzji do nauki! Niestety nie mamy dostępu do takiej ilości danych, ale może bylibyśmy w stanie wykorzystać reprezentacje właściwościowe słów wyuczone na jakimś innym korpusie tekstu (np. na artykułach Wikipedii), nawet jeżeli nie są to recenzje filmów? Jak by nie patrzeć, wyraz „amazing” zasadniczo ma taki sam wydłużek bez względu na to, czy mówimy o filmach czy o czymkolwiek innym. Ponadto być może reprezentacje właściwościowe nadawałyby się do analizy sentymentów nawet wtedy, gdy zostały stworzone do innego zadania: słowa „awesome” i „amazing” mają podobne znaczenie, dlatego istnieje duża szansa, że zostaną zgrupowane w przestrzeni właściwościowej również w innych zadaniach (np. przewidywania następnego słowa w sekwencji). Jeżeli skupienia tworzą wszystkie słowa pozytywne i negatywne, to taki podział pomoże w analizie sentymentów. Zamiast więc wykorzystywać tak wiele parametrów do wyznaczania reprezentacji właściwościowych słów możemy po prostu spróbować użyć gotowych wektorów właściwościowych.

Korzystanie z gotowych reprezentacji właściwościowych

Projekt TensorFlow Hub ułatwia wykorzystywanie gotowych składników we własnych modelach. Składniki te są nazywane **modułami**. Wystarczy przejrzeć repozytorium TF Hub (<https://tfhub.dev/>), znaleźć potrzebny moduł, skopiować jego kod do swojego projektu, a moduł zostanie automatycznie pobrany wraz z wyuczonymi wagami i dołączony do modelu. Proste!

Użyjmy na przykład modułu reprezentacji właściwościowych zdań nnlm-en-dim50 (pierwsza wersja) w naszym modelu analizy sentymentów:

```

import tensorflow_hub as hub

model = keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
                  dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])

```

Warstwa hub.KerasLayer pobiera moduł z podanego adresu URL. Wybrany przez nas moduł jest **koderem zdań** (ang. *sentence encoder*): przyjmuje na wejściułańcuchy znaków i koduje każdy z nich jako pojedynczy wektor (w tym przypadku 50-wymiarowy). Poza naszym wzrokiem przeprowadzana jest analiza składni tychłańcuchów (wyrazy są rozdzielane na podstawie odstępów), a każde słowo jest osadzane za pomocą macierzy właściwościowej wyuczonej na olbrzymim korpusie Google News 7B (składa się on z 7 miliardów wyrazów!). Kolejnym etapem jest obliczenie średniej ze wszystkich reprezentacji właściwościowych słów, w wyniku czego uzyskujemy reprezentację właściwościową zdania⁹. Możemy teraz dodać dwie proste warstwy Dense, aby stworzyć dobry model analizy sentymentów. Domyślnie warstwa hub.KerasLayer nie jest modyfikowalna, ale podczas jej tworzenia możesz wyznaczyć parametr trainable=True, dzięki czemu będzie można ją dostosować do zadania.



Nie wszystkie moduły z repozytorium TF Hub obsługują moduł TensorFlow 2, dlatego upewnij się, że wybrałaś/wybierałeś odpowiedni moduł.

Teraz wystarczy wczytać zestaw IMDb reviews, bez konieczności jego wstępnego przetwarzania (nie licząc tworzenia grup i ladowania wstępnego), i bezpośrednio wyuczyć model:

```

datasets, info = tfds.load("imdb_reviews", as_supervised=True, with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)

```

Zwróć uwagę, że w ostatnim fragmencie adresu URL, z którego pobraliśmy moduł, wyznaczyliśmy wersję pierwszą modelu. Dzięki takiemu mechanizmowi wersjonowania mamy pewność, że w przypadku wydania nowej wersji modułu nie zepsuje nam ona modelu. Dogodnie dla nas po wprowadzeniu tego adresu w przeglądarce zostanie wyświetlona dokumentacja modułu. Domyślnie pobrane pliki są przechowywane w katalogu plików tymczasowych. Być może wolisz je przenieść do bardziej trwałego miejsca, aby nie musieć ich pobierać za każdym razem, gdy będziesz czyścić albo ponownie instalować system. W tym celu wyznacz w zmiennej środowiskowej TFHUB_CACHE_DIR własny katalog (np. os.environ["TFHUB_CACHE_DIR"] = "./moje_dane_tfhub").

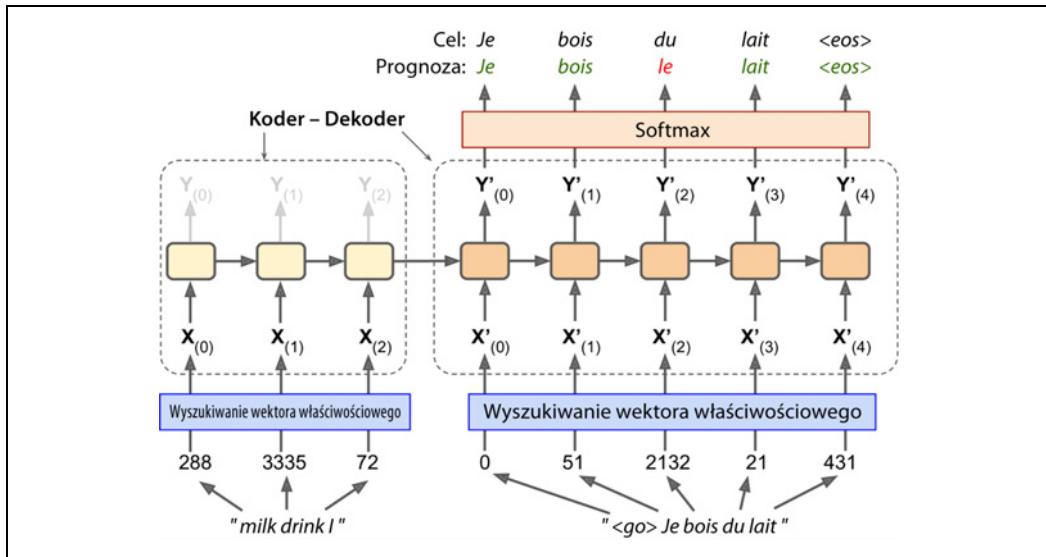
Do tej pory zajmowaliśmy się szeregiem czasowym, generowaniem tekstu za pomocą modelu Char-RNN i analizą sentymentów za pomocą modeli rekurencyjnych opartych na słowach; korzystaliśmy zarówno z własnych reprezentacji właściwościowych słów, jak i wprowadzaliśmy już wyuczone

⁹ Mówiąc dokładniej, reprezentacja właściwościowa zdania jest równa iloczynowi średniej reprezentacji właściwościowej słów i pierwiastka kwadratowego z liczby wyrazów w zdaniu. W ten sposób zostaje zrównoważony fakt, że średnia n wektorów maleje wraz ze wzrostem n .

wektory właściwościowe. Przyjrzyjmy się teraz kolejnemu istotnemu zadaniu przetwarzania języka naturalnego: **neuronowemu tłumaczeniu maszynowemu** (ang. *Neural Machine Translation* — NMT), najpierw za pomocą modelu koder – dekoder, następnie usprawnimy go przy użyciu mechanizmów uwagi, po czym zajmiemy się znakomitą architekturą modelu transformatora.

Sieć typu koder – dekoder służąca do neuronowego tłumaczenia maszynowego

Przyjrzyjmy się prostemu modelowi neuronowego tłumaczenia maszynowego (<https://arxiv.org/abs/1409.3215>)¹⁰, który przekłada angielskie zdania na język francuski (rysunek 16.3).



Rysunek 16.3. Prosty model tłumaczenia maszynowego

W skrócie: wprowadzamy do kodera angielskie zdania, a dekoder generuje przekład w języku francuskim. Zwróć uwagę, że francuskie zdania są również dostarczane na wejście dekodera, ale o jeden takt wcześniej przed angielskim zdaniem. Innymi słowy, na wejściu dekodera pojawia się słowo, które powinno być wynikiem w poprzednim taktie (bez względu na to, jaki był ten wynik). Pierwszym słowem jest znacznik określający początek zdania (ang. *start-of-sentence* — SOS). Na końcu zaś umieszczony jest znacznik oznaczający koniec zdania (ang. *end-of-sentence* — EOS).

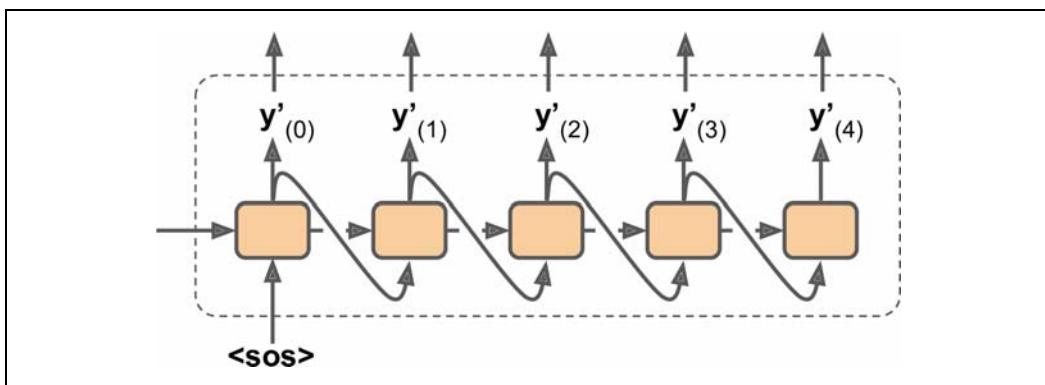
Zwrót uwagę, że szyk angielskich zdań zostaje odwrócony przed ich wczytaniem do kodera. Na przykład zdanie „I drink milk” (ja piję mleko) przyjmuje postać „milk drink I”. W ten sposób sprawiamy, że pierwszy wyraz zostanie wczytany jako ostatni, co okazuje się przydatne, ponieważ w ten sposób będzie on przetłumaczony jako pierwszy przez dekoder.

¹⁰ Ilya Sutskever i in., *Sequence to Sequence Learning with Neural Networks*, arXiv preprint arXiv:1409.3215 (2014).

Każdy wyraz jest na początku reprezentowany przez identyfikator (np. 288 dla słowa „milk”). Następnie warstwa embedding zwraca wektor właściwościowy. To właśnie te wektory są wczytywane do kodera i dekodera.

W każdym taktie dekoder wyświetla wynik dla każdego słowa w wokabularzu wyjściowym (tzn. francuskim), a następnie funkcja softmax przekształca te wyniki w prawdopodobieństwa. Na przykład w pierwszym taktie wyraz „Je” może mieć prawdopodobieństwo 20%, „Tu” — 1% itd. Wyraz o największym prawdopodobieństwie zostaje podany na wyjście. Mechanizm ten bardzo przypomina regularne zadanie klasyfikacji, dlatego możemy wyuczyć ten model za pomocą funkcji straty "sparse_categorical_crossentropy", podobnie jak to robiliśmy w modelu Char-RNN.

Zwróć uwagę, że na etapie wnioskowania (już po zakończeniu nauki) nie mamy do dyspozycji zdania docelowego, które byśmy przekazali dekoderowi. Zamiast tego wprowadzamy po prostu wyraz, który stanowił wynik w poprzednim taktie, co widzimy na rysunku 16.4 (potrzebny jest do tego niepokazany na rysunku węzeł wyszukiwania wektora właściwościowego).



Rysunek 16.4. Dostarczanie słowa będącego wynikiem poprzedniego taktu jako wejście na etapie wnioskowania

Mamy teraz pełnygląd sytuacji. Jeżeli jednak chcesz zaimplementować ten model, musisz zwrócić uwagę na kilka szczegółów:

- Do tej pory zakładaliśmy, że wszystkie sekwencje wejściowe (do kodera i dekodera) są stałe długości. Oczywiście jednak istnieją zdania o różnych długościach. Standardowe tensory mają ustalone wymiary, dlatego mogą przechowywać wyłącznie zdania o takiej samej długości. Jak już wiesz, możemy rozwiązać ten problem za pomocą maskowania. Jeżeli jednak różnice w długościach zdań są bardzo duże, nie można ich przycinać tak, jak to robiliśmy przy okazji analizy sentymentów (gdyż zależy nam na pełnym, a nie uciętym przekładzie). Zamiast tego pogrupuj zdania w bloki o podobnej długości (np. zdania składające się z od jednego do sześciu wyrazów są umieszczane w jednym bloku, w następnym mieścią się zdania liczące od siedmiu do dwunastu słów itd.), a w krótszych zdaniach umieszczaj tokeny uzupełniające, dzięki czemu wszystkie zdania w bloku będą miały taką samą długość (możesz w tym celu użyć funkcji `tf.data.experimental.bucket_by_sequence_length()`). Przykładowo zdanie „I drink milk” zostaje przekształcone na „<pad> <pad> <pad> milk drink I”.
- Chcemy ignorować wszystkie składniki znajdujące się poza znacznikiem EOS, ponieważ nie mogą one stanowić części funkcji straty (muszą zostać zamaskowane). Na przykład jeśli model

umieści na wyjściu zdanie „Je bois du lait <eos> oui”, to wartość funkcji straty dla ostatniego słowa powinna zostać zignorowana.

- W przypadku dużego rozmiaru wokabularza wyjściowego (tak jak w naszym przykładzie) wyliczanie prawdopodobieństwa dla każdego dostępnego słowa byłoby bardzo czasochłonne. Jeżeli docelowy wokabularz zawiera, powiedzmy, 50 000 francuskich słów, to dekoder generowałby wektory zawierające tyle samo wymiarów, co oznacza, że zastosowanie funkcji softmax wobec takiego wektora byłoby bardzo kosztowne obliczeniowo. Jednym ze sposobów uniknięcia tego problemu jest wyszukiwanie generowanych przez model logitów dla właściwego słowa i losowej próby niewłaściwych słów, a następnie obliczenie przybliżenia funkcji straty na podstawie wyłącznie tych logitów. Taka metoda **próbkowanej funkcji softmax** (ang. *sampled softmax*) została zaproponowana w 2015 roku przez Sébastiena Jeana i in. (<https://arxiv.org/abs/1412.2007>)¹¹. W module TensorFlow możemy wykorzystać w tym celu funkcję `tf.nn.sampled_softmax_loss()` podczas uczenia, a w trakcie wnioskowania przestawić się na standardową funkcję softmax (nie można używać próbkowanej funkcji softmax podczas wnioskowania, ponieważ wymaga ona znajomości zmiennej docelowej).

Projekt TensorFlow Addons zawiera wiele narzędzi sekwencyjnych, które umożliwiają łatwe budowanie modeli typu koder – dekoder przygotowanych do środowiska produkcyjnego. Na przykład ten listing tworzy podstawowy model koder – dekoder, podobny do pokazanego na rysunku 16.3:

```
import tensorflow_addons as tfa

encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)

embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell, sampler,
                                                output_layer=output_layer)
final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

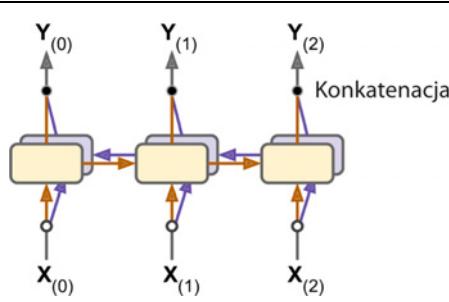
model = keras.Model(inputs=[encoder_inputs, decoder_inputs, sequence_lengths],
                     outputs=[Y_proba])
```

¹¹ Sébastien Jean i in., *On Using Very Large Target Vocabulary for Neural Machine Translation*, „Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing” 1 (2015), s. 1 – 10.

Kod jest całkiem prosty, należy jednak zwrócić uwagę na kilka rzeczy. Przede wszystkim podczas tworzenia warstwy LSTM wyznaczamy `return_state=True`, dzięki czemu otrzymujemy końcowy stan ukryty i przekazujemy go do dekodera. Korzystamy z komórki LSTM, dlatego w rzeczywistości zostają zwrocone dwa stany ukryte (krótkotrwala i długotrwały). Klasa próbująca `TrainingSampler` jest jedną z kilku dostępnych w projekcie TensorFlow Addons — ich zadaniem jest przekazanie dekoderowi w każdym kroku, jaki wynik z poprzedniego kroku powinien udawać. W trakcie wnioskowania wynikiem tym powinna być reprezentacja właściwościowa rzeczywiście wygenerowanego tokena. Podczas uczenia należy tu wstawiać wektor właściwościowy tokena z poprzedniej zmiennej docelowej — właśnie z tego powodu użyliśmy klasy `TrainingSampler`. W praktyce często warto rozpoczęć uczenie z wektorem właściwościowym zmiennej docelowej z poprzedniego taktu i stopniowo przechodzić do reprezentacji właściwościowej tokena otrzymanego w poprzednim taktie. Taką koncepcję opisali Samy Bengio i in. w publikacji z 2015 roku (<https://arxiv.org/abs/1506.03099>)¹². Klasa `ScheduledEmbeddingTrainingSampler` będzie losować pomiędzy zmienną docelową a rzeczywistym wynikiem, z prawdopodobieństwem, które można stopniowo zmieniać w trakcie uczenia.

Dwukierunkowe warstwy rekurencyjne

Standardowa sieć rekurencyjna przed wygenerowaniem wyniku sprawdza w każdym taktie jedynie poprzednie i bieżące dane wejściowe. Inaczej mówiąc, jest „przyczynowa”, co oznacza, że nie może zaglądać w przeszłość. Taki rodzaj sieci rekurencyjnej przydaje się do prognozowania szeregu czasowych, ale w wielu zadaniach przetwarzania języka naturalnego, np. w neuronowym tłumaczeniu maszynowym, należy często sprawdzać następne słowa przez zakodowaniem danego wyrazu. Dobrymi przykładami są tutaj „the Queen of United Kingdom” (królowa Zjednoczonego Królestwa), „the queen of hearts” (dama kier) i „the queen bee” (matka pszczoły): do właściwego zakodowania słowa „queen” (królowa) wymagana jest znajomość kolejnych wyrazów. Rozwiążanie to implementujemy poprzez wprowadzenie dwóch warstw rekurencyjnych przetwarzających te same dane wejściowe: jedna będzie odczytywać wyrazy od lewej do prawej, a druga odwrotnie. Następnie wystarczy połączyć ich wyniki z tego samego taktu, najczęściej za pomocą operacji konkatenacji. Jest to tak zwana **dwukierunkowa warstwa rekurencyjna** (ang. *bidirectional recurrent layer*) (rysunek 16.5).



Rysunek 16.5. Dwukierunkowa warstwa rekurencyjna

¹² Samy Bengio i in., *Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks*, arXiv preprint arXiv:1506.03099 (2015).

Aby zaimplementować dwukierunkową warstwę rekurencyjną w interfejsie Keras, umieść warstwę rekurencyjną w warstwie keras.layers.Bidirectional. Oto przykład utworzenia dwukierunkowej warstwy GRU:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```



Warstwa Bidirectional utworzy kopię warstwy GRU (ale ułożoną w przeciwnym kierunku), po czym przepuści dane przez obydwie warstwy i połączy ich wyniki. Dlatego mimo że warstwa GRU zawiera 10 jednostek, warstwa Bidirectional będzie generować w każdym takcie 20 wartości.

Przeszukiwanie wiązkowe

Założymy, że trenujesz model koder – dekoder i używasz go do przetłumaczenia francuskiego zdania „Comment vas-tu?” („Jak się masz?”) na język angielski. Masz nadzieję, że uzyskasz właściwy przekład („How are you?”), ale niestety zostaje wygenerowane tłumaczenie „How will you?”. Zagładasz do zbioru uczącego iauważasz wiele zdań, takich jak „Comment vas-tu jouer?” („Jak będziesz się bawić?”), które można przetłumaczyć na angielski jako „How will you play?”, zatem model postąpił całkiem logicznie, gdy wygenerował tłumaczenie „How will” po odczytaniu „Comment vas”. Niestety w tym przypadku model popełnił błąd, ale nie był już w stanie cofnąć się i go poprawić, więc postarał się dokończyć zdanie najlepiej, jak mógł. Z powodu zachłannego wybierania najbardziej prawdopodobnego słowa w każdym takcie wygenerował przekład daleki od ideału. Czy jesteśmy w stanie umożliwić modelowi cofanie się i naprawianie popełnionych wcześniej błędów? Jednym z najpopularniejszych rozwiązań jest **przeszukiwanie wiązkowe** (ang. *beam search*), które polega na tym, że zostaje utworzona krótka lista k najbardziej obiecujących zdań (powiedzmy, że trzech) i w każdym takcie dekoder stara się rozszerzyć je o jedno słowo, przy czym pozostawia jedynie k najbardziej prawdopodobnych zdań. Parametr k jest nazywany **szerokością wiązki** (ang. *beam width*).

Założymy na przykład, że do przetłumaczenia zdania „Comment vas-tu?” wykorzystujemy model zawierający algorytm przeszukiwania wiązkowego o szerokości wiązki równej 3. W pierwszej fazie dekodera model będzie obliczał szacowane prawdopodobieństwo dla każdego możliwego słowa. Powiedzmy, że trzema najbardziej prawdopodobnymi słowami są „How” (prawdopodobieństwo rzędu 75%), „What” (3%) i „You” (1%). W tym momencie jest to nasza krótka lista. Tworzymy następnie trzy kopie modelu i wykorzystujemy je do wyszukania następnego słowa w każdym z trzech zdań. Każdy z tych modeli oszacuje prawdopodobieństwo dla każdego wyrazu w wokabularzu. Pierwszy model spróbuje określić następne słowo w zdaniu „How” i zapewne oszacuje 36% prawdopodobieństwa dla słowa „will”, 32% dla słowa „are”, 16% dla słowa „do” itd. Zwróć uwagę, że mamy tu do czynienia z prawdopodobieństwami **warunkowymi**, ponieważ jest narzucony początek zdania w postaci słowa „How”. Drugi model spróbuje dokończyć zdanie „What” i być może oszacuje prawdopodobieństwo rzędu 50%, że następnym słowem będzie „are” itd. Przy założeniu, że wokabularz składa się z 10 000 wyrazów, każdy model oszacuje właśnie tyle prawdopodobieństw.

Następnie obliczamy prawdopodobieństwa dla każdego z 30 000 dwuwyrzazowych zdań rozpatrywanych przez modele ($3 \times 10\ 000$). Robimy to poprzez pomnożenie szacowanego prawdopodobieństwa warunkowego występowania każdego słowa przez szacowane prawdopodobieństwo wy-

stępowania zdania, które to słowo uzupełnia. Na przykład szacowane prawdopodobieństwo dla zdania „How” wyniosło 75%, podczas gdy szacowane prawdopodobieństwo występowania słowa „will” (przy założeniu, że pierwszym wyrazem jest „How”) wyniosło 36%, zatem szacowane prawdopodobieństwo występowania zdania „How will” wynosi $75\% \times 36\% = 27\%$. Po obliczeniu prawdopodobieństwa dla wszystkich 30 000 dwuwyrzazowych zdań pozostawiamy tylko trzy najbardziej prawdopodobne. Zapewne wszystkie trzy będą zaczynały się słowem „How”: „How will” (27%), „How are” (24%) i „How do” (12%). Na razie zdanie „How will” jest najbardziej prawdopodobne, ale zdanie „How are” nie zostało wykluczone.

I znowu powtarzamy cały proces: wykorzystujemy trzy modele do przewidzenia następnego słowa w każdym z trzech zdań i obliczamy prawdopodobieństwa dla wszystkich 30 000 trójwyrzazowych zdań. Być może teraz najbardziej prawdopodobne są zdania „How are you” (10%), „How do you” (8%) i „How will you” (2%). W kolejnym przebiegu możemy otrzymać zdania „How do you do” (7%), „How are you <eos>” (6%) i „How are you doing” (3%). Zauważ, że zdanie „How will” zostało wyeliminowane i pozostały trzy całkowicie logiczne przekłady. Usprawniliśmy wydajność modelu koder – dekoder bez konieczności jego dodatkowego uczenia, a po prostu użyliśmy go nieco mądrzej.

Możemy dość prosto zaimplementować model wykorzystujący przeszukiwanie wiązkowe za pomocą projektu TensorFlow Addons:

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width, output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

Najpierw tworzymy klasę BeamSearchDecoder, która otacza wszystkie kopie dekodera (w tym przypadku mamy do czynienia z dziesięcioma kopiami). Następnie dodajemy po jednej kopią stanu końcowego kodera dla każdej kopii dekodera i przekazujemy jej stany do dekodera wraz z tokenami początkowymi i końcowymi.

W ten sposób możesz uzyskać całkiem dobre przekłady dość krótkich zdań (zwłaszcza jeżeli wykorzystujesz gotowe reprezentacje właściwościowe słów). Niestety model ten nie nadaje się do tłumaczenia długich zdań. Jak łatwo się domyślić, przyczynę stanowi ograniczona pamięć krótkotrwała sieci rekurencyjnych. Rozwiązaniem tego problemu są rewolucyjne **mechanizmy uwagi** (ang. *attention mechanisms*).

Mechanizmy uwagi

Spójrz na ścieżkę łączącą słowo „milk” z jego przekładem „lait” na rysunku 16.3: jest ona całkiem dłużą! Oznacza to, że reprezentacja tego wyrazu (a także pozostałych) musi być przechowywana przez wiele etapów, zanim zostanie rzeczywiście użyta. Czy nie można jakoś skrócić tej ścieżki?

Odpowiedź na to pytanie stanowi sedno przełomowej publikacji z 2014 roku (<https://arxiv.org/abs/1409.0473>),¹³ której autorami są Dzmitry Bahdanau i in. Zaprezentowali oni technikę umożliwiającą dekoderowi koncentrowanie się na odpowiednich słowach (zakodowanych przez koder) w każdym taktie. Na przykład w taktie, w którym dekoder musi przesłać na wyjście słowo „lait”, skoncentruje swoją uwagę na wyrazie „milk”. Oznacza to, że teraz ścieżka od wyrazu wejściowego do jego przekładu jest znacznie krótsza, zatem ograniczenia pamięci krótkotrwałej, cechujące sieci RNN, mają teraz znacznie mniejszy wpływ na wydajność modelu. Mechanizmy uwagi zrewolucjonizowały zadania neuronowego tłumaczenia maszynowego (a w zasadzie całą dziedzinę przetwarzania języka naturalnego) i znacznie usprawniły nowoczesne modele, zwłaszcza w kwestii przetwarzania długich zdań (składających się z ponad 30 słów)¹⁴.

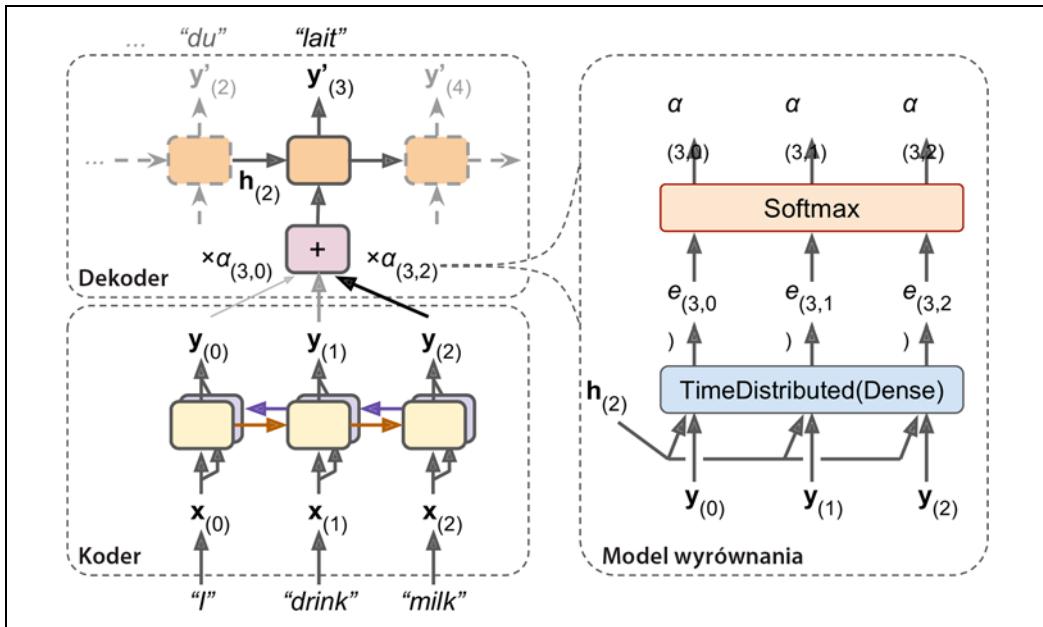
Rysunek 16.6 przedstawia strukturę modelu (jak się przekonasz, została ona nieco uproszczona). Po lewej widoczne są koder i dekoder. Przesyłamy tu nie tylko końcowy stan ukryty kodera do dekodera (proces ten jest realizowany, chociaż nie został zaznaczony na rysunku), lecz także wszystkie sygnały końcowe. W każdym taktie komórka pamięci dekodera oblicza sumę ważoną wszystkich wyników kodera — w ten sposób zostają wyznaczone słowa, na których dekoder będzie koncentrował się w tym taktie. Waga $\alpha_{(t, i)}$ jest wagą i -tego wyjścia dekodera w t -tym taktie dekodera. Na przykład jeżeli waga $\alpha_{(3, 2)}$ jest znacznie większa od wag $\alpha_{(3, 0)}$ i $\alpha_{(3, 1)}$, to dekoder będzie poświęcał o wiele więcej uwagi drugiemu słwu („milk”) niż dwóm pozostałym, przynajmniej w tym taktie. Pozostała część dekodera działa w znany już nam sposób: w każdym taktie komórka pamięci otrzymuje omówione dane wejściowe oraz stan ukryty z poprzedniego taktu, a w końcu także (choć nie jest to pokazane na rysunku) słwo docelowe w poprzedniego taktu (lub, w fazie wnioskowania, wynik z poprzedniego taktu).

Skąd jednak biorą się wagi $\alpha_{(t, i)}$? W rzeczywistości nie ma w tym nic skomplikowanego: są one generowane przez pewną odmianę małej sieci neuronowej, zwaną **modelem wyrównania** (ang. *alignment model*) lub **warstwą uwagi** (ang. *attention layer*), która jest uczona wraz z pozostałą częścią modelu koder – dekoder. Model dopasowania został pokazany po prawej stronie rysunku 16.6. Na początku widzimy jednoneuronową, rozwijaną w czasie warstwę Dense¹⁵, do której dostarczane są wszystkie wyniki generowane przez koder wraz z poprzednim stanem ukrytym dekodera (np. $h_{(2)}$). Warstwa ta generuje wynik (energię) dla każdego sygnału wyjściowego kodera (np. $e_{(3,2)}$) — wynik ten określa stopień dopasowania każdego sygnału wyjściowego z poprzednim stanem ukrytym dekodera. Na koniec wszystkie wyniki przechodzą przez warstwę softmax, co pozwala uzyskać wagę końcową dla każdego sygnału wyjściowego kodera (np. $\alpha_{(3, 2)}$). Wszystkie wagi w danym taktie dekodera sumują się do wartości 1 (warstwa softmax nie jest rozwijana w czasie). Ten mechanizm

¹³ Dzmitry Bahdanau i in., *Neural Machine Translation by Jointly Learning to Align and Translate*, arXiv preprint arXiv:1409.0473 (2014).

¹⁴ Najczęściej wykorzystywany wskaźnikiem w przetwarzaniu języka naturalnego jest BLEU (ang. *BiLingual Evaluation Understudy*), który porównuje każdy przekład wykonany przez maszynę z kilkoma dobrymi przekładami sporządzonymi przez ludzi: określa on liczbę n -gramów (sekwencji n słów) występującą w każdym tłumaczeniu docelowym i dostosowuje wynik tak, aby brana była pod uwagę częstość utworzonych n -gramów w przekładaach docelowych.

¹⁵ Jak pamiętasz, rozwijana w czasie warstwa Dense jest równoważna standardowej warstwie Dense, którą można niezależnie stosować w każdym taktie (okazuje się jednak znacznie szybsza).



Rysunek 16.6. Neuronowe uczenie maszynowe za pomocą sieci typu kodér – dekoder zawierającej warstwę uwagi

uwagi jest nazywany **modelem uwagi Bahdanau** (ang. *Bahdanau attention*; od nazwiska pierwszego autora publikacji). Z powodu łączenia w nim wyników kodera z poprzednim stanem ukrytym dekodera czasami jest nazywany **konkatenacyjnym mechanizmem uwagi** (ang. *concatenative attention*) lub **addytywnym mechanizmem uwagi** (ang. *additive attention*).



Jeżeli zdanie wejściowe składa się z n słów i założymy, że zdanie wyjściowe będzie podobnej długości, to model ten będzie musiał obliczyć około n^2 wag. Na szczęście taka kwadratowa złożoność obliczeniowa jest policzalna, ponieważ nawet długie zdania nie zawierają tysięcy słów.

Niedługo później, bo w publikacji z 2015 roku (<https://arxiv.org/abs/1508.04025>)¹⁶ został zaproponowany inny popularny mechanizm uwagi, którego autorami są Minh-Thang Luon i in. Celem mechanizmów uwagi jest pomiar podobieństwa pomiędzy wynikami kodera a poprzednim stanem ukrytym dekodera, dlatego autorzy zaproponowali obliczenie zwykłego **iloczynu skalarnego** (zob. rozdział 4.) tych dwóch wektorów, ponieważ rozwiązywanie to często stanowi dobry wskaźnik podobieństwa, a współczesne komputery potrafią go policzyć bardzo szybko. Aby można było to zrobić, obydwa wektory muszą mieć taką samą wymiarowość. Jest to tzw. **model uwagi Luonga** (ang. *Luong attention*; również wywodzi się od nazwiska pierwszego autora publikacji), zwany również **mnożycielskim mechanizmem uwagi** (ang. *multiplicative attention*). Iloczyn skalarny daje wynik, a wszystkie wyniki (w danym takcie dekodera) przechodzą przez warstwę softmax, co pozwala uzyskać wagi (podobnie jak w modelu Bahdanau). Kolejnym zaproponowanym uproszczeniem jest wykorzystanie w danym takcie nie poprzedniego, lecz bieżącego stanu ukrytego dekodera

¹⁶ Minh-Thang Luong i in., *Effective Approaches to Attention-Based Neural Machine Translation*, „Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing” (2015), s. 1412 – 1421.

($\mathbf{h}_{(t)}$ zamiast $\mathbf{h}_{(t-1)}$), a następnie bezpośrednie obliczenie prognoz dekodera za pomocą wyjścia mechanizmu uwagi (zapisywanego jako $\tilde{\mathbf{h}}_{(t)}$; zatem nie używamy go tutaj do obliczania bieżącego stanu ukrytego dekodera). Autorzy zaproponowali również wariant mechanizmu iloczynu skalarnego, w którym wyniki kodera przechodzą najpierw przez warstwę przekształcenia liniowego (tzn. rozwijaną w czasie warstwę Dense pozbawioną członu obciążenia), zanim zostanie policzony iloczyn skalarny. Jest to tak zwana strategia „ogólna” iloczynu skalarnego. Obydwa rozwiązania porównano z konkatenacyjnym mechanizmem uwagi (z dodanym wektorem skalowania \mathbf{v}) i stwierdzono, że warianty iloczynu skalarnego spisywały się lepiej od modelu konkatenacyjnego. Z tego powodu konkatenacyjny mechanizm uwagi jest obecnie używany o wiele rzadziej. Wzory wszystkich trzech mechanizmów uwagi zostały zaprezentowane w równaniu 16.1.

Równanie 16.1. Mechanizmy uwagi

$$\tilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

$$\text{gdzie } \alpha_{(t,i)} = \frac{\exp(e_{(t,i)})}{\sum_{i'} \exp(e_{(t,i')})}$$

$$\begin{aligned} \text{i } e_{(t,i)} = & \begin{cases} \mathbf{h}_{(t)}^T \mathbf{y}_{(i)} & \text{iloczyn skalarny} \\ \mathbf{h}_{(t)}^T \mathbf{W} \mathbf{y}_{(i)} & \text{strategia ogólna} \\ \mathbf{v}^T \tanh(\mathbf{W} [\mathbf{h}_{(t)}; \mathbf{y}_{(i)}]) & \text{konkatenacja} \end{cases} \end{aligned}$$

Oto sposób wprowadzenia modelu uwagi Luonga do modelu typu koder – dekoder za pomocą projektu TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

Umieszczamy po prostu komórkę dekodera wewnętrz klasy AttentionWrapper i wybieramy odpowiedni mechanizm uwagi (w tym przypadku model Luonga).

Mechanizm uwagi wizualnej

Obecnie mechanizmy uwagi mają wiele różnych zastosowań. Jednym z pierwszych, wykraczających poza dziedzinę przetwarzania języka naturalnego, było generowanie opisów obrazów za pomocą modelu uwagi wizualnej (<https://arxiv.org/abs/1502.03044>)¹⁷: najpierw splotowa sieć neuronowa przetwarza obraz i generuje pewne mapy cech, a następnie dekoder rekurencyjny, zaopatrzony w mechanizm uwagi, tworzy podpis, sekwencyjnie po jednym słowie. W każdym takcie dekodera (przy

¹⁷ Kelvin Xu i in., Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, „Proceedings of the 32nd International Conference on Machine Learning” (2015), s. 2048 – 2057.

każdym wyrazie) model uwagi koncentruje się na właściwym obszarze obrazu. Na przykład dla zdjęcia widocznego na rysunku 16.7 model wygenerował podpis „A woman is throwing a frisbee in a park” („Kobieta rzuca frisbee w parku”); widzimy także, na jakim obszarze obrazu skoncentrował się model podczas wybierania słowa „frisbee” — jak widać, jego uwagę pochłaniała głównie ta zabawka.



Rysunek 16.7. Mechanizm uwagi wizualnej: obraz wejściowy (po lewej) i obszar zainteresowania modelu przed wybraniem słowa „frisbee” (po prawej)¹⁸

Wyjaśnialność

Jedną z dodatkowych zalet mechanizmów uwagi jest fakt, że dzięki nim łatwiej jest zrozumieć, w jaki sposób model uzyskał określony wynik. Jest to tak zwana **wyjaśnialność** (ang. *explainability*). Jest ona przydatna zwłaszcza wtedy, gdy model popełni błąd — na przykład jeżeli zdjęcie psa brodzącego po śniegu zostanie podpisane jako „wilk brodzący po śniegu”, to możemy sprawdzić, na jakim elemencie model koncentrował uwagę, gdy wybierał słowo „wilk”. Być może skupiał się nie tylko na psie, ale także na śniegu, co może stanowić wyjaśnienie: możliwe, że model nauczył się rozróżniać psy od wilków na podstawie ilości śniegu. Możesz rozwiązać ten problem poprzez wprowadzanie większej liczby zdjęć wilków w środowisku pozbawionym śniegu oraz psów biegających po zaśnieżonym terenie. Przykład ten pochodzi ze znakomitej publikacji z 2016 roku (<https://arxiv.org/abs/1602.04938>)¹⁹, której autorzy, Marco Tulio Ribeiro i in., wykorzystali wyjaśnialność w inny sposób: poznawanie interpretowalnego modelu lokalnie wokół prognozy klasyfikatora.

W pewnych zastosowaniach wyjaśnialność nie stanowi jedynie narzędzia pozwalającego naprawić model, lecz może być również wymogiem prawnym (np. w systemie przyznającym pożyczki).

Mechanizmy uwagi są tak potężne, że w zupełności wystarczą do tworzenia nowoczesnych modeli.

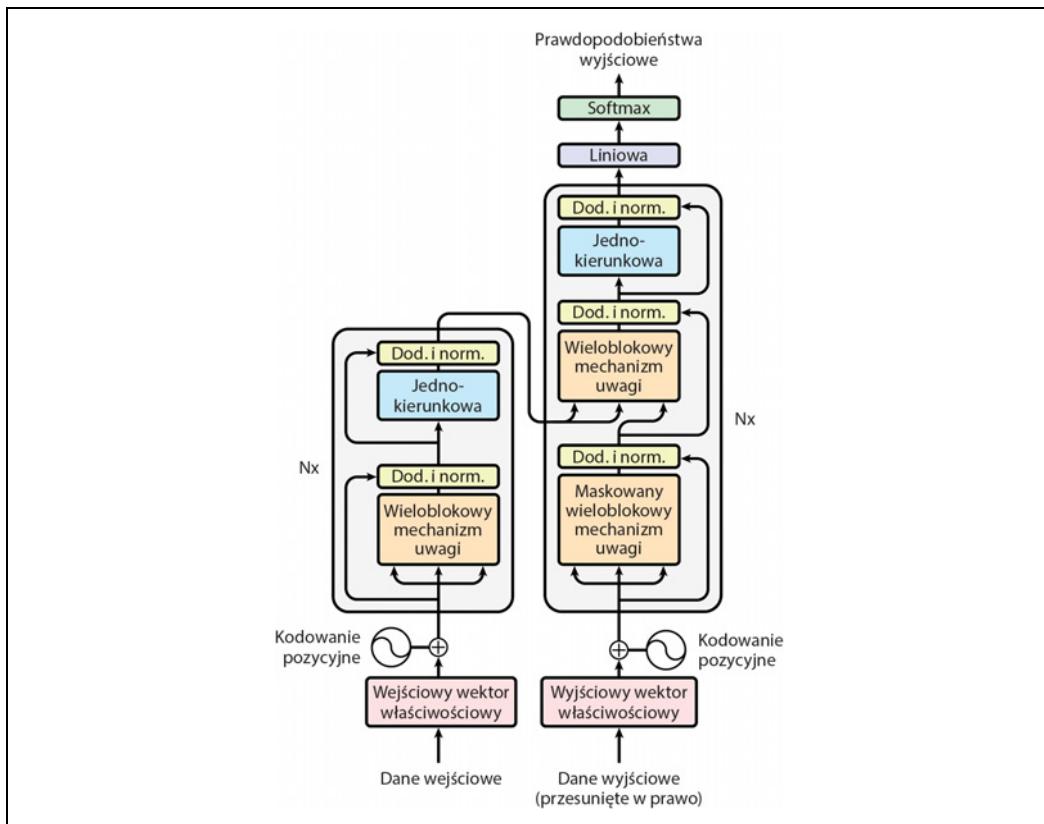
¹⁸ Jest to część rysunku 3. z oryginalnej publikacji. Rysunek ten został umieszczony w niniejszej książce za zgodą autorów.

¹⁹ Marco Tulio Ribeiro i in., „Why Should I Trust You?": Explaining the Predictions of Any Classifier, „Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining” (2016), s. 1135 – 1144.

Licz się tylko uwaga, czyli architektura transformatora

W przełomowej publikacji z 2017 roku (<https://arxiv.org/abs/1706.03762>)²⁰ zespół badaczy z firmy Google zasugerował, że „licz się tylko uwaga”. Autorzy zdołali utworzyć strukturę nazwaną **transformatorem** (ang. *transformer*), która znacznie usprawniła nowoczesne modele neuronowe do uczenia maszynowego bez konieczności wprowadzania jakichkolwiek warstw rekurencyjnych lub splotowych²¹ — stosowane są wyłącznie mechanizmy uwagi (a także warstwy reprezentacji właściwościowych, gęste, normalizacji i kilka innych składników). Ponadto struktura ta okazała się bardzo szybka i łatwa w zrównolegleniu, więc można ją było wytrenować za ułamek kosztów (i w ułamku czasu) potrzebnych do przygotowania wcześniejszych modeli.

Architektura transformatora została przedstawiona na rysunku 16.8.



Rysunek 16.8. Architektura transformatora²²

²⁰ Ashish Vaswani i in., *Attention Is All You Need*, „Proceedings of the 31st International Conference on Neural Information Processing Systems” (2017), s. 6000 – 6010.

²¹ Znajdziemy tu rozwijane w czasie warstwy Dense, zatem można się spierać, czy wykorzystywane są tu jednowymiarowe warstwy splotowe o rozmiarze jądra równym 1.

²² Jest to rysunek 1. z oryginalnej publikacji. Rysunek ten został umieszczony w niniejszej książce za pozwoleniem autorów.

Przeanalizujmy ten schemat:

- Po lewej stronie widzimy koder. Podobnie jak wcześniej, przyjmuje on na wejściu grupę zdań reprezentowanych jako sekwencje identyfikatorów wyrazów (wymiary sygnału wejściowego to [rozmiar grupy, maksymalna długość zdania wejściowego]) i koduje każde słowo jako 512-wymiarową reprezentację (zatem sygnał wyjściowy kodera ma wymiary [rozmiar grupy, maksymalna długość zdania wejściowego, 512]). Zwróć uwagę, że górna struktura kodera jest powtarzana N -krotnie (w oryginalnym artykule $N = 6$).
- Po prawej stronie znajduje się dekoder. W czasie uczenia przyjmuje on na wejściu zdanie docelowe (również reprezentowane jako sekwencja identyfikatorów słów) przesunięte o jeden takt w prawo (tzn. na początku zdania zostaje wstawiony znacznik początku sekwencji). Odbiera także wyniki kodera (strzałki biegące w prawo). Zauważ, że górna część dekodera także jest powtarzana N razy, a końcowe wyniki kodera są przekazywane każdemu z N poziomów dekodera. Podobnie jak wcześniej, dekoder oblicza prawdopodobieństwo występowania każdego słowa w każdym taktie (wymiary sygnału wyjściowego to [rozmiar grupy, maksymalna długość zdania wyjściowego, długość wokabularza]).
- W fazie wnioskowania nie możemy dostarczać modelowi zmiennych docelowych, zatem przekazujemy mu wyrazy otrzymane wcześniej na wyjściu (począwszy od tokena początku sekwencji). Należy więc ciągle wywoływać model i za każdym razem przewidywać po jednym słowie (które będą w następnych przebiegach przekazywane dekoderowi aż do momentu natrafienia na znacznik końca sekwencji).
- Jeżeli przyjrzesz się uważniej, dostrzeżesz, że większość składników jest już nam dobrze znaną: występują tu dwie warstwy wektorów właściwościowych, $5 \times N$ połączeń pomijających, po których występują warstwa normalizująca, $2 \times N$ modułów „jednokierunkowych”, z których każdy zawiera dwie warstwy gęste (pierwsza ma funkcję ReLU, druga jest pozbawiona funkcji aktywacji), a warstwa wyjściowa to warstwa gęsta wykorzystująca funkcję softmax. Wszystkie te warstwy są rozwijane w czasie, zatem każde słowo jest przetwarzane niezależnie od pozostałych. Ale w jaki sposób można przetłumaczyć zdanie, gdy patrzy się tylko na jedno słowo? Właśnie tutaj przydają się nowe składniki:
 - Warstwa **wieloblokowego mechanizmu uwagi** (ang. *multi-head attention*) koduje powiązania każdego słowa z pozostałymi wyrazami w zdaniu i poświęca więcej uwagi najistotniejszym wyrazom. Na przykład wynik tej warstwy uzyskany dla słowa „Queen” w zdaniu „They welcomed the Queen of the United Kingdom” („Powitali królową Zjednoczonego Królestwa”) będzie zależeć od wszystkich wyrazów tworzących to zdanie, ale prawdopodobnie model poświęci więcej uwagi słowom „United” i „Kingdom” niż „They” albo „welcomed”. Mechanizm ten zwany jest **samouwagą** (ang. *self-attention*; zdanie samo zwraca uwagę na siebie). Niebawem omówię to zjawisko. Warstwa **maskowanego wieloblokowego mechanizmu uwagi** (ang. *masked multi-head attention*) dekodera pełni tę samą funkcję, ale każde słowo może tu towarzyszyć wyłącznie wyrazom znajdującym się przed nim. Góra warstwa wieloblokowego mechanizmu uwagi jest miejscem, w którym dekoder zwraca uwagę na słowa tworzące zdanie wejściowe. Na przykład będzie on prawdopodobnie koncentrował się na słowie „Queen” ze zdania wejściowego podczas generowania tłumaczenia tego wyrazu.

– **Pozycyjne wektory właściwościowe** (ang. *Positional Embeddings* — PE) to nic innego jak wektory gęste (bardzo przypominają reprezentacje właściwościowe słów) symbolizujące położenie wyrazu w zdaniu. W każdym zdaniu dodawany jest n -ty pozycyjny wektor właściwościowy do reprezentacji właściwościowej n -tego słowa. W ten sposób model uzyskuje dostęp do położenia każdego wyrazu, co jest niezbędne, gdyż warstwy wieloblokowego mechanizmu uwagi nie interesują się kolejnością lub położeniem słów, lecz jedynie analizują związki pomiędzy nimi. Wszystkie warstwy są rozwijane w czasie, więc nie mają żadnej możliwości określania pozycji każdego słowa (ani względnej, ani bezwzględnej). Oczywiście względne i bezwzględne położenia wyrazów są istotną kwestią, dlatego musimy w jakiś sposób przekazać te informacje transformatorowi, a pozycyjne wektory właściwościowe okazują się dobrym rozwiązańiem.

Przyjrzymy się nieco uważniej obydwu rodzajom tych nowych elementów, począwszy od pozycyjnych wektorów właściwościowych.

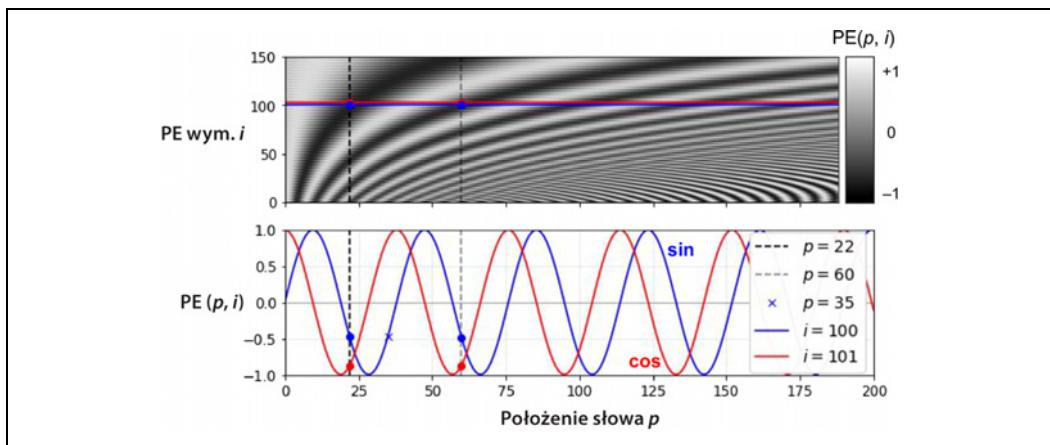
Pozycyjne wektory właściwościowe

Pozycyjny wektor właściwościowy jest wektorem gęstym, który koduje położenie słowa w zdaniu: i -ty pozycyjny wektor właściwościowy zostaje dodany do reprezentacji właściwościowej i -tego słowa w zdaniu. Model jest w stanie samodzielnie poznać te pozycyjne wektory właściwościowe, ale autorzy artykułu woleli wykorzystywać ustalone wektory pozycyjne, definiowane za pomocą funkcji sinus i cosinus o różnych częstotliwościach. Pozycyjna macierz właściwościowa P została zdefiniowana w równaniu 16.2 i jest widoczna w dolnej części rysunku 16.9 (jej transpozycja), gdzie $P_{i,j}$ to i -ta składowa wektora właściwościowego dla słowa umieszczonego na p -tej pozycji w zdaniu.

Równanie 16.2. Sinusoidalne/cosinusoidalne pozycyjne wektory właściwościowe

$$P_{p,2i} = \sin(p / 10000^{2i/d})$$

$$P_{p,2i+1} = \cos(p / 10000^{2i/d})$$



Rysunek 16.9. Macierz sinusoidalnych/cosinusoidalnych pozycyjnych wektorów właściwościowych (transponowana na górze); z zaznaczonymi dwiema wartościami i (na dole)

Rozwiążanie to zapewnia taką wydajność jak wyuczone wektory pozycyjne, ale można je rozszerzyć na dowolnie długie zdania i to decyduje o jego przewadze. Po dołączeniu wektorów pozycyjnych do reprezentacji właściwościowych słów pozostała część modelu uzyskuje dostęp do bezwzględnej pozycji każdego wyrazu w zdaniu, ponieważ każde położenie jest reprezentowane przez niepowtarzalny wektor właściwościowy (np. pozycyjny wektor właściwościowy dla słowa umieszczonego na 22. pozycji w zdaniu jest symbolizowany na rysunku 16.9 pionową linią przerwaną; widać wyraźnie, że jest ona niepowtarzalna dla tej pozycji). Ponadto dzięki wyborowi funkcji oscylacyjnych (sinusa i cosinusa) możliwe jest również wyuczenie położen względnych przez model. Na przykład wyrazy rozdzielone 38 słowami (np. na pozycjach $p = 22$ i $p = 60$) mają zawsze takie same wartości pozycyjne w wymiarach właściwościowych $i = 100$ oraz $i = 101$, co widać na rysunku 16.9. Dlatego właśnie dla każdej częstotliwości wymagana jest zarówno funkcja sinus, jak i cosinus, bo gdybyśmy używali tylko funkcji sinus (niebieska fala w $i = 100$), to model nie byłby w stanie rozróżnić pozycji $p = 25$ i $p = 35$ (krzyżyk na rysunku).

Moduł TensorFlow nie zawiera zaimplementowanej warstwy PositionalEmbedding, ale jej utworzenie nie stanowi większego problemu. Z powodów wydajnościowych wstępnie obliczamy pozycyjną macierz właściwościową w konstruktorze (musimy więc znać maksymalną długość zdania, max_steps, i liczbę wymiarów dla reprezentacji każdego słowa, max_dims). Następnie metoda call() skraca tę macierz do rozmiaru danych wejściowych i dodaje ją do sygnału wejściowego. Podczas tworzenia pozycyjnej macierzy właściwościowej dodaliśmy pierwszy wymiar o rozmiarze 1, dlatego reguły transmitowania sprawią, że macierz zostanie dodana do każdego zdania w danych wejściowych:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims musi być wartością parzystą
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty([1, max_steps, max_dims])
        pos_emb[0, :, ::2] = np.sin(p / 10000**((2 * i) / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**((2 * i) / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Możemy teraz utworzyć pierwsze warstwy transformatora:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Przyjrzymy się głównemu elementowi modelu transformatora: warstwie wieloblokowego mechanizmu uwagi.

Wieloblokowy mechanizm uwagi

Aby zrozumieć, jak działa warstwa wieloblokowego mechanizmu uwagi, trzeba najpierw poznać stanowiącą jej podstawę warstwę **skalowanego mechanizmu uwagi wykorzystującego iloczyn skalarny** (ang. *scaled dot-product attention*). Założymy, że koder przeanalizował zdanie wejściowe „They played chess” („Oni grali w szachy”) i wywnioskował, że „They” jest podmiotem, a „played” orzeczeniem, więc zakodował tę informację w reprezentacjach tych słów. Założymy teraz, że dekoder przetłumaczył już podmiot i „stwierdza”, że należy przełożyć teraz orzeczenie. W tym celu musi je pobrać ze zdania wejściowego. Proces ten przypomina przeszukiwanie wokabularza: można to porównać do sytuacji, w której koder tworzy wokabularz {„podmiot”: „They”, „orzeczenie”: „played” itd.}, a dekoder chce sprawdzić wartość odpowiadającą kluczowi „orzeczenie”. Niestety model nie zawiera dyskretnych znaczników reprezentujących klucze (takie jak „podmiot” czy „orzeczenie”) — przechowuje on wektoryzowane reprezentacje tych pojęć (poznawane w procesie uczenia), dlatego klucz użyty do wyszukiwania (zwany **kwerendą**) nie będzie idealnie zgodny z żadnym kluczem umieszczonym w wokabularzu. Rozwiążaniem okazuje się obliczenie miary podobieństwa pomiędzy kwerendą a każdym kluczem w wokabularzu, a następnie przekształcenie uzyskanych wyników za pomocą funkcji softmax w wagi dające po dodaniu wartość 1. Jeżeli klucz reprezentujący czasownik okaże się najbardziej podobny do kwerendy, to waga klucza będzie zbliżona do 1. Model może następnie obliczyć sumę ważoną powiązanych wartości, zatem jeśli waga klucza „orzeczenie” będzie bliska 1, to suma ważona znajdzie się bardzo blisko reprezentacji wyrazu „played”. Mówiąc krótko, możesz traktować cały ten proces jako różniczkowalne przeszukiwanie wokabularza. Wykorzystywany w transformatorze wskaźnikiem podobieństwa jest zwykły iloczyn skalarny, podobnie jak w modelu uwagi Luonga. W rzeczywistości wzory obydwu modelu są niemal takie same, nie licząc czynnika skalującego. Wzór ten został zaprezentowany (w formie wektorowej) w równaniu 16.3.

Równanie 16.3. Skalowany mechanizm uwagi wykorzystujący iloczyn skalarny

$$\text{Uwaga}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{klucze}}}}\right)\mathbf{V}$$

W tym równaniu:

- \mathbf{Q} jest macierzą zawierającą po jednym rzędzie na każdą kwerendę. Jej wymiary to $[n_{\text{kwerendy}}, d_{\text{klucze}}]$, gdzie n_{kwerendy} definiuje liczbę kwerend, d_{klucze} wyznacza liczbę wymiarów każdej kwerendy i każdego klucza.
- \mathbf{K} to macierz zawierająca po jednym rzędzie na każdy klucz. Jej wymiary to $[n_{\text{klucze}}, d_{\text{klucze}}]$, gdzie n_{klucze} określa liczbę kluczów i wartości.
- \mathbf{V} oznacza macierz zawierającą po jednym rzędzie na każdą wartość. Jej wymiary to $[n_{\text{klucze}}, d_{\text{wartości}}]$, gdzie $d_{\text{wartości}}$ określa liczbę każdej wartości.
- Wymiary macierzy $\mathbf{Q}\mathbf{K}^T$ to $[n_{\text{kwerendy}}, n_{\text{klucze}}]$: zawiera po jednym wskaźniku podobieństwa na każdą parę kwerenda – klucz. Wynik funkcji softmax ma takie same wymiary, ale wszystkie rzędy sumują się do wartości 1. Wynik końcowy ma wymiary $[n_{\text{kwerendy}}, d_{\text{wartości}}]$: na każdą kwerendę przypada jeden rząd, który reprezentuje rezultat kwerendy (sumę ważoną wartości).

- Czynnik skalujący skaluje wyniki podobieństwa w celu uniknięcia nasycenia funkcji softmax, które prowadziłoby do małych wartości gradientów.
- Możliwe jest maskowanie niektórych par klucz – wartość poprzez wprowadzenie bardzo dużej wartości ujemnej do odpowiednich wyników podobieństwa tuż przed obliczeniem funkcji softmax. Rozwiązanie to jest wykorzystywane w warstwie maskowanego wieloblokowego mechanizmu uwagi.

W koderze równanie to jest stosowane wobec każdego zdania wejściowego w grupie, gdzie macierze Q , K i V są równe liście słów w zdaniu wejściowym (zatem każde słowo w zdaniu będzie porównywane do wszystkich pozostałych słów w tym zdaniu, włącznie ze sobą samym). Podobnie w warstwie maskowanego mechanizmu uwagi w dekoderze równanie będzie stosowane wobec każdego zdania docelowego w grupie, gdzie macierze Q , K i V są równe liście słów w zdaniu docelowym, tutaj jednak zostaje wprowadzona maska po to, aby zapobiec porównywaniu dowolnego słowa z wyrazami znajdującymi się po nim (w fazie wnioskowania dekoder będzie miał dostęp jedynie do wygenerowanych już słów, a nie do przyszłych, zatem w trakcie uczenia musimy zamaskować przyszłe tokeny wynikowe). W górnej warstwie uwagi dekodera klucze K i wartości V są po prostu listą zakodowanych słów wygenerowanych przez koder, natomiast kwerendy Q stanowią listę zakodowanych słów wygenerowanych przez dekoder.

Warstwa `keras.layers.Attention` implementuje skalowany mechanizm uwagi wykorzystujący iloczyn skalarny i w wydajny sposób wykorzystuje równanie 16.3 wobec wielu zdań w grupie. Dane wejściowe są takie same jak Q , K i V plus dodatkowy (pierwszy) wymiar rozmiaru grupy.



W module TensorFlow, jeżeli tensory A i B mają więcej niż dwa wymiary, powiedzmy, odpowiednio, $[2, 3, 4, 5]$ i $[2, 3, 5, 6]$, to funkcja `tf.matmul(A, B)` będzie traktować je jako tablice 2×3 , w których każda komórka zawiera macierz, i będzie mnożyć odpowiednie macierze: macierz w i -tym rzędzie i j -tej kolumnie tensora A zostanie pomnożona przez macierz znajdująca się w i -tym rzędzie i j -tej kolumnie tensora B . Iloczynem macierzy 4×5 i 5×6 jest macierz 4×6 , więc wynikiem funkcji `tf.matmul(A, B)` będzie tablica o wymiarach $[2, 3, 4, 6]$.

Jeżeli zignorujemy połączenia pomijające, warstwy normalizujące, bloki jednokierunkowe i fakt, że występuje tu warstwa skalowanego mechanizmu uwagi wykorzystującego iloczyn skalarny zamiast warstwy wieloblokowego mechanizmu uwagi, to pozostała część modelu transformatora można zaimplementować w następujący sposób:

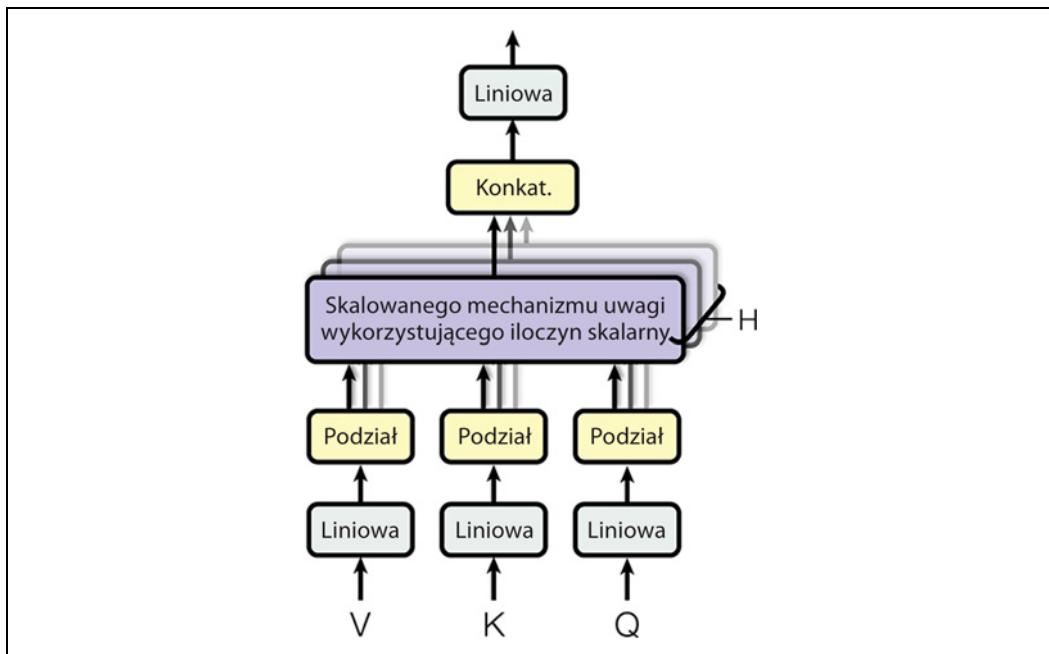
```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

Argument `use_scale=True` tworzy dodatkowy parametr, pozwalający warstwie uczyć się prawidłowo skalować wyniki podobieństwa. Jest to pewna modyfikacja w stosunku do klasycznego modelu transformatora, w którym wyniki podobieństwa są zawsze skalowane o ten sam współczynnik ($\sqrt{d_{klucze}}$). Użyty podczas tworzenia drugiej warstwy uwagi argument `causal=True` sprawia, że każdy token wynikowy będzie przyłączany do poprzednich, a nie następnych znaczników wynikowych.

Czas przejść do ostatniego elementu układanki: czym jest warstwa wieloblokowego mechanizmu uwagi? Jej struktura została zaprezentowana na rysunku 16.10.



Rysunek 16.10. Struktura warstwy wieloblokowego mechanizmu uwagi²³

Jak widać, mamy tu do czynienia jedynie z grupą warstw skalowanego mechanizmu uwagi wykorzystującego iloczyn skalarny, z których każda jest poprzedzona przekształceniem liniowym wartości, klucz i kwerend (tzn. rozwijaną w czasie warstwą Dense pozbawioną funkcji aktywacji). Wszystkie sygnały wyjściowe są ze sobą łączone i przechodzą przez ostatnią warstwę przekształcenia liniowego (również rozwijaną w czasie). Dlaczego jednak tak jest? Czy można intuicyjnie zrozumieć mechanizm działania tej struktury? Rozpatrzmy użyte już wcześniej słowo „played” (w zdaniu „They played chess”). Koder był na tyle bystry, że rozpoznał w tym słowie orzeczenie. W ramach reprezentacji wyrazu mieści się również jego położenie w tekście, co zawiadamiamy pozycyjnym wektorom właściwościowym, a także prawdopodobnie wiele innych cech wykorzystywanych w procesie tłumaczenia tekstu, na przykład informacje o formie czasowej. Mówiąc krótko, reprezentacja słowa zawiera zakodowanych wiele różnych informacji o danym wyrazie. Gdybyśmy użyli tylko jednej warstwy skalowanego mechanizmu uwagi wykorzystującego iloczyn

²³ Jest to rysunek 2. z oryginalnej publikacji. Rysunek ten został umieszczony w niniejszej książce za zgodą autorów.

skalarny, moglibyśmy przeanalizować wszystkie te parametry słowa tylko z jednej perspektywy. Z tego powodu warstwa wieloblokowego mechanizmu uwagi wykorzystuje wiele różnych przekształceń liniowych wartości, kluczy i kwerend — model może w ten sposób wprowadzać wiele różnych odwzorowań reprezentacji wyrazu na różne podprzestrzenie, z których każda koncentruje się na określonym parametrze słowa. Być może jedna z warstw liniowych będzie rzutować reprezentację słowa na przestrzeń, w której pozostawiona zostaje jedynie informacja o tym, że wyraz jest orzeczeniem, inna warstwa liniowa wyodrębnia informację o tym, że słowo wyraża czas przeszły itd. Następnie warstwy skalowanego mechanizmu uwagi wykorzystującego iloczyn skalarny implementują fazę wyszukiwania, a na koniec wszystkie rezultaty są łączone i rzutowane z powrotem na przestrzeń pierwotną.

W czasie pisania książki nie istniała jeszcze implementacja warstwy Transformer lub MultiHead →Attention w module TensorFlow 2, dostępny jest jednak znakomity samouczek opisujący tworzenie modelu transformatora uczącego się języka (<https://www.tensorflow.org/tutorials/text/transformer>). Ponadto zespół TF Hub przenosi obecnie kilka klas opartych na transformatorze do modułu TensorFlow 2 i już wkrótce powinny być one publicznie udostępnione. Tymczasem liczę na to, że przekonałem Cię o łatwości sporządzenia własnej implementacji transformatora oraz o tym, iż jest to znakomite doświadczenie!

Współczesne innowacje w modelach językowych

Rok 2018 uznano za „przełom dla zadań przetwarzania języka naturalnego na skalę rezultatów ImageNet”, postępy bowiem były w nim zdumiewające i projektowano coraz większe struktury LSTM i transformatorowe, które uczyono na olbrzymich zestawach danych. Gorąco zachęcam do zapoznania się z następującymi publikacjami z 2018 roku:

- W artykule ELMo (<https://arxiv.org/abs/1802.05365>)²⁴ Matthew Peters zaprezentował **wektory właściwościowe z modeli językowych** (ang. *Embedding from Language Models* — ELMo). Są to kontekstowe reprezentacje właściwościowe słów otrzymywane ze stanów wewnętrznych głębokiego, dwukierunkowego modelu językowego. Na przykład słowo „queen” nie będzie miało takiego samego wektora właściwościowego w wyrażeniu „Queen of the United Kingdom” jak w wyrażeniu „queen bee”.
- W publikacji ULMFiT (<https://arxiv.org/abs/1801.06146>)²⁵ Jeremy Howard i Sebastian Ruder pokazali skuteczność wstępnego uczenia nienadzorowanego w zadaniach NLP. Autorzy wytrenowali model językowy oparty na strukturze LSTM za pomocą algorytmu samouczającego się (tzn. automatycznie generującego etykiety z danych) na olbrzymim korpusie tekstowym, a następnie dostroili go do różnych zadań. Ich model uzyskał lepsze wyniki od wielu nowoczesnych architektur w sześciu zadaniach klasyfikowania tekstu, i to z dużym marginesem (w większości przypadków zmniejszał stopę błędów o 18 – 24%). Co więcej, autorzy udowodnili, że poprzez

²⁴ Matthew Peters i in., *Deep Contextualized Word Representations*, „Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies” 1 (2018), s. 2227 – 2237.

²⁵ Jeremy Howard i Sebastian Ruder, *Universal Language Model Fine-Tuning for Text Classification*, „Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics” 1 (2018), s. 328 – 339.

dostrojenie modelu za pomocą zaledwie stu oznakowanych przykładów może on osiągnąć taką samą wydajność jak model od podstaw wyuczony za pomocą 10 000 przykładów.

- W artykule GPT (<https://homl.info/gpt>)²⁶ Alec Radford wraz z innymi badaczami z zespołu OpenAI także skoncentrował się na wydajności nienadzorowanego uczenia wstępnego, wykorzystał tu jednak architekturę przypominającą transformator. Autorzy wstępnie wytrenowali dużą, ale w miarę prostą strukturę, składającą się z 12 modułów transformatora (użyli tylko warstw maskowanego wieloblokowego mechanizmu uwagi) na sporym zestawie danych i także wykorzystali algorytm samouczący się. Następnie dostroili model do różnych zadań językowych, z tym że w każdym zadaniu wprowadzali tylko drobne modyfikacje. Zadania były dość zróżnicowane — od klasyfikacji tekstu, poprzez **wynikanie** (ang. *entailment*; zadanie sprawdzające, czy ze zdania A wynika zdanie B)²⁷ i podobieństwo (np. zdanie „Ładną dzisiaj mamy pogodę” jest bardzo podobne do zdania „Dzisiaj jest słonecznie”), po odpowiadanie na pytania (po zapoznaniu się z kilkoma akapitami wyznaczającymi kontekst model musi odpowiedzieć na pytania wielokrotnego wyboru). Zaledwie kilka miesięcy później, w lutym 2019 roku, Alec Radford, Jeffrey Wu i kilku innych naukowców z zespołu OpenAI opublikowali artykuł GPT-2 (<https://homl.info/gpt2>)²⁸, w którym zaproponowali podobną, ale jeszcze większą strukturę (zawierającą ponad 1,5 miliarda parametrów!) i udowodnili, że za jej pomocą model może uzyskać dobrą wydajność w wielu zadaniach bez konieczności jego dostrajania. Jest to tak zwane **uczenie od strzału** (ang. *Zero-Shot Learning* — ZSL). Na stronie <https://github.com/openai/gpt-2> dostępna jest mniejsza wersja modelu GPT-2 (zawiera „zaledwie” 117 milionów parametrów) wraz z gotowymi wagami.
- W publikacji BERT (<https://arxiv.org/abs/1810.04805>)²⁹ Jacob Devlin wraz z innymi naukowcami z firmy Google jako kolejna osoba prezentuje skuteczność wstępnego samouczenia się na dużym korpusie tekstowym i wykorzystuje strukturę przypominającą GPT, ale zawierającą niemaskowane warstwy wieloblokowego mechanizmu uwagi (podobnie jak w koderze transformatora). Oznacza to, że model ten jest w naturalny sposób dwukierunkowy (stąd litera B w nazwie BERT, od ang. *Bidirectional Encoder Representations from Transformers* — dwukierunkowe reprezentacje kodera z transformatorów). Co ważniejsze, autor zaproponował dwa gotowe zadania, które wyjaśniają najważniejsze mocne strony modelu:
- **Maskowany model językowy** (ang. *Masked Language Model* — MDM) — każde słowo w zdaniu ma wyznaczone 15% prawdopodobieństwa, że zostanie zamaskowane, a model uczy się przewidywać zamaskowane wyrazy. Na przykład jeżeli pierwotne zdanie ma postać „Bardzo podobało jej się przyjęcie urodzinowe”, to model może otrzymać zdanie „Bardzo <maska> jej się <maska> urodzinowe” i musi przewidzieć słowa „podobało” i „przyjęcie” (pozostałe wyniki będą

²⁶ Alec Radford i in., *Improving Language Understanding by Generative Pre-Training*, (2018).

²⁷ Na przykład ze zdania „Joasia świetnie bawiła się na przyjęciu urodzinowym brata” wynika, że „Joasi podobało się przyjęcie”, ale zaprzecza ono zdaniu „Nikomu nie podobało się przyjęcie” oraz jest zupełnie niezwiązane ze zdaniem „Ziemia jest płaska”.

²⁸ Alec Radford i in., *Language Models Are Unsupervised Multitask Learners* (2019).

²⁹ Jacob Devlin i in., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, „Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies” 1 (2019).

ignorowane). Mówiąc dokładniej, każde wybrane słowo ma 80% prawdopodobieństwa, że zostanie zamaskowane, 10%, że zostanie zastąpione jakimś losowym wyrazem (po to, aby zmniejszyć rozbieżność pomiędzy uczeniem wstępny a strojeniem modelu, gdyż w czasie dostrajania model nie będzie widział znaczników <maska>), i 10%, że zostanie nietknięte (aby obciążyć model w kierunku właściwej odpowiedzi).

- **Przewidywanie następnego zdania** (ang. *Next Sentence Prediction* — NSP) — model jest uczony przewidywania, czy dwa zdania następują po sobie. Na przykład powinien przewidzieć, że zdania „Pies śpi” i „Chrapie głośno” łączy następcość, natomiast nie zachodzi ona ze zdaniami „Pies śpi” i „Ziemia kręci się wokół Słońca”. Jest to trudne zadanie i można znaczco poprawić wydajność modelu, jeżeli zostanie dostrojony do takich zadań jak odpowiadanie na pytania i wynikanie.

Jak widać, głównymi innowacjami w latach 2018 i 2019 są lepsza tokenizacja elementów słowo-twórczych, przejście z komórek LSTM na transformatory, a także uczenie wstępne uniwersalnych modeli językowych za pomocą algorytmów samouczenia się, po których wystarczy je dostroić poprzez wprowadzenie bardzo niewielkich zmian strukturalnych (ewentualnie można całkowicie pominąć ten krok). Dziedzina ta rozwija się bardzo szybko i nikt nie wie, jakie architektury będą dominować za rok. Obecnie główną rolę odgrywają transformatory, ale jutro ich miejsce mogą zająć sieci splotowe (np. sprawdź publikację z 2018 roku (<https://arxiv.org/abs/1808.03867>)³⁰ Mahy Elbayad i in., w której autorzy wykorzystują maskowane dwuwymiarowe warstwy splotowe w zadaniach sekwencyjnych). A może będą to nawet sieci rekurencyjne, jeżeli czeka je jakiś nieoczekiwany powrót (np. zajrzyj do artykułu z 2018 roku (<https://arxiv.org/abs/1803.04831>)³¹, którego autorzy, Shuai Li i in., udowadniają, że jeśli w warstwie rekurencyjnej każdy neuron będzie niezależny od pozostałych, to możliwe stanie się uczenie znacznie głębszych sieci rekurencyjnych, zdolnych do poznawania o wiele dłuższych sekwencji).

W kolejnym rozdziale przekonasz się, jak można uczyć reprezentacje głębokie w nienadzorowany sposób za pomocą autokoderów, będącymi też generować obrazy za pomocą generatywnych sieci przeciwnych, a to nie koniec niespodzianek!

Ćwiczenia

1. Porównaj stanowe i bezstanowe sieci rekurencyjne.
2. Dlaczego w zadaniach tłumaczenia automatycznego lepiej korzystać z sieci rekurencyjnych typu koder – dekoder niż z klasycznych sekwencyjnych sieci RNN?
3. Jak możemy poradzić sobie z sekwencjami wejściowymi o zmiennej długości, a jak z sekwencjami wyjściowymi o zmiennej długości?
4. Czym jest przeszukiwanie wiązkowe i jak możemy z niego skorzystać? Za pomocą jakiego narzędzia można je zaimplementować?

³⁰ Maha Elbayad i in., *Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction*, arXiv preprint arXiv:1808.03867 (2018).

³¹ Shuai Li i in., *Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN*, „Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition” (2018), s. 5457 – 5466.

- 5.** Czym jest mechanizm uwagi? Jak może się przydać?
- 6.** Która warstwa jest najważniejsza w modelu transformatora? Jakie jest jej zadanie?
- 7.** Kiedy należy korzystać z próbkowanej funkcji softmax?
- 8.** W publikacji dotyczącej komórek LSTM (<https://homl.info/93>) Hochreiter i Schmidhuber wykorzystali **osadzone gramatyki Rebera** (ang. *embedded Reber grammars*). Jest to rodzina sztucznych gramatyk generujących ciągi znaków typu „BPBTSXXVPSEPE”. Jenny Orr opracowała przystępne wprowadzenie (<http://www.willamette.edu/~gorr/classes/cs449/reber.html>) do tego tematu. Wybierz określona osadzoną gramatykę Rebera (np. pokazaną na stronie Jenny), a następnie wytrenuj sieć RNN rozpoznającą, czy ciąg znaków przestrzega jej reguł. Musisz najpierw napisać funkcję generującą grupę danych uczących składających się w połowie z próbek zgodnych z regułami wybranej gramatyki Rebera, a w połowie z przykładów nieprzestrzegających tych reguł.
- 9.** Wytrenuj model typu koder – dekoder, który przekształcała łańcuch znaków reprezentujący datę z jednego formatu w inny (np. z formatu „22 kwietnia 2019” w format „2019-04-22”).
- 10.** Zapoznaj się z samouczkiem neuronowego uczenia maszynowego wykorzystującego mechanizm uwagi opracowanym przez zespół TensorFlow (https://www.tensorflow.org/tutorials/text/nmt_with_attention).
- 11.** Użyj jednego z nowych modeli językowych (np. BERT) do wygenerowania bardziej przekonującego tekstu szekspirowskiego.

Rozwiązania ćwiczeń znajdziesz w dodatku A.

Uczenie reprezentacji za pomocą autokoderów i generatywnych sieci przeciwnych

Autokoderami nazywamy sztuczne sieci neuronowe zdolne do uczenia się gęstych reprezentacji danych wejściowych, tzw. **reprezentacji ukrytych** (ang. *latent representations*) lub **kodowań** (ang. *codings*), bez użycia jakiejkolwiek formy nadzorowania (zbiór uczący nie jest oznakowany etykietami). Kodowania te mają zazwyczaj znacznie mniejszą wymiarowość od danych wejściowych, dzięki czemu autokodery nadają się bardzo dobrze do redukowania wymiarowości (zob. rozdział 8.), zwłaszcza w zadaniach wizualizacji. Co ważniejsze, autokodery mogą również pełnić funkcje wykrywaczy cech i nieraz służą do wstępnego nienadzorowanego uczenia głębokich sieci neuronowych (zob. rozdział 11.). Potrafią też losowo generować nowe dane bardzo przypominające zbiory danych uczących — jest to tak zwany **model generatywny** (ang. *generative model*). Przykładowo możemy wyuczyć autokoder na zdjęciach twarzy, dzięki czemu będzie w stanie generować nowe twarze, jednak takie wygenerowane przykłady bywają rozmażane i niezupełnie realistyczne.

Z kolei twarze generowane przez **generatywne sieci przeciwnawne** (ang. *generative adversarial networks* — GAN) są obecnie tak wiarygodne, że trudno uwierzyć, że przedstawiają nieistniejące osoby. Możesz się o tym przekonać, jeżeli odwiedzisz stronę <https://thispersondoesnotexist.com/>, gdzie można zobaczyć twarze generowane przez współczesną architekturę GAN o nazwie StyleGAN (możesz również zajrzeć pod adres <https://thisrentaldoesnotexist.com/>, gdzie znajdziesz wygenerowane mieszkania firmy Airbnb). Sieci GAN są obecnie powszechnie stosowane w zadaniach nadrozdzielczości (zwiększenia rozdzielczości obrazu), koloryzowania (<https://github.com/jantic/DeOldify>), rozbudowanego edytowania zdjęć (np. zastępowanie mistrzów drugiego planu realistycznym tłem), przekształcania prostych szkiców w fotorealistyczne obrazy, przewidywania kolejnych klatek w obrazach wideo, dogenerowywania danych (w celu uczenia innych modeli), generowania innych typów danych (np. tekstowych, dźwiękowych czy szeregów czasowych), rozpoznawania słabych stron w innych modelach i ich niwelowania, itd.

Zarówno autokodery, jak i sieci GAN są nienadzorowane, poznają reprezentacje gęste, mogą być używane jako modele generatywne i mają wiele podobnych zastosowań. Ich mechanizmy działania są jednak bardzo odmienne:

- Autokodery uczą się po prostu kopiować swoje sygnały wejściowe na wyjścia. Pozornie wydaje się to banalne, ale jak się przekonasz, ograniczanie sieci na różne sposoby może znacznie skomplikować to zadanie. Możemy na przykład nałożyć limit na rozmiar reprezentacji ukrytych lub dodać szum do danych wejściowych i wytrenować sieć do odzyskiwania pierwotnych informacji. Dzięki tym ograniczeniom autokodery nie mogą jedynie przesyłać danych wejściowych na wyjścia, przez co są zmuszane do uczenia się skutecznych sposobów reprezentowania danych. Mówiąc krótko, kodowania stanowią produkt uboczny prób rozpoznawania przez autokodery funkcji tożsamościowej w określonych warunkach.
- Sieci GAN składają się z dwóch sieci neuronowych: **generatora** (ang. *generator*), starającego się wygenerować dane przypominające przykłady uczące, i **dyskryminatora** (ang. *discriminator*), którego zadaniem jest odróżnianie prawdziwych danych od fałszywych przykładów. Jest to bardzo charakterystyczna architektura w dziedzinie uczenia maszynowego, gdyż generator i dyskryminator współzawodniczą ze sobą w fazie uczenia: generator często jest porównywany do przestępca próbującego podrobić pieniądze, a dyskryminator do śledczego mającego na celu odróżnienie fałszywek od prawdziwych pieniędzy. **Uczenie przeciwstawnne** (ang. *adversarial learning*; uczenie współzawodniczących sieci neuronowych) jest powszechnie uznawane za jedną z najważniejszych koncepcji ostatnich lat. W 2016 roku Yann LeCun określił je nawet jako „najbardziej interesujący w ostatniej dekadzie pomysł uczenia maszynowego”.

Ten rozdział zacznąć od dokładniejszego omówienia mechanizmu działania autokoderów, a także pokażę, jak używać ich do redukowania wymiarowości, wydobywania cech, nienadzorowanego uczenia wstępnego oraz jako modeli generatywnych. W ten sposób przejdziemy płynnie do sieci GAN. Najpierw utworzymy prostą sieć GAN generującą fałszywe obrazy, przekonasz się jednak, że proces uczenia bywa dość skomplikowany. Zapoznasz się z głównymi problemami wiążącymi się z uczeniem przeciwstawnym i metodami ich zwalczania. Przejdzmy więc do autokoderów!

Efektywne reprezentacje danych

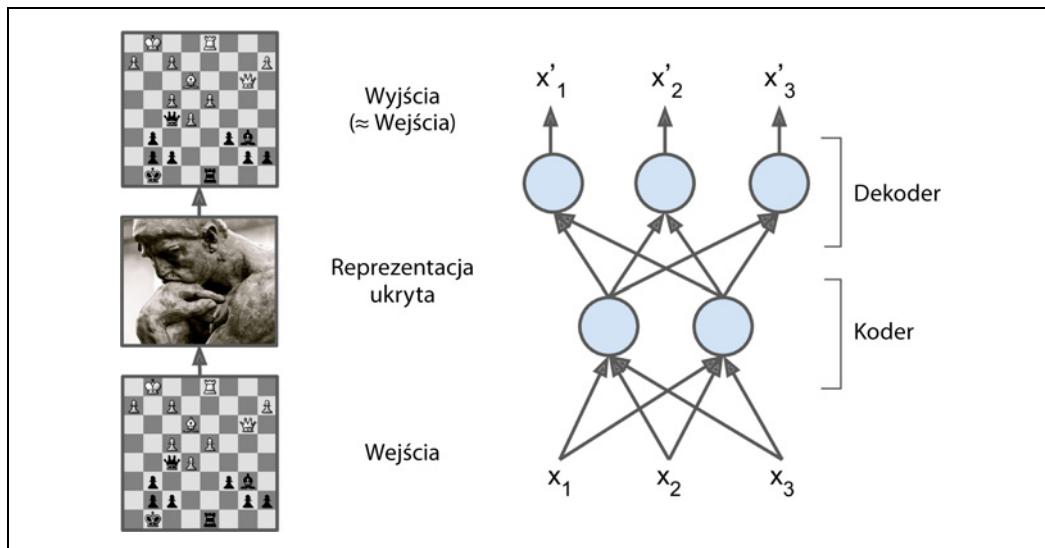
Która z poniższych sekwencji liczb jest łatwiejsza do zapamiętania:

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68,
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14?

Na pierwszy rzut oka wydaje się, że łatwiej zapamiętać pierwszy ciąg, ponieważ jest krótszy. Jeśli jednak przyjrzymy się uważniej drugiej sekwencji, zauważymy, że jest to po prostu sekwencja liczb parzystych malejących od liczby 50 do 14. Po dostrzeżeniu wzorca drugi ciąg staje się znacznie łatwiejszy do zapamiętania, ponieważ wystarczy zapamiętać ten wzorzec (malejące liczby parzyste) oraz wartości początkową i końcową (odpowiednio, 50 i 14). Zwróci uwagę, że gdybyśmy potrafili szybko i z łatwością zapamiętywać bardzo długie ciągi znaków, nie musielibyśmy sobie zwracać głowy istnieniem wzorca w drugim ciągu. Wystarczyłoby jedynie „wykuć na blachę” kolejność cyfr. To właśnie z powodu ułomności naszej pamięci warto rozpoznawać wzorce. Mam nadzieję, że dzięki takiemu wstępowi rozumiesz już, dlaczego ograniczanie autokodera podczas uczenia zmusza go do odkrywania i wykorzystywania wzorców w danych.

Związek pomiędzy pamięcią, postrzeganiem i rozpoznawaniem wzorców był dogłębnie analizowany przez Williama Chase'a i Herberta Simona na początku lat 70. ubiegłego wieku (<https://homl.info/111>)¹. Badacze zaobserwowali, że profesjonalnym zawodnikom szachowym wystarczy pięć sekund patrzenia na szachownicę, aby zapamiętać położenie wszystkich figur — wynik przez wielu ludzi uważany za nieosiągalny. Wniosek ten jednak jest prawdziwy dla realistycznego rozmieszczenia figur (spotykanego w rzeczywistych meczach), a nie ich losowego rozkładu. Mistrzowie szachowi nie mają znacznie lepszej pamięci ani od Ciebie, ani ode mnie, lecz po prostu dzięki doświadczeniu łatwiej dostrzegają wzorce w szachach. Rozpoznawanie wzorców pomaga im w zapamiętywaniu informacji.

Autokoder, podobnie jak gracze szachowi ze wspomnianego eksperymentu, patrzy na dane wejściowe, przekształca je w efektywną reprezentację ukrytą, a następnie umieszcza na wyjściu wynik przypominający (przy odrobinie szczęścia) informacje otrzymane na wejściu. Autokoder zawsze składa się z dwóch części: **kodera** (ang. *encoder*), zwanego też **siecią rozpoznawania** (ang. *recognition error*), przekształcającego dane wejściowe do postaci reprezentacji ukrytej, oraz **dekodera** (ang. *decoder*), zwanego też **siecią generatywną** (ang. *generative network*), którego zadaniem jest konwersja tej reprezentacji ukrytej na dane wyjściowe (rysunek 17.1).



Rysunek 17.1. Eksperyment z zapamiętywaniem położenia figur szachowych (po lewej) i prosty autokoder (po prawej)

Jak widać, autokoder zazwyczaj ma prawie taką samą architekturę jak perceptron wielowarstwowy (MLP; zob. rozdział 10.), a jedyna różnica polega na tym, że liczba neuronów wyjściowych musi być identyczna jak na wejściu. W omawianym przykładzie występuje tylko jedna warstwa ukryta, składająca się z dwóch neuronów (koder), a warstwa wyjściowa (dekoder) zawiera trzy neurony. Dane wyjściowe są nazywane **rekonstrukcjami** (ang. *reconstructions*), ponieważ dekoder stara się zrekonstruować dane wejściowe, natomiast funkcja kosztu zawiera **stratę rekonstrukcji** (ang. *reconstruction loss*), karzącą model w sytuacji, gdy rekonstrukcje różnią się od danych wejściowych.

¹ William G. Chase i Herbert A. Simon, *Perception in Chess*, „Cognitive Psychology” 4, no. 1 (1973), s. 55 – 81.

Wewnętrzna reprezentacja ma mniejszą wymiarowość niż dane wejściowe (jest dwu-, a nie trójwymiarowa), dlatego mówimy, że autokoder jest **niedopełniony** (ang. *undercomplete*). Niedopełniony autokoder nie może po prostu kopiować danych wejściowych do kodowań, ale musi znaleźć jakiś sposób wyświetlenia na wyjściu kopii danych wejściowych. Jest zmuszony do wyuczenia się najistotniejszych cech zawartych w danych wejściowych (i pominięcia cech nieistotnych).

Zobaczmy, jak można zaimplementować bardzo prosty niedopełniony autokoder przeprowadzający redukcję wymiarowości.

Analiza PCA za pomocą niedopełnionego autokodera liniowego

Jeżeli autokoder korzysta wyłącznie z liniowych funkcji aktywacji, a funkcją kosztu jest błąd średnio-kwadratowy (MSE), to okazuje się, że będzie on przeprowadzał analizę PCA (zob. rozdział 8.).

Za pomocą poniższego kodu tworzymy prosty autokoder liniowy przeprowadzający analizę PCA na trójwymiarowym zbiorze danych, które są rzutowane na przestrzeń dwuwymiarową:

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

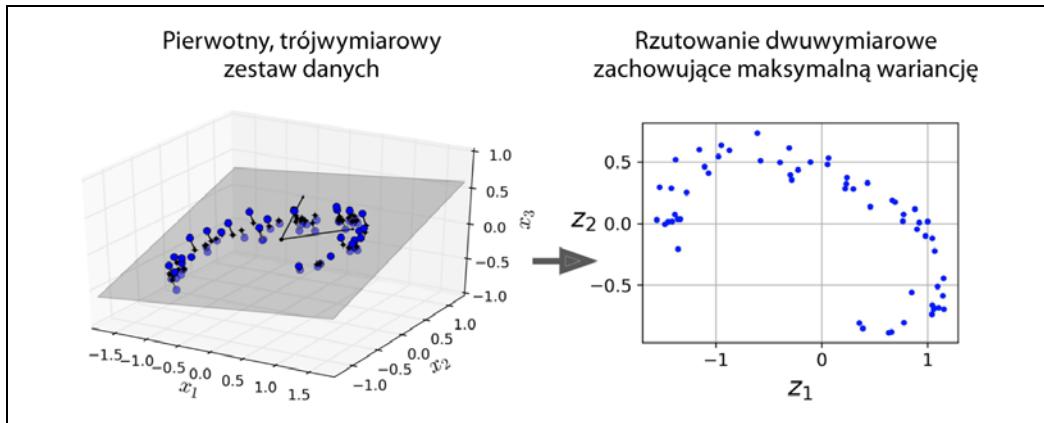
Kod ten nie różni się zbytnio od architektur perceptronów wielowarstwowych tworzonych w poprzednich rozdziałach, należy jednak zwrócić uwagę na kilka rzeczy:

- Podzieliliśmy autokoder na dwie części: koder i dekoder. Obydwa są standardowymi modelami Sequential, zawierającymi po jednej warstwie Dense, natomiast sam autokoder jest modelem Sequential, w którym po koderze występuje dekoder (jak pamiętamy, dany model może być użyty jako warstwa w innym modelu).
- Liczba wyjść jest równa liczbie wejść (np. 3).
- W celu przeprowadzenia prostej analizy PCA nie wykorzystujemy żadnej funkcji aktywacji (tzn. wszystkie neurony są liniowe), natomiast funkcją kosztu jest MSE. Niebawem zajmiemy się bardziej zaawansowanymi autokoderami.

Wyuczmy teraz model za pomocą wygenerowanego prostego zestawu danych trójwymiarowych i wykorzystajmy go do zakodowania tego zbioru danych (tzn. rzutowania go na przestrzeń dwuwymiarową):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Zwróć uwagę, że ten sam zestaw danych, `X_train`, zostaje użyty zarówno jako dane wejściowe, jak i dane docelowe. Rysunek 17.2 przedstawia pierwotny, trójwymiarowy zbiór danych (po lewej) i wynik warstwy ukrytej autokodera (tzn. warstwy kodowania; po prawej). Jak widać, autokoder znalazł



Rysunek 17.2. Analiza PCA przeprowadzona przez niedopełniony autokoder liniowy

najlepszą dwuwymiarową płaszczyznę rzutowania, zachowującą optymalną wartość wariancji (podobnie jak w analizie PCA).

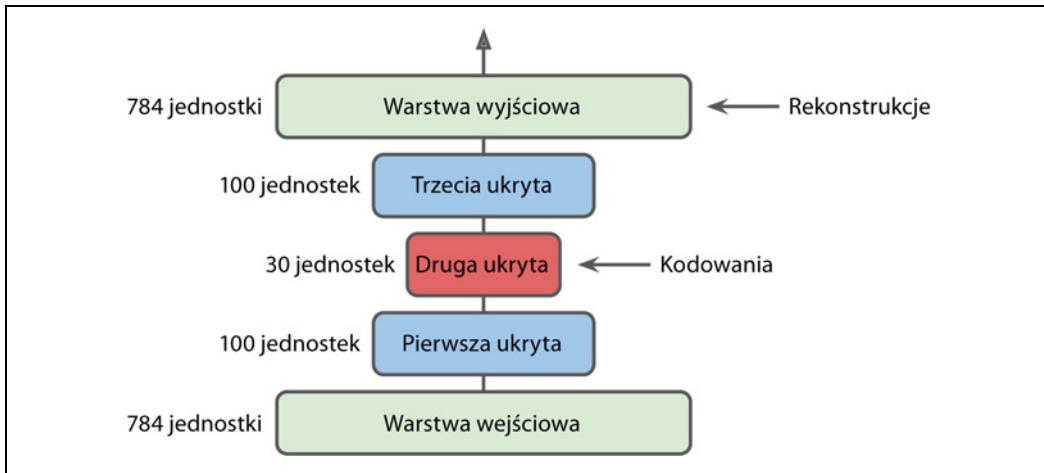


Możesz traktować autokodery jako jedną z postaci uczenia samonadzorowanego (techniki uczenia nadzorowanego wykorzystującej automatycznie generowane etykiety, w tym przypadku takie same jak dane wejściowe).

Autokodery stosowe

Autokodery, podobnie jak w przypadku innych rodzajów sieci neuronowych, również mogą mieć wiele warstw ukrytych. W takiej sytuacji są one nazywane **autokoderami stosowymi** (ang. *stacked autoencoders*) lub **głębokimi** (ang. *deep autoencoders*). Kolejne warstwy ukryte pozwalają autokoderowi uczyć się bardziej skomplikowanych kodowań. Jednak musimy uważać, aby nie stworzyć zbyt potężnego modelu. Wyobraź sobie tak wydajny autokoder, że nauczyłby się rzutować każdy przykład wejściowy do postaci dowolnej pojedynczej liczby (a dekoder uczyłby się odwrotnego rzutowania). Oczywiście taki autokoder potrafiłby doskonale rekonstruować dane uczące, ale w trakcie tego procesu nie nauczy się żadnej przydatnej reprezentacji danych (i raczej nie będzie w stanie dobrze generalizować przewidywań na nowe próbki).

Architektura autokodera stosowego najczęściej jest symetryczna względem centralnej warstwy ukrytej (kodowania). Najprościej mówiąc, przypomina ona kanapkę. Na przykład autokoder klasyfikujący obrazy MNIST (zbiór ten został opisany w rozdziale 3.) może zawierać 784 wejścia, po których następuje stuneuronowa warstwa ukryta, następnie środkowa warstwa ukryta zawierająca 30 jednostek, a po niej kolejna stuneuronowa warstwa ukryta przechodząca w warstwę wyjściową mającą 784 wyjścia. Schemat takiego stosowego autokodera został zaprezentowany na rysunku 17.3.



Rysunek 17.3. Autokoder stosowy

Implementacja autokodera stosowego za pomocą interfejsu Keras

Możesz zaimplementować interfejs stosowy w bardzo podobny sposób, jak robiliśmy to z głębokimi perceptronami wielowarstwowymi. Możemy w szczególności wykorzystać omówione w rozdziale 11. techniki uczenia sieci głębokich. Na przykład poniższy listing tworzy autokoder stosowy przeznaczony dla zestawu danych Fashion MNIST (wczytanego i znormalizowanego w sposób omówiony w rozdziale 10.) zawierający funkcję aktywacji SELU:

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                    optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                          validation_data=[X_valid, X_valid])
```

Przeanalizujmy ten kod:

- Podobnie jak wcześniej, rozdzielimy model autokodera na dwa modele podrzędne: kodera i dekodera.
- Koder przyjmuje czarno-białe obrazy o wymiarach 28×28 , spłaszcza je do postaci wektora o rozmiarze 784, po czym przetwarza te wektory przez dwie coraz mniejsze warstwy Dense (mają one, odpowiednio 100 i 30 jednostek), zawierające warstwę aktywacji SELU (możesz również dołączyć inicjalizację normalną LeCuna, ale sieć nie jest bardzo głęboka, więc nie odczujesz większej różnicy). Z każdego obrazu wejściowego dekoder generuje wektor o rozmiarze 30.

- Dekoder przyjmuje kodowania o rozmiarze 30 (wynik kodera) i przepuszcza je przez dwie coraz większe warstwy Dense (mające, odpowiednio, 100 jednostek i 784 jednostek), po czym końcowe wektory zostają przekształcone w tablice o wymiarach 28×28 , dzięki czemu sygnał wejściowy dekodera ma takie same wymiary jak sygnał wejściowy kodera.
- Podczas kompilowania autokodera stosowego używamy entropii krzyżowej jako funkcji krzyżowej, a nie błędu średniokwadratowego. Traktujemy zadanie rekonstrukcji jako problem wieloetykietowej klasyfikacji binarnej: intensywność każdego piksela reprezentuje prawdopodobieństwo określające, czy piksel ten jest czarny. Takie zdefiniowanie problemu (zrezygnowanie z zadania regresji) przyspiesza uzyskiwanie zbieżności przez model².
- Na koniec trenujemy model i używamy zestawu danych X_{train} zarówno jako danych wejściowych, jak i docelowych (w podobny sposób wykorzystujemy zestaw danych walidacyjnych X_{valid}).

Wizualizowanie rekonstrukcji

Jednym ze sposobów sprawdzenia, czy autokoder został właściwie wytrenowany, jest porównanie danych wejściowych z wyjściowymi — różnice nie powinny być zbyt duże. Wyświetlmy kilka obrazów ze zbioru walidacyjnego, a także ich rekonstrukcje:

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

Na rysunku 17.4 widzimy otrzymane obrazy.

Rekonstrukcje są rozpoznawalne, ale nieco zbyt stratne. Być może trzeba model trenować trochę dłużej albo dodać warstwy ukryte w koderze i dekoderze, ewentualnie zwiększyć rozmiar kodowań. Jeśli jednak jeszcze bardziej zwiększymy możliwości sieci, to będzie ona generować perfekcyjne rekonstrukcje, ale nie pozna żadnych przydatnych wzorców z danych. Pozostajemy na razie przy tym modelu.

² Wizja skorzystania ze wskaźnika dokładności może być kusząca, ale nie działałaby ona właściwie, ponieważ wskaźnik ten oczekuje, że dla każdego piksela etykieta będzie przyjmowała wartość 0 lub 1. Można z łatwością obejść ten problem poprzez utworzenie niestandardowego wskaźnika, który obliczałby dokładność po zaokrągleniu zmiennych docelowych i prognoz do wartości 0 lub 1.



Rysunek 17.4. Obrazy oryginalne (na górze) i ich rekonstrukcje (na dole)

Wizualizowanie zestawu danych Fashion MNIST

Po wyuczeniu autokodera stosowego możemy wykorzystać go do redukowania wymiarowości zestawu danych. W przypadku wizualizacji nie uzyskujemy tak dobrych rezultatów w porównaniu do innych algorytmów redukowania wymiarowości (np. opisanych w rozdziale 8.), ale jedną z zalet autokoderów jest możliwość przetwarzania bardzo dużych zestawów danych, zawierających mnóstwo przykładów i cech. Zatem jedną ze strategii jest zredukowanie wymiarowości do rozsądnego poziomu za pomocą autokodera, a następnie przeprowadzenie wizualizacji za pomocą innego algorytmu redukcji wymiarowości. Użyjmy tej strategii do zwizualizowania zestawu danych Fashion MNIST. Najpierw wykorzystamy koder z naszego autokodera stosowego do zmniejszenia wymiarowości do 30 wymiarów, a potem wprowadzimy implementację algorytmu t-SNE (z modułu Scikit-Learn) w celu dalszego ograniczenia wymiarowości do dwóch wymiarów:

```
from sklearn.manifold import TSNE  
  
X_valid_compressed = stacked_encoder.predict(X_valid)  
tsne = TSNE()  
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Możemy teraz utworzyć wykres zestawu danych:

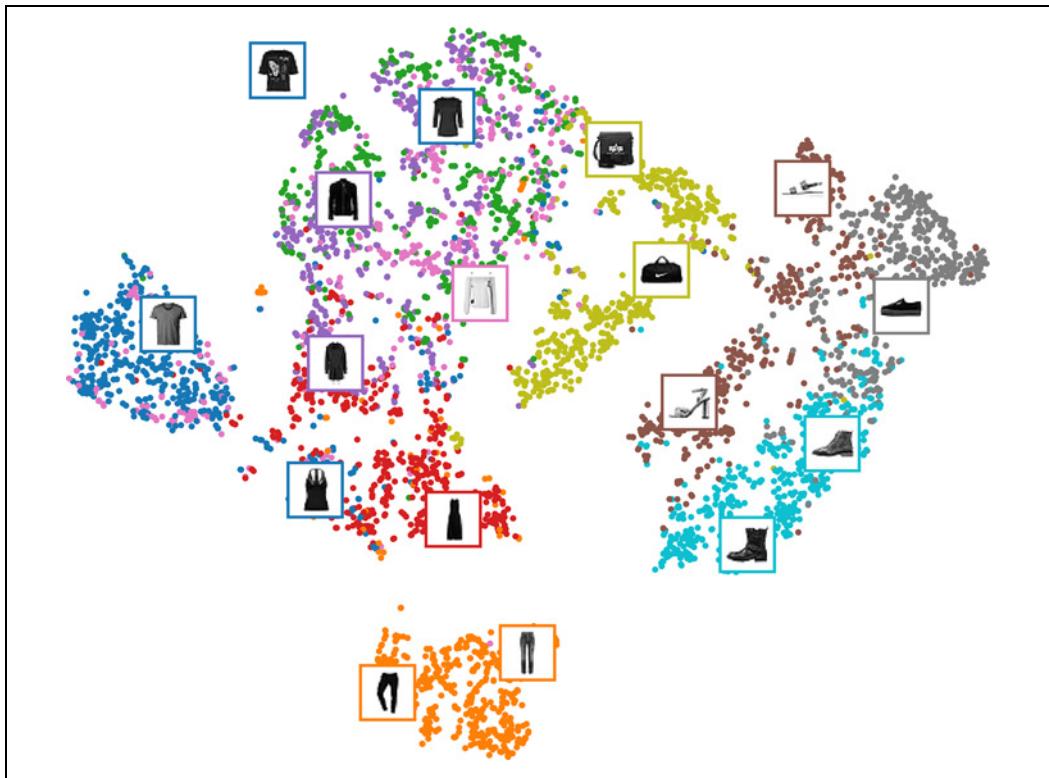
```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

Rysunek 17.5 prezentuje otrzymany wykres punktowy (został nieco upiększony poprzez wyświetlenie niektórych przykładów). Algorytm t-SNE rozpoznał kilka skupień, które dość wiernie odpowiadają poszczególnym klasom (każda klasa ma przypisany swój kolor).

Możemy więc używać autokoderów do redukowania wymiarowości. Innym ich zastosowaniem jest nienadzorowane uczenie wstępne.

Nienadzorowane uczenie wstępne za pomocą autokoderów stosowych

Jak już wiesz z rozdziału 11., jeżeli zajmujemy się złożonym, nadzorowanym zadaniem, ale nie mamy do dyspozycji wielu oznakowanych danych uczących, jednym z rozwiązań jest znalezienie sieci neuronowej pełniącej podobną funkcję i wykorzystanie jej niższych warstw. W ten sposób możemy

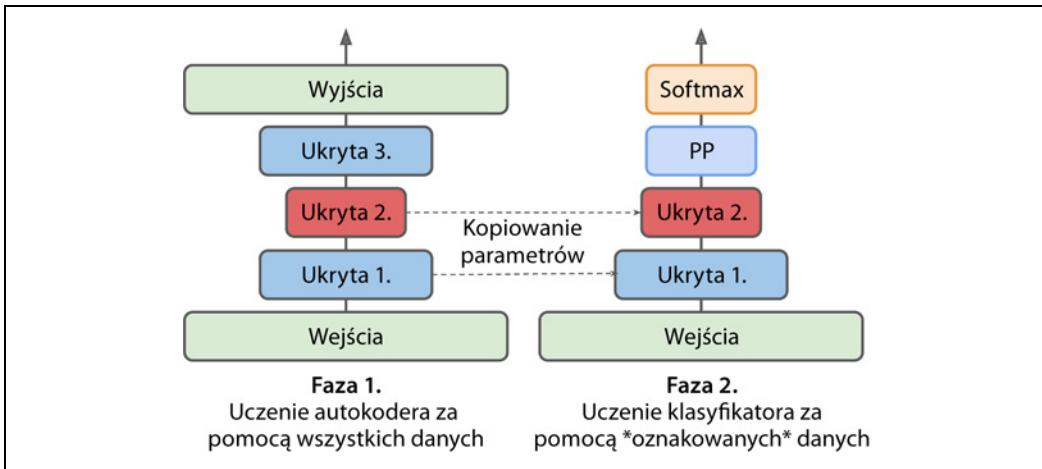


Rysunek 17.5. Wizualizacja zestawu danych Fashion MNIST za pomocą kombinacji autokodera i algorytmu t-SNE wytrenować bardzo wydajny model przy użyciu niewielkiego zbioru danych uczących, gdyż dzięki temu Twój sieć neuronowa nie będzie musiała uczyć się najprostszego cech — będzie po prostu korzystać z gotowych wykrywaczy cech.

Podobnie ma się rzecz w przypadku dużych zbiorów danych, które są w większości nieoznakowane: najpierw możemy uczyć autokoder stosowy za pomocą wszystkich dostępnych danych, następnie wykorzystać niższe warstwy do stworzenia sieci neuronowej przeznaczonej do określonego zadania i wytrenować ją przy użyciu oznakowanych danych. Na rysunku 17.6 widzimy przykład wykorzystania autokodera stosowego w nienadzorowanym uczeniu wstępny klasyfikującej sieci neuronowej. Mając niewielką liczbę oznakowanych danych uczących przeznaczonych do uczenia klasyfikatora, możemy chcieć zamrozić uprzednio wytrenowane warstwy (przynajmniej znajdującej się niżej w sieci).



Zestawy danych zawierające olbrzymią liczbę danych nieoznakowanych i niewiele danych oznakowanych są powszechnie spotykane. Tworzenie rozbudowanych, nieoznakowanych danych uczących nie jest kosztowne (wystarczy np. napisać prosty skrypt pobierający miliony obrazów z internetu), ale tylko człowiek może je rzetelnie etykietować (np. klasyfikując je jako nadające się do uczenia lub nie). Etykietowanie przykładów jest drogim i czasochłonnym zajęciem, zatem często mamy do czynienia z zaledwie kilkoma tysiącami oznakowanych danych.



Rysunek 17.6. Nienadzorowane uczenie wstępne za pomocą autokoderów

Implementacja tego rozwiązania nie powinna nastręczać większych trudności: wystarczy wytrenować autokoder za pomocą wszystkich dostępnych danych uczących (oznakowanych i nieoznakowanych), a następnie wykorzystać ponownie jego warstwy kodera do utworzenia nowej sieci neuronowej (przykład znajdziesz w ćwiczeniach umieszczonej na końcu rozdziału).

Przyjrzyjmy się teraz kilku technikom uczenia autokoderów stosowanych.

Wiązanie wag

W przypadku dość symetrycznych autokoderów (np. takiego jak utworzony przez nas) popularnym rozwiązaniem jest **wiązanie wag** (ang. *tying weights*) warstw dekodera z wagami warstw kodera. Redukujemy w ten sposób o połowę liczb wag w modelu, co powoduje przyspieszenie procesu uczenia i ograniczenie ryzyka przetrenowania. Dokładniej mówiąc, jeśli autokoder zawiera N warstw (nie licząc warstwy wejściowej), a W_L wyznacza wagę połączeń w L -tej warstwie (tzn. warstwa 1. jest pierwszą warstwą ukrytą, warstwa $N/2$ stanowi warstwę kodowania, a N określa warstwę wyjściową), to wagi warstwy dekodera możemy zdefiniować jako: $W_{N-L+1} = W_L^T$ (dla $L = 1, 2, \dots, N/2$).

Aby powiązać wagi pomiędzy warstwami za pomocą interfejsu Keras, zdefiniujemy najpierw niestandardową warstwę:

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="obciazenie", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

Warstwa ta działa jak zwykła warstwa Dense, ale wykorzystuje transponowane wagi innej warstwy Dense (wyznaczenie transpose_b=True jest równoznaczne transponowaniu drugiego argumentu, ale rozwiązywanie to jest skuteczniejsze wtedy, gdy przeprowadza na bieżąco operację transponowania wewnętrz funkcji matmul()). Zawiera ona jednak własny wektor obciążen. Możemy teraz zbudować nowy autokoder stosowy, który będzie przypominał utworzony przez nas wcześniej model, ale tym razem warstwy Dense dekodera będą powiązane z warstwami Dense kodera:

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

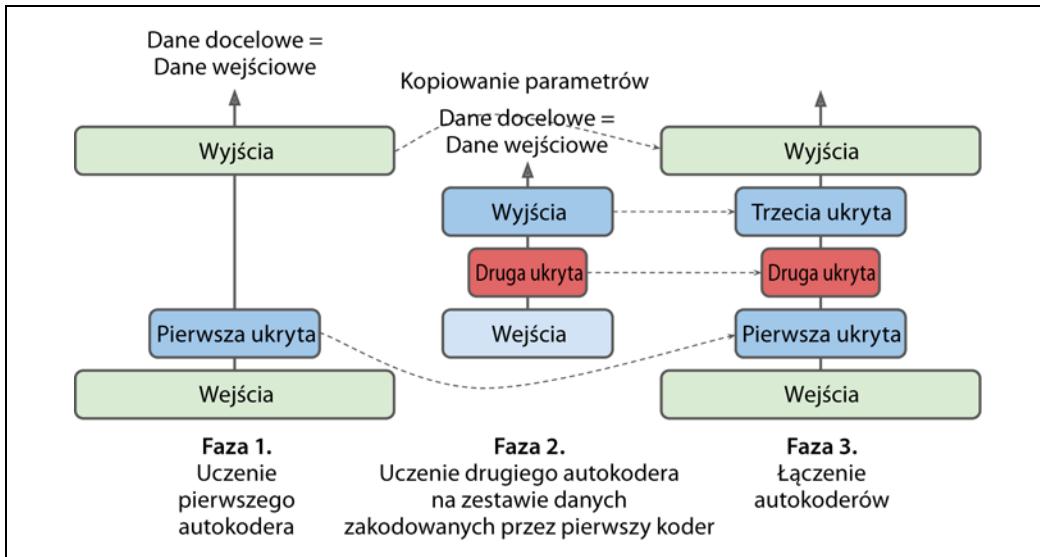
tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

Model osiąga nieco mniejszą wartość błędu rekonstrukcji w porównaniu do poprzedniego modelu, przy zaledwie połowie liczby parametrów.

Uczenie autokoderów pojedynczo

Zamiast uczyć naraz cały autokoder stosowy (tak, jak to przed chwilą zrobiliśmy), często możemy znacznie zaoszczędzić czas poprzez trenowanie pojedynczo poszczególnych autokoderów składowych, a następnie łączenie ich w jeden autokoder stosowy (stąd bierze się jego nazwa), co zostało pokazane na rysunku 17.7. Obecnie rozwiązanie to nie jest tak często stosowane, ale niekiedy można natrafić na artykuły wspominające o „zachłannym uczeniu warstwowym” (ang. *greedy layerwise training*), dobrze więc wiedzieć, co oznacza ten termin.

W pierwszej fazie trenowania pierwszy autokoder uczy się rekonstruować dane wejściowe. Następnie kodujemy cały zestaw danych uczących za pomocą pierwszego autokodera i uzyskujemy w ten sposób nowy (skompresowany) zbiór uczący. Za pomocą tego zestawu trenujemy drugi autokoder. Jest to faza druga uczenia modelu. Na końcu składamy po prostu „kanapkę” ze wszystkich dostępnych autokoderów, tak jak pokazano na rysunku 17.7 (tzn. najpierw łączymy warstwy ukryte każdego kodera, a potem warstwy wyjściowe w odwrotnej kolejności). Tak uzyskujemy ostateczny autokoder stosowy (jego implementację znajdziesz w sekcji „Uczenie autokoderów pojedynczo” w notatniku Jupyter). Możemy w ten sposób trenować wiele pojedynczych autokoderów i konstruować bardzo głębokie modele.



Rysunek 17.7. Uczenie autokoderów pojedynczo

Jak już wiesz, jedną z przyczyn ponownego zainteresowania dziedziną uczenia głębokiego było dokonane w 2006 roku przez Geoffreya Hintona i in. odkrycie (<https://www.cs.toronto.edu/~hinton/absps/ncfast.pdf>), że można uczyć wstępnie sieci neuronowe w sposób nienadzorowany, za pomocą zachłannej techniki warstwowej. Wykorzystano w tym celu **ograniczone maszyny Boltzmanna** (ang. *Restricted Boltzmann Machines* — RBM) (zob. dodatek E), ale w 2006 roku Joshua Bengio i in. udowodnili (<https://papers.nips.cc/paper/3048-greedy-layer-wise-training-of-deep-networks.pdf>)³, że również dobrze nadają się do tego zadania autokodery. Przez kilka lat był to jedyny sposób skutecznego uczenia sieci głębokich, aż do momentu pojawienia się omówionych w rozdziale 11. technik umożliwiających trenowanie sieci głębokiej za jednym podejściem.

Autokodery nie ograniczają się do sieci gęstych: możemy tworzyć autokodery splotowe, a nawet rekuencyjne. Zobaczmy, jak to robić.

Autokodery splotowe

Jeżeli przetwarzasz obrazy, to dotychczas omawiane autokodery okazują się niezbyt przydatne (chyba że obrazy są bardzo małe). Jak wiesz z rozdziału 14., splotowe sieci neuronowe nadają się do pracy z obrazami znacznie bardziej niż sieci gęste. Jeżeli więc chcesz zbudować autokoder przeznaczony do obrazów (np. nienadzorowanego uczenia wstępnego lub redukowania wymiarowości), to musisz utworzyć **autokoder splotowy** (ang. *convolutional autoencoder*; <http://people.idsia.ch/~ciresan/data/icann2011.pdf>)⁴. Koderem jest tradycyjna sieć splotowa składająca się z warstw splotowych

³ Yoshua Bengio i in., *Greedy Layer-Wise Training of Deep Networks*, „Proceedings of the 19th International Conference on Neural Information Processing Systems” (2006), s. 153 – 160.

⁴ Jonathan Masci i in., *Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction*, „Proceedings of the 21st International Conference on Artificial Neural Networks” 1 (2011), s. 52 – 59.

i łączących. Zazwyczaj zmniejsza ona wymiarowość danych wejściowych (wysokość i szerokość obrazu), a jednocześnie zwiększa ich głębokość (liczbę map cech). Dekoder musi przeprowadzać odwrotną operację (zwiększyć rozdzielcość obrazu i zredukować głębokość do pierwotnej liczby wymiarów), dlatego możemy w tym celu wykorzystać transponowane warstwy splotowe (ewentualnie możesz łączyć warstwy ekspansji z warstwami splotowymi). Oto prosta implementacja autokodera splotowego dostosowanego do zestawu danych Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                               activation="selu",
                               input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                               activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                               activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Autokodery rekurencyjne

Jeżeli chcesz tworzyć autokodery przeznaczone do przetwarzania sekwencji, takich jak szeregi czasowe czy teksty (np. nienadzorowanego uczenia wstępnego lub redukowania wymiarowości), to rekurencyjne sieci neuronowe (zob. rozdział 15.) mogą nadawać się do tego zadania lepiej niż sieci gęste. Budowanie **autokodera rekurencyjnego** (ang. *recurrent autoencoder*) jest proste: koder stanowi zazwyczaj sieć sekwencyjno-wektorową, która kompresuje sekwencję wejściową do postaci pojedynczego wektora. Dekoder to sieć wektorowo-sekwencyjna przeprowadzająca odwrotną operację:

```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

Taki autokoder rekurencyjny może przetwarzać sekwencje o dowolnej długości, które w każdym takcie zawierają 28 wymiarów. Jest to dla nas dogodne, ponieważ oznacza, że możemy traktować obrazy z zestawu Fashion MNIST tak, jakby każdy obraz stanowił sekwencję rzędów: w każdym takcie sieć rekurencyjna będzie przetwarzać pojedynczy, 28-elementowy rząd pikseli. Oczywiście autokoder rekurencyjny może być stosowany wobec dowolnego rodzaju sekwencji. Zwróć uwagę,

że jeśli jako pierwszą warstwę dekodera wstawimy RepeatVector, to musimy sprawić, żeby w każdym takcie do dekodera był dostarczany wektor wejściowy.

Zatrzymajmy się tu na chwilę. Poznaliśmy jak na razie wiele typów autokoderów (podstawowe, stosowe, splotowe i rekurencyjne) i nauczyliśmy się je trenować (albo za jednym zamachem, albo warstwa po warstwie). Znamy także już ich dwa zastosowania: wizualizowanie danych i nienadzorowane uczenie wstępne.

Do tej pory zmuszaliśmy autokoder do rozpoznawania interesujących cech, ograniczając rozmiar warstwy kodowania, przez co był on niedopełniony. W rzeczywistości istnieje wiele rodzajów ograniczeń, które możemy wykorzystać, w tym również umożliwiające stosowanie warstwy kodowania o takim samym rozmiarze jak wejściowa, a nawet jeszcze większej, co pozwala uzyskać **autokoder przepelniony** (ang. *overcomplete autoencoder*). Sprawdźmy teraz te inne rozwiązania.

Autokodery odszumiające

Kolejną metodą zmuszania autokodera do poznawania przydatnych cech jest dodawanie szumu do danych wejściowych i uczenie go odzyskiwania pierwotnych, niezaszumionych informacji. Pierwsze koncepcje wykorzystania autokoderów do odszumiania danych pojawiły się już w latach 80. ubiegłego wieku (m.in. pomysł ten został wspomniany w pracy magisterskiej Yanna LeCuna w 1987 roku). Pascal Vincent i in. udowodnili w publikacji z 2008 roku (https://www.iro.umontreal.ca/~vincentp/Publications/denoising_autoencoders_tr1316.pdf)⁵, że autokodery mogą być również używane do wydobywania cech. Z kolei ten sam autor i in. w artykule z 2010 roku (<http://jmlr.csail.mit.edu/papers/v11/vincent10a>)⁶ zaprezentowali **odszumiające autokodery stosowe** (ang. *stacked denoising autoencoders*).

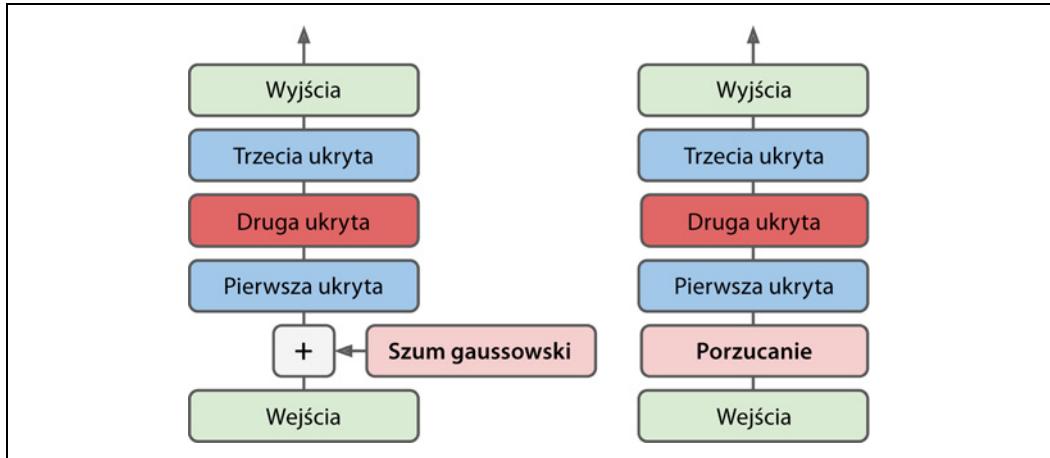
Zaszumienie może być standardowym szumem gaussowskim dodawanym do danych wejściowych lub może przybrać postać losowo wyłączanych wejść za pomocą metody porzucania (zob. rozdział 11.). Obydwa rozwiązania zostały pokazane na rysunku 17.8.

Implementacja nie stanowi wyzwania: jest to standardowy autokoder stosowy zawierający dodatkową warstwę Dropout, przez którą przechodzą dane wejściowe (możesz zastąpić ją warstwą GaussianNoise). Jak pamiętamy, warstwa Dropout jest aktywna jedynie w fazie uczenia (podobnie jak warstwa GaussianNoise):

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
```

⁵ Pascal Vincent i in., *Extracting and Composing Robust Features with Denoising Autoencoders*, „Proceedings of the 25th International Conference on Machine Learning” (2008), s. 1096 – 1103.

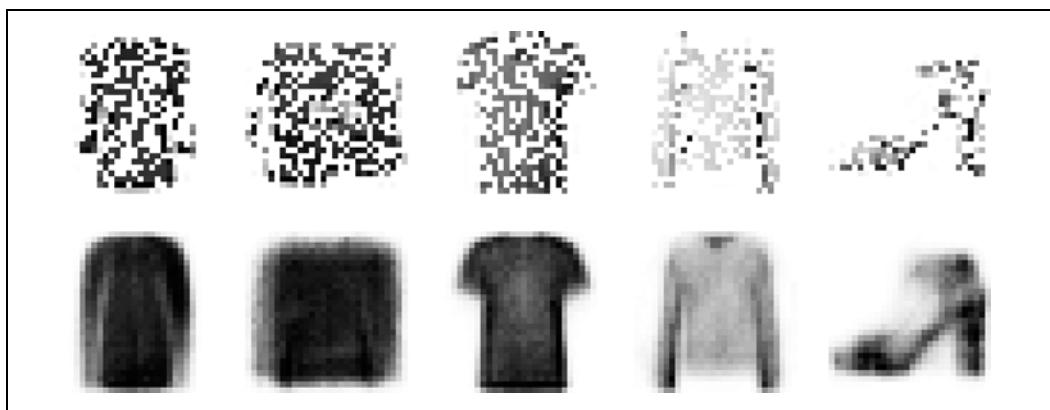
⁶ Pascal Vincent i in., *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, „Journal of Machine Learning Research” 11 (2010), s. 3371 – 3408.



Rysunek 17.8. Autokodery odszumiające: wykorzystujące szum gaussowski (po lewej) lub metodę porzucania (po prawej)

```
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

Rysunek 17.9 przedstawia kilka zaszumionych obrazów (połowa pikseli została „wyłączona”), a także ich rekonstrukcje uzyskane za pomocą autokodera odszumiającego (bazującego na warstwie porzucania). Zwróć uwagę, że autokoder w istocie „zgaduje” szczegóły niewystępujące w obrazach wejściowych, na przykład górną część białej sukienki (czwarty obraz w dolnym rzędzie). Jak widać, autokodery odszumiające służą nie tylko do wizualizowania danych lub nienadzorowanego uczenia wstępnego, lecz również w dość prosty i skuteczny sposób mogą usuwać szum z obrazów.



Rysunek 17.9. Zaszumione obrazy (na górze) i ich rekonstrukcje (na dole)

Autokodery rzadkie

Innym ograniczeniem prowadzącym do skutecznego wydobywania cech jest **rzadkość** (ang. *sparsity*): przez dodanie odpowiedniego członu do funkcji kosztu autokoder zostaje zmuszony do zmniejszenia liczby aktywnych neuronów w warstwie kodowania. Możemy sprawić na przykład, żeby w warstwie kodującej występowało tylko 5% znacząco aktywnych neuronów. W ten sposób wymuszamy na autokoderze reprezentowanie każdego wejścia jako kombinacji niewielkiej liczby pobudzeń. Dzięki temu każdy neuron warstwy kodowania zazwyczaj uczy się wykrywać jakąś przydatną cechę (gdybyśmy wypowiadali w ciągu miesiąca tylko kilka słów, prawdopodobnie chcielibyśmy przekazywać za ich pomocą wartościowe informacje).

Prostym rozwiążaniem okazuje się wprowadzenie sigmoidalnej funkcji aktywacji w warstwie kodowania (co ogranicza wartości kodowań do zakresu od 0 do 1), wprowadzenie dużej warstwy kodowania (np. składającej się z 300 jednostek) i dodanie regularyzacji ℓ_1 do pobudzeń warstwy kodowania (w dekoderze nie wprowadzamy żadnych zmian):

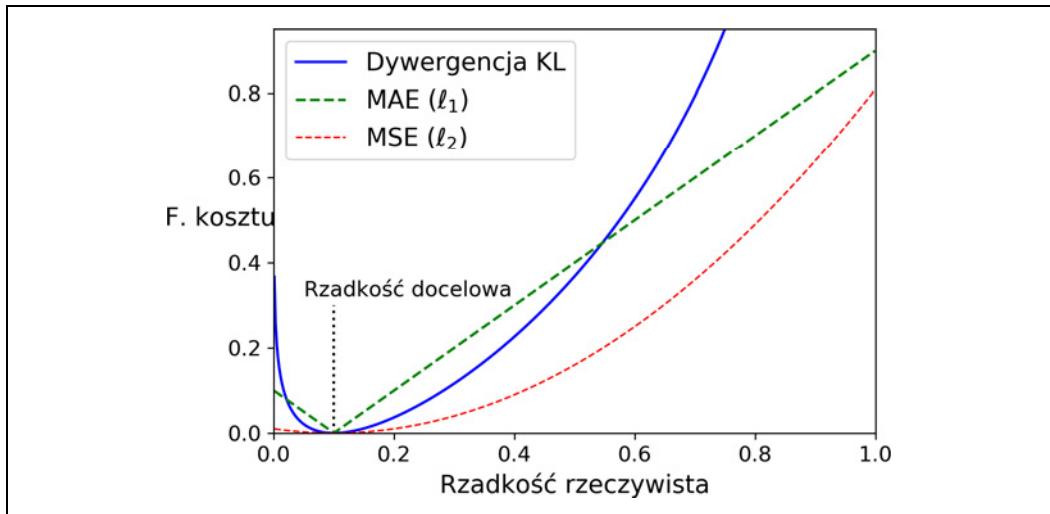
```
sparse_11_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_11_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_11_ae = keras.models.Sequential([sparse_11_encoder, sparse_11_decoder])
```

Taka warstwa ActivityRegularization zwraca jedynie swoje dane wejściowe, ale jako skutek uboczny dodaje funkcję straty uczenia równą sumie wartości bezwzględnych tychże sygnałów wejściowych (warstwa ta jest aktywna wyłącznie w fazie uczenia). Ewentualnie możesz usunąć warstwę ActivityRegularization i wyznaczyć atrybut activity_regularizer=keras.regularizers.l1(1e-3) w warstwie poprzedzającej. Kara ta będzie zmuszała sieć neuronową do tworzenia kodowań o wartościach bliskich 0, ale jednocześnie model będzie również karany za niewłaściwe rekonstruowanie danych wejściowych, więc będzie musiał wyznaczać co najmniej kilka wartości niezerowych. Norma ℓ_1 sprawia, że sieć będzie przechowywała najważniejsze kodowania i eliminowała nieprzydatne z perspektywy obrazu wejściowego (w przeciwieństwie do normy ℓ_2 , która jedynie redukowałaby wszystkie kodowania).

Innym rozwiążaniem, często dającym lepsze rezultaty, jest pomiar rzeczywistej rzadkości warstwy kodowania w każdym przebiegu uczenia i karanie modelu w sytuacji, gdy zmierzona rzadkość różni się od rzadkości docelowej. Robimy to, obliczając średnią aktywację każdego neuronu w tej warstwie dla całej grupy próbek uczących. Rozmiar tej grupy nie może być zbyt mały, gdyż wyliczona wartość średniej nie będzie wtedy dokładna.

Po uzyskaniu średniej wartości aktywacji każdego neuronu chcemy nałożyć karę na zbyt aktywne neurony poprzez dodanie **funkcji straty rzadkości** (ang. *sparsity loss*) do funkcji kosztu. Przykładowo jeśli zmierzymy średnią wartość aktywacji neuronu rzędu 0,3, ale aktywność docelowa powinna wynosić

sić 0,1, musimy zmniejszyć jego aktywność. Jednym ze sposobów jest dodanie kwadratu błędu $(0,3-0,1)^2$ do funkcji kosztu, w praktyce jednak lepszym rozwiązaniem okazuje się wprowadzenie dywergencji Kullbacka-Leiblera (zob. rozdział 4.), której gradienty są znacznie większe niż w błędzie średniokwadratowym (rysunek 17.10).



Rysunek 17.10. Funkcja straty rzadkości

Mając dwa dyskretne rozkłady prawdopodobieństwa P i Q , możemy obliczyć rozbieżność pomiędzy nimi ($D_{KL}(P\|Q)$) za pomocą równania 17.1.

Równanie 17.1. Dywergencja Kullbacka-Leiblera

$$D_{KL}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

W naszym przypadku chcemy zmierzyć rozbieżność pomiędzy docelowym prawdopodobieństwem p aktywacji neuronu w warstwie kodowania a rzeczywistym prawdopodobieństwem q (tzn. średnią aktywacją dla grupy przykładów uczących). Zatem dywergencja KL ulega uproszczeniu do postaci widocznej w równaniu 17.2.

Równanie 17.2. Dywergencja KL pomiędzy docelową rzadkością p a rzeczywistą rzadkością q

$$D_{KL}(p\|q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}$$

Po obliczeniu funkcji straty rzadkości dla każdego neuronu w warstwie kodowania wystarczy je zsumować i dodać wynik do funkcji kosztu. W celu regulowania względnej istotności funkcji straty rzadkości i funkcji straty rekonstrukcji możemy pomnożyć tę pierwszą przez hiperparametr wagi rzadkości. Jeżeli wartość wagi będzie za duża, model pozostanie blisko rzadkości docelowej, ale jednocześnie może nie być w stanie prawidłowo rekonstruować danych wejściowych, przez co okaże się

bezużyteczny. Z kolei przy zbyt małej wartości wagi rzadkości model będzie ignorował cel rzadkości i nie nauczy się rozpoznawać przydatnych cech.

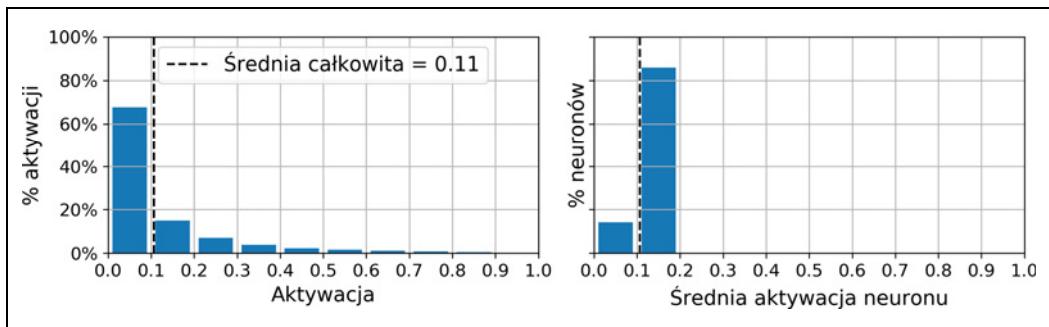
Dysponujemy już wszystkimi składnikami niezbędnymi do zaimplementowania autokodera rzadkiego opartego na dywergencji KL. Utwórzmy najpierw niestandardowy regularyzator wprowadzający dywergencję KL:

```
K = keras.backend  
kl_divergence = keras.losses.kullback_leibler_divergence  
  
class KLDivergenceRegularizer(keras.regularizers.Regularizer):  
    def __init__(self, weight, target=0.1):  
        self.weight = weight  
        self.target = target  
    def __call__(self, inputs):  
        mean_activities = K.mean(inputs, axis=0)  
        return self.weight * (  
            kl_divergence(self.target, mean_activities) +  
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Możemy teraz zbudować autokoder rzadki i wykorzystać KLDivergenceRegularizer do aktywacji warstwy kodowania:

```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)  
sparse_kl_encoder = keras.models.Sequential([  
    keras.layers.Flatten(input_shape=[28, 28]),  
    keras.layers.Dense(100, activation="selu"),  
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)  
])  
sparse_kl_decoder = keras.models.Sequential([  
    keras.layers.Dense(100, activation="selu", input_shape=[300]),  
    keras.layers.Dense(28 * 28, activation="sigmoid"),  
    keras.layers.Reshape([28, 28])  
])  
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

Po wyuczeniu tego autokodera rzadkiego na zestawie danych Fashion MNIST wartości pobudzeń neuronów w warstwie kodowań są przeważnie bliskie 0 (ok. 70% wszystkich aktywacji nie przekracza wartości 0,1), a średnia wartość pobudzenia wynosi ok. 0,1 (średnie pobudzenie w 90% neuronów mieści się w zakresie od 0,1 do 0,2), co widać na rysunku 17.11.



Rysunek 17.11. Rozkład wszystkich pobudzeń w warstwie kodowania (po lewej) i rozkład średniego pobudzenia każdego neuronu (po prawej)

Autokodery wariacyjne

Kolejną istotną kategorię autokoderów zaproponowali w 2013 roku (<https://arxiv.org/abs/1312.6114>) Diederik Kingma i Max Welling; bardzo szybko zyskała ona znaczną popularność. Mowa o **autokodermach wariacyjnych** (ang. *variational autoencoders*)⁷.

Różnią się one istotnie od dotychczas omówionych rodzajów autokoderów. Oto główne różnice:

- Są **autokoderami probabilistycznymi** (ang. *probabilistic autoencoders*), co oznacza, że generują częściowo losowe wyniki, nawet po wyuczeniu modelu (w przeciwieństwie do autokoderów odszumiających, w których losowość jest dodawana wyłącznie na wejściu).
- Co ważniejsze, stanowią klasę **autokoderów generatywnych**, czyli są w stanie tworzyć nowe próbki przypominające dane zawarte w zbiorze uczącym.

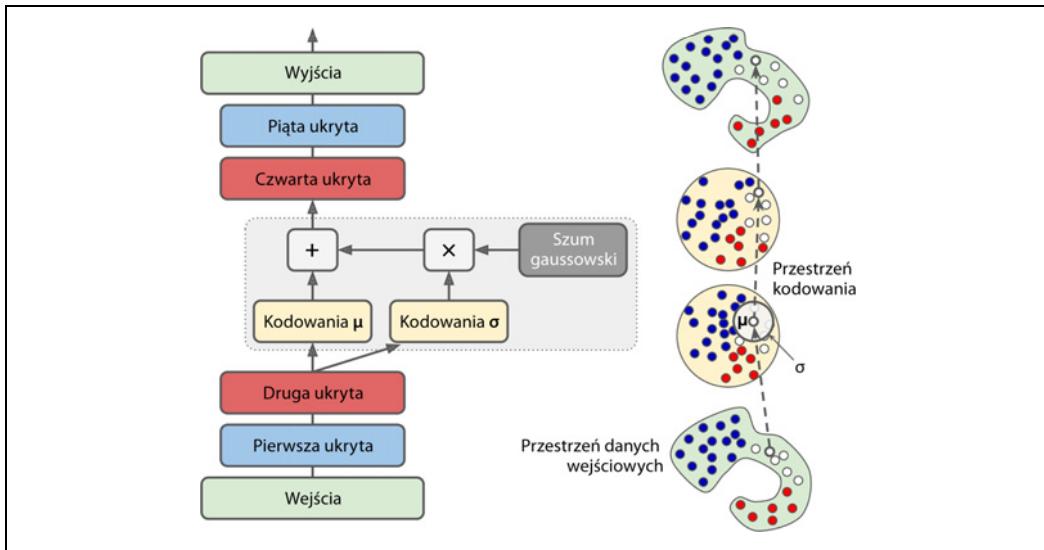
Cechy te upodabniają te autokodery do maszyn RBM, ale są łatwiejsze do trenowania, a proces próbkiowania przebiega znacznie szybciej (w przypadku ograniczonych maszyn Boltzmanna musimy poczekać, aż sieć osiągnie stan „równowagi termicznej” przed wprowadzeniem nowego przykładu). Istotnie, jak sama nazwa wskazuje, autokodery wariacyjne przeprowadzają wariacyjne wnioskowania bayesowskie (zob. rozdział 9.), stanowiące skuteczny mechanizm aproksymowanego wnioskowania bayesowskiego.

Przyjrzymy się mechanizmowi ich działania. Rysunek 17.12 (po lewej) pokazuje schemat autokoderu wariacyjnego. Możemy tu oczywiście rozpoznać podstawową architekturę autokoderów — po koderze występuje dekoder (w omawianym przykładzie obydwa elementy zawierają po dwie warstwy ukryte), ale mamy tutaj do czynienia z pewną modyfikacją: koder generuje nie bezpośrednie kodowanie danej próbki wejściowej, lecz **uśrednione kodowanie μ** (ang. *mean coding*) i **odchylenie standarde σ** . Rzeczywiste kodowanie jest następnie próbkiowane losowo z rozkładu gaussowskiego przy użyciu tej średniej μ i odchylenia standardowego σ . Dekoder może teraz w standardowy sposób dekodować próbkiowe kodowanie. Po prawej stronie widzimy przykład uczący przechodzący przez taki autokoder. Najpierw koder generuje wartości μ i σ , następnie kodowanie podlega losowemu próbkiowaniu (zwróć uwagę, że nie znajduje się ono dokładnie w punkcie μ), a na końcu zostaje zdekodowane; końcowy wynik przypomina próbkę wejściową.

Jak widać na załączonym schemacie, mimo skomplikowanego rozkładu danych wejściowych autokoder wariacyjny jest w stanie generować kodowania przypominające próbkiowania za pomocą prostego szumu gaussowskiego⁸: w czasie uczenia funkcja kosztu (wkrótce ją omówimy) zmusza kodowania do stopniowego poruszania się po przestrzeni kodowania (zwanej również **przestrzenią ukrytą** — ang. *latent space*) w poszukiwaniu miejsca wewnątrz obszaru przypominającego chmurę punktów gaussowskich. Bardzo ważną konsekwencją takiego zachowania jest fakt, że po wyuczeniu autokodera wariacyjnego możemy bardzo łatwo wygenerować nową próbkię — wystarczy próbkiować losowe kodowanie z rozkładu gaussowskiego, rozkodować je i to wszystko!

⁷ Diederik Kingma i Max Welling, *Auto-Encoding Variational Bayes*, arXiv preprint arXiv:1312.6114 (2013).

⁸ Autokodery wariacyjne są w rzeczywistości bardziej ogólne: kodowania nie ograniczają się wyłącznie do rozkładu gaussowskiego.



Rysunek 17.12. Autokoder wariacyjny (po lewej) i przechodzący przez niego przykład uczący (po prawej)

Wróćmy teraz do funkcji kosztu. Składa się ona z dwóch części. Pierwszym jej członem jest tradycyjna funkcja straty rekonstrukcji zmuszająca autokoder do rekonstruowania danych wejściowych (jak już wiesz, możemy w tym celu wykorzystać entropię krzyżową). Drugi element nosi nazwę **funkcji straty ukrytej** (ang. *latent loss*) i sprawia, że autokoder uzyskuje kodowania przypominające uzyskane z prostego rozkładu gaussowskiego — stosujemy w tym przypadku dywergencję KL pomiędzy docelowym rozkładem (gaussowskim) a rzeczywistym rozkładem kodowań. Obliczenia są tu nieco bardziej skomplikowane niż w poprzednim przypadku, zwłaszcza z powodu szumu gaussowskiego, który ogranicza ilość informacji dostarczanych do warstwy kodowania (i zmusza autokoder do uczenia się przydatnych cech). Na szczeble wymagane wzory przyjmują uproszoną postać i możemy je obliczyć za pomocą równania 17.3⁹.

Równanie 17.3. Funkcja straty ukrytej w autokoderze wariacyjnym

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2$$

W tym równaniu \mathcal{L} jest funkcją straty ukrytej, n wyznacza wymiarowość kodowań, a μ_i i σ_i to, odpowiednio, średnia i odchylenie standardowe i -tej składowej kodowania. Koder generuje na wyjściu wektory μ i σ (przechowujące wszystkie wartości μ_i oraz σ_i), co zaprezentowano na rysunku 17.12 (po lewej).

Często spotykaną modyfikacją struktury autokodera wariacyjnego jest zastąpienie σ na wyjściu kodera operacją $\gamma = \log(\sigma^2)$. Można wtedy obliczyć funkcję straty ukrytej zgodnie z równaniem 17.4. Rozwiążanie to jest stabilniejsze numerycznie i przyspiesza proces uczenia.

⁹ Osoby zainteresowane szczegółami mogą zapoznać się z oryginalną publikacją omawiającą autokodery wariacyjne lub ze znakomitym wprowadzeniem autorstwa Carla Doerscha (<https://arxiv.org/abs/1606.05908>, 2016).

Równanie 17.4. Funkcja straty ukrytej w autokoderze wariacyjnym zmodyfikowana za pomocą wyrażenia $\gamma = \log(\sigma^2)$

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

Zacznijmy od zbudowania autokodera wariacyjnego dla zestawu danych Fashion MNIST (takiego jak na rysunku 17.12, ale z dodaną modyfikacją γ). Najpierw musimy przygotować niestandardową warstwę służącą do próbkowania kodowań przy danych μ i γ :

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

Ta warstwa Sampling przyjmuje na wejściu średnią (mean ; μ) i logarytm wariancji (log_var ; γ). Wykorzystujemy tu funkcję `K.random_normal()` do próbkowania losowego wektora (o takich samych wymiarach jak γ) z rozkładu normalnego o średniej 0 i odchyleniu standardowym 1. Wektor ten zostaje następnie pomnożony przez $\exp(\gamma/2)$ (możesz sprawdzić, że wartość ta jest równa σ), a na koniec dodajemy μ i zostaje zwrocony rezultat. Uzyskujemy w ten sposób wektor kodowań o rozkładzie normalnym, średniej μ i odchyleniu standardowym σ .

Następnie tworzymy koder za pomocą interfejsu funkcyjnego, ponieważ model nie jest w całości sekwencyjny:

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z) # μ
codings_log_var = keras.layers.Dense(codings_size)(z) # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Zwróć uwagę, że warstwy Dense generujące na wyjściu `codings_mean` (μ) i `codings_log_var` (γ) mają takie same wejścia (czyli sygnały wyjściowe z drugiej warstwy Dense). Przekazujemy następnie `codings_mean` i `codings_log_var` do warstwy Sampling. Model `variational_encoder` zawiera trzy wyjścia na wypadek, gdybyśmy chcieli sprawdzać wartości `codings_mean` i `codings_log_var`. Będziemy wykorzystywać tylko ostatnie wyjście (`codings`). Przygotujmy teraz dekoder:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

Ten dekoder można bez problemu zbudować za pomocą interfejsu sekwencyjnego, ponieważ tworzy go jedynie szereg warstw i nie różni się od innych konstruowanych przez nas dekoderów. Czas w końcu utworzyć model autokodera wariacyjnego:

```
_ , _ , codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])
```

Zwróć uwagę, że ignorujemy dwa pierwsze wyjścia kodera (chcemy tylko przekazywać kodowania). Pozostało nam jeszcze wprowadzenie funkcji straty ukrytej i funkcji straty rekonstrukcji:

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Najpierw obliczamy funkcję straty ukrytej dla każdego przykładu w grupie za pomocą równania 17.4 (sumujemy wzduż osi ostatniego wymiaru). Następnie obliczamy średnią funkcję straty ze wszystkich przykładów w grupie i dzielimy wynik przez 784, dzięki czemu mamy pewność, że rezultat będzie w takiej samej skali jak funkcja straty rekonstrukcji. Rzeczywiście, funkcja straty rekonstrukcji w autokoderze wariancyjnym powinna być sumą błędów rekonstrukcji pikseli, jednak interfejs Keras podczas obliczania funkcji straty `binary_crossentropy` wyznacza średnią z 784 pikseli, a nie sumę. Zatem funkcja straty rekonstrukcji jest 784 razy mniejsza, niż powinna być. Moglibyśmy zdefiniować niestandardową funkcję straty obliczającą sumę, a nie średnią, ale łatwiej po prostu podzielić funkcję straty ukrytej przez 784 (końcowa funkcja straty będzie 784 razy mniejsza, niż powinna być, ale oznacza to jedynie, że należy użyć większego współczynnika uczenia).

Zauważ, że korzystamy z optymalizatory RMSProp, który sprawdza się tu całkiem dobrze. W końcu możemy wytrenować autokoder!

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

Generowanie obrazów Fashion MNIST

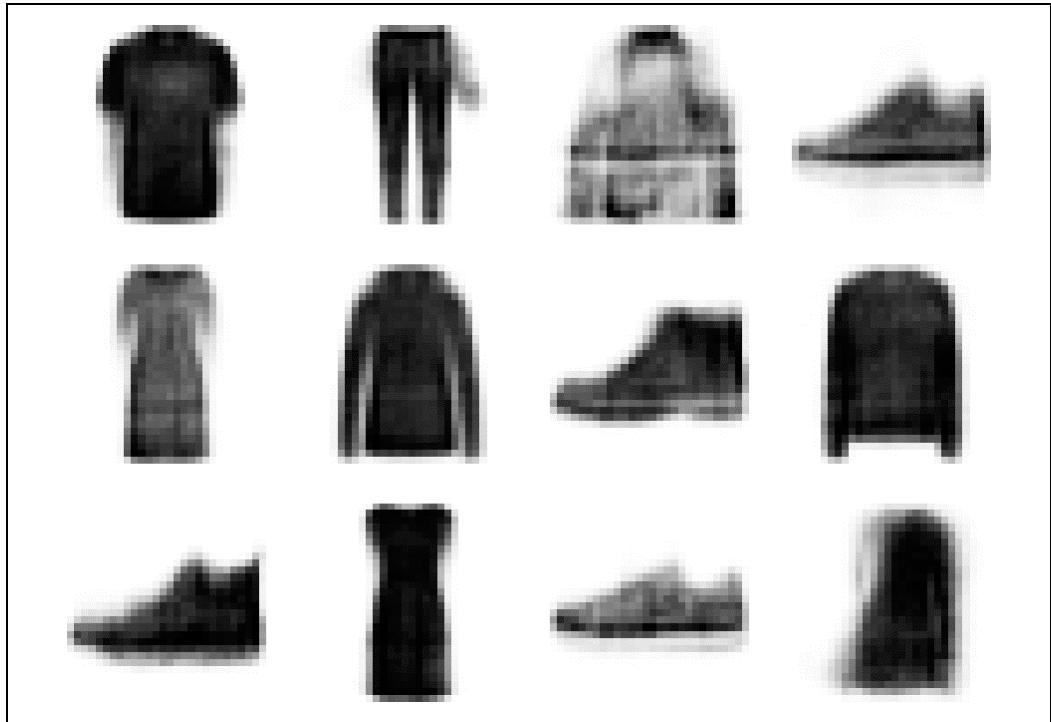
Użyjmy teraz tego autokodera wariancyjnego do wygenerowania obrazów przypominających elementy zestawu danych Fashion MNIST. Wystarczy próbować losowe kodowania i rozkładu gaussowskiego i zdekodować je:

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

Na rysunku 17.13 widzimy 12 wygenerowanych obrazów.

Większość tych obrazów wygląda przekonująco, chociaż są nieco zbyt mocno rozmazane. Inne nie są takie ładne, ale nie bądźmy zbyt surowi dla autokodera, wszak miał tylko kilka minut na naukę! Poświęćmy trochę czasu na dokładniejsze dostrojenie i uczenie, a obrazy powinny wyglądać lepiej.

Autokodery wariancyjne umożliwiają przeprowadzanie **interpolacji semantycznej** (ang. *semantic interpolation*): zamiast przeprowadzać interpolację dwóch obrazów na poziomie pikseli (przez co często wydaje się, że obydwa obrazy są na siebie nałożone) możemy to robić na poziomie kodowań. Przepuszczamy najpierw obydwa obrazy przez koder, następnie interpolujemy uzyskane kodowania, po czym dekodujemy interpolowane kodowanie w obraz wynikowy. Otrzymamy obraz przypominający



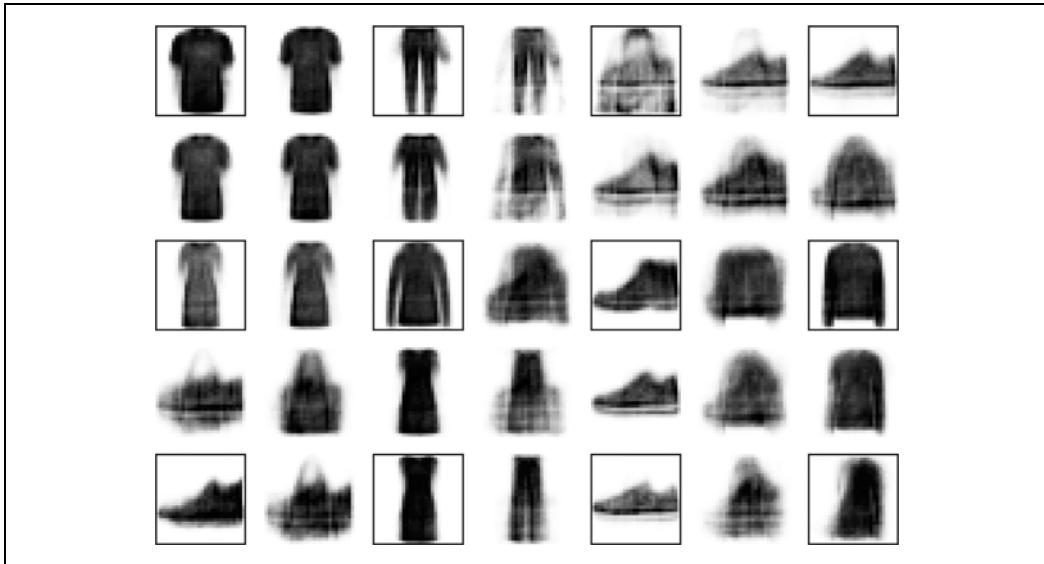
Rysunek 17.13. Obrazy zestawu danych Fashion MNIST wygenerowane za pomocą autokodera wariacyjnego

przykład z zestawu danych Fashion MNIST, ale będzie on stanowił coś pośredniego pomiędzy obydwoma obrazami wejściowymi. W poniższym przykładzie wykorzystamy 12 utworzonych przed chwilą kodowań, ułożymy je w siatkę o rozmiarze 3×4 i zastosujemy funkcję `tf.image.resize()` tak, aby powiększyć tę siatkę do rozmiarów 5×7 . Domyslnie funkcja `resize()` wykonuje interpolację dwuliniową, więc co drugi wiersz i kolumna będą zawierały interpolowane kodowania. Następnie użyjemy dekodera do wygenerowania wszystkich obrazów:

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

Rysunek 17.14 zawiera wygenerowane obrazy. Pierwotne obrazy zostały otoczone ramkami, natomiast pozostałe stanowią rezultaty interpolacji przeprowadzonej pomiędzy sąsiadującymi przykładami. Zwróć na przykład uwagę, że but widoczny w czwartym rzędzie i piątej kolumnie stanowi ładną interpolację dwóch butów znajdujących się powyżej i poniżej.

Autokodery wariacyjne były popularne przez kilka lat, ale w końcu wyprzedziły je sieci GAN, głównie dzięki temu, że te drugie generują znacznie bardziej realistyczne i wyraźniejsze obrazy. Przejdźmy więc do generatywnych sieci przeciwnych.



Rysunek 17.14. Interpolacja semantyczna

Generatywne sieci przeciwnostawne

Generatywne sieci przeciwnostawne (ang. *Generative Adversarial Networks* — GAN) zaproponowali w artykule z 2014 roku (<https://arxiv.org/abs/1406.2661>)¹⁰ Ian Goodfellow i in. Mimo że badacze niemal natychmiast dostrzegli potencjał tej struktury, rozwiązywanie pewnych problemów z jego uczeniem zajęło kilka lat. Z perspektywy czasu pomysł ten, podobnie jak wszystkie błyskotliwe koncepcje, wydaje się bardzo prosty: sprawmy, żeby sieci neuronowe współzawodniczyły ze sobą, w nadziei, że w ten sposób będą wzajemnie się udoskonalać. Jak widać na rysunku 17.15, struktura GAN zawiera dwie sieci neuronowe:

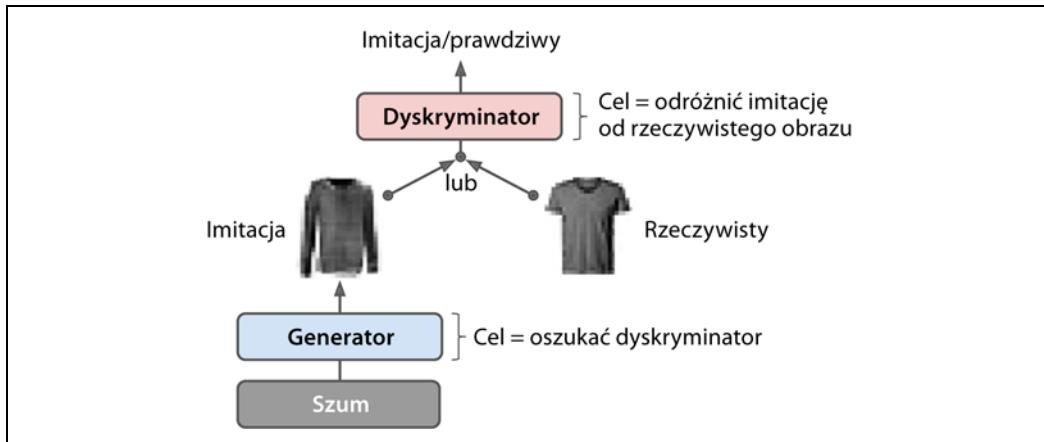
Generator

Na wejściu przyjmuje rozkład losowy (najczęściej gaussowski) i generuje z niego jakieś dane, przeważnie obraz. Możemy traktować losowe dane wejściowe jako reprezentacje ukryte (np. kodowania) obrazu, który ma zostać wygenerowany. Jak więc widać, generator pełni tę samą funkcję co dekoder w autokoderze wariancyjnym i można go w taki sam sposób wykorzystać do generowania nowych obrazów (wystarczy dostarczyć mu szum gaussowski, a otrzymamy z niego zupełnie nowy obraz). Składowa ta jednak jest uczona zupełnie inaczej, o czym przekonasz się już wkrótce.

Dyskryminator

Na wejściu przyjmuje albo imitację obrazu utworzoną przez generator, albo rzeczywisty obraz z zestawu danych i musi „zgadnąć”, czy obraz ten jest rzeczywisty czy fałszywy.

¹⁰ Ian Goodfellow i in., *Generative Adversarial Nets*, „Proceedings of the 27th International Conference on Neural Information Processing Systems” 2 (2014), s. 2672 – 2680.



Rysunek 17.15. Generatywna sieć przeciwnostawna

W trakcie uczenia generator i dyskryminator mają wyznaczone wykluczające się wzajemnie cele: dyskryminator stara się odróżniać imitacje od rzeczywistych obrazów, natomiast generator próbuje tworzyć na tyle przekonujące imitacje, żeby oszukać generator. Sieć GAN tworzą dwie struktury o przeciwnostawnych zadaniach, więc nie można jej trenować w tradycyjny sposób. Każdy przebieg uczenia dzieli się na dwie fazy:

- W pierwszej fazie trenowany jest dyskryminator. Z zestawu danych uczących losowana jest grupa rzeczywistych obrazów i uzupełniana taką samą liczbą imitacji utworzonych przez generator. Imitacje zostają oznakowane etykietami o wartości 0, a rzeczywistym obrazom zostają przypisane etykiety o wartości 1. Dyskryminator uczy się na tej grupie tylko przez jeden krok za pomocą binarnej entropii krzyżowej jako funkcji kosztu. Co ważne, w tej fazie algorytm propagacji wstecznej optymalizuje jedynie wagi dyskryminatora.
- W drugiej fazie przechodzimy do uczenia generatora. Najpierw pozwalamy mu utworzyć kolejną grupę imitacji, po czym dyskryminator ma znowu sprawdzić, czy obrazy są prawdziwe. Tym razem jednak nie dodajemy do grupy żadnych rzeczywistych obrazów, a imitacje oznaczamy etykietami o wartości 1 (rzeczywiste) — inaczej mówiąc, chcemy aby generator tworzył obrazy, które dyskryminator będzie uważa (błędnie) za rzeczywiste! W fazie tej wagi dyskryminatora zostają zamrożone, dlatego algorytm propagacji wstecznej oddziałuje jedynie na wagi generatora.



Tak naprawdę generator nigdy nie otrzymuje rzeczywistych obrazów do oglądania, a mimo to stopniowo uczy się tworzyć coraz bardziej wiarygodne imitacje! Ma on dostęp wyłącznie do gradientów przepływających do początku dyskryminatora. Na szczęście im lepiej sobie radzi dyskryminator, tym więcej informacji na temat rzeczywistych obrazów jest zawartych w tych gradientach, dlatego generator może dzięki nim robić znaczne postępy.

Utwórzmy teraz prostą sieć GAN przeznaczoną dla zestawu danych Fashion MNIST.

Musimy najpierw zbudować generator i dyskryminator. Generator przypomina standardowy dekoder autokodera, natomiast dyskryminator to zwykły klasyfikator binarny (na wejściu przyjmuje obraz,

na jego końcu zaś znajduje się jednoneuronowa warstwa Dense wykorzystująca sigmoidalną funkcję aktywacji). W drugiej fazie każdego przebiegu uczenia będziemy potrzebować także pełnego modelu GAN, wyposażonego w generator, po którym znajduje się dyskryminator:

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Następnie musimy skompilować te modele. Dyskryminator jest klasyfikatorem binarnym, możemy więc w naturalny sposób wprowadzić binarną entropię krzyżową jako funkcję straty. Generator będzie uczyony wyłącznie poprzez sieć gan, zatem nie musimy go wcale kompilować. Model gan jest również klasyfikatorem binarnym, więc także tutaj możemy wykorzystać binarną entropię krzyżową. Co istotne, nie należy trenować dyskryminatora w drugiej fazie uczenia, dlatego przed skompilowaniem modelu gan sprawimy, że dyskryminator będzie niemodyfikowalny:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```



Interfejs Keras bierze pod uwagę atrybut trainable jedynie podczas kompilowania modelu, zatem po uruchomieniu kodu model **discriminator** **będzie** modyfikowalny, jeżeli wywołamy jego metodę `fit()` lub `train_on_batch()` (będziemy z niej korzystać), natomiast **nie będzie** modyfikowalny, jeżeli wywołamy te metody względem modelu `gan`.

Pętla uczenia jest niestandardowa, nie możemy więc użyć klasycznej metody `fit()`. Zamiast tego napiszemy własną pętlę uczenia. W tym celu musimy najpierw utworzyć obiekt `Dataset`, aby móc korzystać z zestawu danych:

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Możemy teraz utworzyć własną pętlę uczenia. Umieścmy ją w funkcji `train_gan()`:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # Faza pierwsza - uczenie dyskryminatora
            noise = tf.random.normal(shape=[batch_size, codings_size])
```

```

generated_images = generator(noise)
X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
discriminator.trainable = True
discriminator.train_on_batch(X_fake_and_real, y1)
# Faza druga - uczenie generatora
noise = tf.random.normal(shape=[batch_size, codings_size])
y2 = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size)

```

Jak już wiesz, obydwie fazy występują w każdej iteracji uczenia:

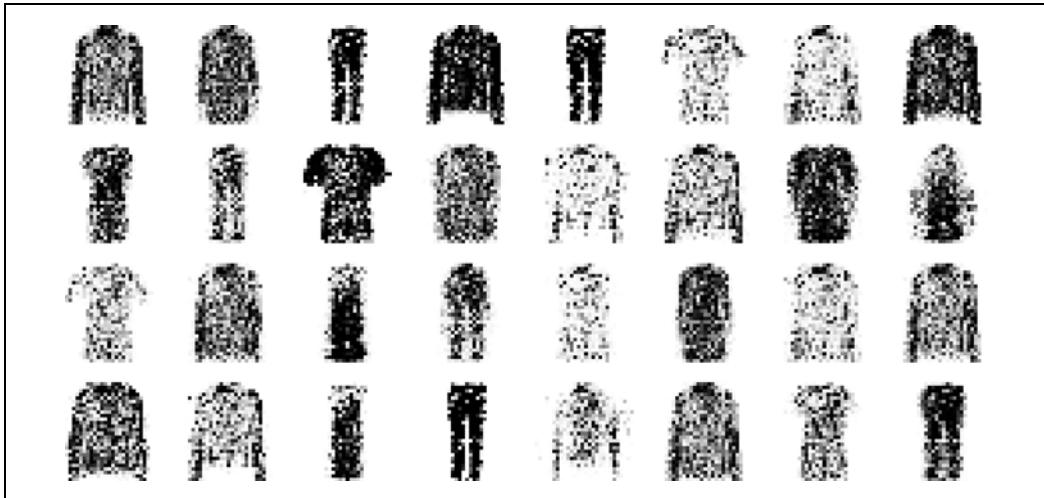
- W pierwszej fazie wprowadzamy do generatora szum gaussowski, który służy do tworzenia imitacji obrazów, i tak uzyskaną grupę przykładów uzupełniamy taką samą liczbą rzeczywistych obrazów. Etykiety y1 mają wartość 0 dla imitacji i 1 dla rzeczywistych obrazów. Następnie za pomocą tej grupy danych uczymy dyskryminatora. Zwróć uwagę, że wartością atrybutu trainable dyskryminatora jest tutaj True, co ma na celu jedynie uniknięcie ostrzeżenia wyświetlanego przez interfejs Keras w przypadku, gdy „zauważ”, że atrybut ten ma teraz wartość False, chociaż miał wartość True (lub odwrotnie) w czasie komplikacji modelu.
- W drugiej fazie dostarczamy szum gaussowski do sieci GAN. Jej generator tworzy imitacje obrazów, a dyskryminator stara się stwierdzić, które obrazy są rzeczywiste. Chcemy, by uwierzył, że imitacje są rzeczywistymi przykładami, dlatego etykiety y2 mają wartość 1. Zwróć uwagę, że wyznaczamy tu wartość False w atrybucie trainable, a powód jest taki sam jak w pierwszej fazie.

To wszystko! Jeżeli wyświetlisz wygenerowane imitacje (rysunek 17.16), to zauważysz, że już pod koniec pierwszej epoki zaczynają przypominać obrazy (bardzo zaszumione) z zestawu Fashion MNIST.

Niestety imitacje rzadko kiedy wyglądają lepiej od zaprezentowanych na rysunku 17.16, a zdarzają się wręcz epoki, w których sieć GAN zdaje się zapominać o dokonanych postępach. Dlaczego tak się dzieje? Okazuje się, że uczenie sieci GAN może stanowić wyzwanie. Zobaczmy, dlaczego tak jest.

Problemy związane z uczeniem sieci GAN

W trakcie uczenia obydwie sieci starają się wzajemnie przechytrzyć w grze o sumie stałej. W miarę postępów gra może utknąć w punkcie zwanym przez teoretyków **równowagą Nasha** (ang. *Nash equilibrium*, od nazwiska matematyka Johna Nasha): jest to moment, w którym żadnemu z graczy nie opłaca się zmieniać bieżącej strategii działania, jeżeli przeciwnik nie zrobi tego pierwszy. Na przykład równowaga Nasha zostaje osiągnięta, gdy wszyscy kierowcyjadą lewą stroną drogi: żadnemu kierowcy nie opłaca się zmiana strony. Istnieje oczywiście też druga równowaga Nasha: wszyscy mogą jechać prawą stroną drogi. Różne stany początkowe i dynamika układu mogą prowadzić do którejś z tych równowag. W tym przykładzie istnieje jedna optymalna strategia po osiągnięciu równowagi Nasha (jazda po tej samej stronie co pozostali kierowcy), ale w innych przypadkach mogą występować różne, współzawodniczące ze sobą strategie (np. drapieżnik ściga zwierzynę łowną, zwierzyna stara się uciec i żaden z „graczy” nie może sobie pozwolić na zmianę strategii).



Rysunek 17.16. Obrazy wygenerowane przez sieć GAN po jednej epoce uczenia

Jak to się więc ma do sieci GAN? Autorzy publikacji udowodnili, że sieć GAN może osiągnąć tylko jedną równowagę Nasha: wtedy, gdy generator tworzy doskonałe imitacje obrazów, a dyskryminator zostaje zmuszony do zgadywania (po 50% szans na to, że obraz jest albo rzeczywisty, albo imitacją). Teoretycznie jest to bardzo dobra informacja: oznaczałoby to, że wystarczy uczyć sieć GAN wystarczająco długo, aby osiągnąć ten stan równowagi, i otrzymalibyśmy idealny generator. Niestety nie jest to takie proste, nie mamy bowiem żadnej gwarancji, że równowaga zostanie kiedykolwiek osiągnięta.

Największą trudność stanowi tzw. **załamanie modu** (ang. *mode collapse*). Występuje ono wtedy, gdy wyniki generatora stopniowo stają się coraz mniej zróżnicowane. Jak do tego dochodzi? Założymy, że generatorowi udaje się tworzyć bardziej przekonujące obrazy butów niż innych klas. Będzie udawało mu się oszukiwać dyskryminator nieco częściej imitacjami tej klasy, co w konsekwencji będzie prowadziło do generowania jeszcze większej liczby imitacji obrazów butów. Generator będzie stopniowo zapominał, jak należy tworzyć imitacje innych klas. Jednocześnie jedynymi imitacjami dostarczonymi do dyskryminatora będą obrazy butów, zatem zacznie zapominać, jak należy rozpoznawać imitacje pozostałych klas. W końcu, gdy dyskryminator nauczy się odróżniać imitacje obrazów butów od rzeczywistych przykładów, generator będzie zmuszony zająć się inną klasą. Może zacząć tworzyć doskonałe imitacje obrazów bluz i stopniowo zapominać o imitacjach butów, co przełoży się na skuteczność dyskryminatora. Sieć GAN może cyklicznie koncentrować się na kilku klasach i nigdy nie osiągnąć doskonałych wyników w żadnej z nich.

Ponadto generator i dyskryminator ciągle ze sobą „walczą”, dlatego ich parametry mogą oscylować i stać się niestabilne. Proces uczenia może rozpocząć się prawidłowo, a następnie bez wyraźnego powodu stać się rozbieżny w wyniku tych niestabilności. Wiele czynników wpływa na dynamikę tego układu i przez to sieci GAN są bardzo czułe na wartości hiperparametrów, a ich strojenie niekiedy zabiera mnóstwo czasu.

Problemy te spędzały badaczom sen z powiek już od 2014 roku. Opublikowano na ten temat wiele prac — w niektórych proponowano nowe funkcje kosztu¹¹ (choć w przygotowanym przez firmę Google artykule z 2018 roku (<https://arxiv.org/abs/1711.10337>)¹² została podważona ich skuteczność), w innych opracowano techniki stabilizowania procesu uczenia lub unikania załamania modu. Przykładowo popularne rozwiążanie zwane **odtwarzaniem doświadczenia** (ang. *experience replay*) polega na przechowywaniu obrazów utworzonych w każdej iteracji przez generator wewnątrz bufora odtwarzania (z którego stopniowo są usuwane starsze imitacje) i uczeniu dyskryminatora za pomocą kombinacji rzeczywistych obrazów i imitacji przechowywanych w tym buforze (a nie imitacji tworzonych na bieżąco przez generator). Zmniejszamy w ten sposób ryzyko przetrenowania najnowszych wyników generatora przez dyskryminator. Inną popularną techniką jest **rozróżnianie miniwsadami** (ang. *mini-batch discrimination*): mierzono jest tutaj podobieństwo obrazów w danej grupie, po czym statystyka ta jest przekazywana dyskryminatorowi, dzięki czemu z łatwością może odrzucić całą grupę imitacji o zbyt małym zróżnicowaniu. W ten sposób generator jest zmuszany do tworzenia różnorodnych obrazów, co zmniejsza ryzyko załamania modu. W jeszcze innych publikacjach autorzy proponują architektury, które w danych warunkach sprawdzają się bardzo dobrze.

Mówiąc krótko, dziedzina ta jest ciągle aktywnie rozwijana i dynamika sieci GAN nie została jeszcze w pełni zrozumiana. Dobre wieści są jednak takie, że poczyniono olbrzymie postępy i niektóre rezultaty są doprawdy zdumiewające! Przejrzymy się teraz najpopularniejszym strukturalnie, począwszy od głębokich splotowych sieci GAN, które jeszcze kilka lat temu stanowiły najnowocześniejsze osiągnięcie. Następnie przejrzymy się dwóm nowszym (i bardziej skomplikowanym) architekturam.

Głębokie splotowe sieci GAN

W artykule z 2014 roku autorzy eksperymentowali z warstwami splotowymi, ale próbowały generować jedynie małe obrazy. Niedługo później wielu naukowców próbowało stworzyć sieci GAN oparte na głębszych sieciach splotowych, umożliwiających generowanie większych obrazów. Zadanie to okazało się niełatwwe, ponieważ proces uczenia był bardzo niestabilny, ale Alec Radford i in. w końcu dokonali tego pod koniec 2015 roku, po przeprowadzeniu wielu prób na różnych architekturach i hiperparametrach. Nowa struktura została nazwana **głęboką splotową siecią GAN** (ang. *deep convolutional GAN* — DCGAN; <https://arxiv.org/abs/1511.06434>)¹³. Oto główne wytyczne tworzenia stabilnych splotowych sieci GAN zaproponowane przez autorów:

- Zastąp wszystkie warstwy łączące kroczącymi warstwami splotowymi (w dyskryminatorze) i transponowanymi warstwami splotowymi (w generatorze).
- Wprowadź normalizację wsadową zarówno w dyskryminatorze, jak i w generatorze (oprócz warstwy wyjściowej generatora i warstwy wejściowej dyskryminatora).

¹¹ Dobre porównanie głównych funkcji straty stosowanych w sieciach GAN znajdziesz w doskonałym projekcie GitHub, którego autorem jest Hwalsuk Lee (<https://github.com/hwalsuklee/tensorflow-generative-model-collections>).

¹² Mario Lucic i in., *Are GANs Created Equal? A Large-Scale Study*, „Proceedings of the 32nd International Conference on Neural Information Processing Systems” (2018), s. 698 – 707.

¹³ Alec Radford i in., *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, arXiv preprint arXiv:1511.06434 (2015).

- W głębszych architekturach usuń w pełni połączone warstwy ukryte.
- W niemal wszystkich warstwach generatora wprowadź funkcję aktywacji ReLU. Jedynym wyjątkiem jest warstwa wyjściowa, która powinna wykorzystywać funkcję tanh.
- We wszystkich warstwach dyskryminatora wprowadź „przeciekającą” funkcję aktywacji ReLU.

Wskazówki te sprawdzają się w większości przypadków, ale nie zawsze, dlatego być może przyjdzie Ci poeksperymentować z różnymi hiperparametrami (w rzeczywistości czasami wystarczy zmiana zarodka losowości i ponowne wyuczenie modelu). Oto przykład malej sieci DCGAN, która całkiem dobrze radzi sobie z zestawem Fashion MNIST:

```

codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                               activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                               activation="tanh")
])

discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                      activation=keras.layers.LeakyReLU(0.2),
                      input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                      activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])

gan = keras.models.Sequential([generator, discriminator])

```

Generator przyjmuje kodowania o rozmiarze 100 i rzutuje je na 6272 wymiary ($7 \times 7 \times 128$), po czym przekształca rezultat do postaci tensora o wymiarach $7 \times 7 \times 128$. Tensor ten zostaje poddany normalizacji wsadowej i przekazany do transponowanej warstwy splotowej o kroku równym 2, co powoduje zwiększenie rozdzielczości z 7×7 do 14×14 i zmniejsza głębokość ze 128 na 64. Uzyskany wynik znów zostaje poddany normalizacji wsadowej i przesłany do kolejnej transponowanej warstwy splotowej o kroku 2, gdzie następuje ponowne zwiększenie rozdzielczości (z 14×14 na 28×28) przy jednoczesnym zredukowaniu głębokości z 64 do 1. Warstwa ta zawiera funkcję aktywacji tanh, zatem zakres wartości wynikowych będzie mieścił się w przedziale od -1 do 1. Z tego powodu, zanim przystąpimy do uczenia sieci GAN, musimy przeskalać zestaw danych tak, aby mieścił się w tym samym zakresie. Musimy także dodać do niego wymiar kanału:

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # Zmiana wymiarów i przeskalowanie
```

Dyskryminator przypomina tradycyjną sieć splotową służącą do klasyfikacji binarnej, ale zastępujemy tutaj maksymalizujące warstwy łączące (zmniejszające rozmiar obrazu) kroczącymi warstwami splotowymi (strides=2). Ponadto używamy tu „przeciekającej” funkcji ReLU.

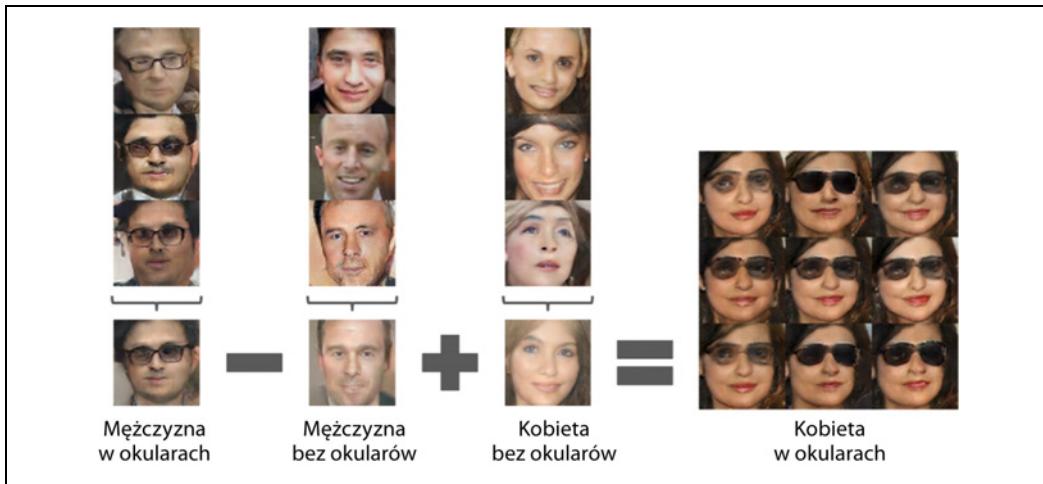
Zasadniczo przestrzegamy tu wspomnianych wytycznych, jedynie zastąpiliśmy warstwy Batch Normalization w dyskryminatorze warstwami Dropout (gdybyśmy tego nie zrobili w tym przypadku, proces uczenia byłby niestabilny), a w generatorze zamiast funkcji ReLU użyliśmy funkcji SELU. Nie bój się modyfikować tej architektury, a przekonasz się, jak bardzo jest ona czuła na wartości hiperparametrów (zwłaszcza względnych współczynników uczenia dwóch sieci).

Do utworzenia zestawu danych, skompilowania i wytrenowania modelu wykorzystujemy ten sam kod co poprzednio. Po 50 epokach uczenia generator zacznie tworzyć obrazy podobne do zaprezentowanych na rysunku 17.17. Ciągle brakuje im trochę do ideału, ale wiele z nich jest już całkiem przekonujących.



Rysunek 17.17. Obrazy wygenerowane przez sieć DCGAN po 50 epokach uczenia

Gdybyśmy przeskalowali tę architekturę i dostosowali ją do poznawania dużego zestawu zdjęć twarzy, generowałby ona całkiem realistyczne imitacje. Sieci DCGAN są w stanie poznawać całkiem istotne reprezentacje ukryte, co widać na rysunku 17.18: zostało wygenerowanych wiele obrazów i wybrano ręcznie dziewięć z nich (lewa góra strona), a mianowicie obrazy trzech mężczyzn w okularach, trzech mężczyzn bez okularów i trzech kobiet bez okularów. W przypadku każdej z tych kategorii uśredniono kodowania użyte do wygenerowania obrazów i z tej średniej uzyskano obraz widoczny pod każdą kategorią (lewa dolna strona). Inaczej mówiąc, każdy z obrazów widocznych w lewej dolnej części rysunku 17.18 stanowi średnią z trzech obrazów widocznych powyżej. Nie jest to jednak standardowa średnia obliczona na poziomie pikseli (uzyskalibyśmy w ten sposób trzy nakładające się twarze), lecz średnia obliczona w przestrzeni ukrytej, zatem na obrazach wynikowych nadal widać realistyczne twarze. Zdumiewający jest fakt, że po wygenerowaniu obrazu z połączenia obrazów twarzy mężczyzn w okularach, mężczyzn bez okularów i kobiet bez okularów (gdzie każda klasa odpowiada jednemu z uśrednionych kodowań) otrzymamy w środku widocznej po prawej stronie siatki 3×3 obraz twarzy kobiety w okularach! Osiem pozostałych obrazów w siatce zostało wygenerowanych za pomocą tego samego wektora z dodatkową szumem, co ma za zadanie unaoczyć potencjał interpolacji semantycznej w sieci DCGAN. Przeprowadzanie operacji arytmetycznych na zdjęciach twarzy przypomina fantastykę naukową!



Rysunek 17.18. Arytmetyka wektorowa w koncepcjach wizualnych (część rysunku 7. z publikacji prezentującej sieć DCGAN)¹⁴



Jeżeli dadasz klasę każdego obrazu jako dodatkowy sygnał wejściowy zarówno do generatora, jak i do dyskryminatora, obydwie sieci dowiedzą się, jak ta klasa wygląda, więc będziesz w stanie kontrolować klasę każdego obrazu stworzonego przez generator. Jest to tzw. **warunkowa sieć GAN** (ang. *conditional GAN* — CGAN; <https://arxiv.org/abs/1411.1784>)¹⁵.

Sieci DCGAN nie są jednak doskonałe. Jeżeli na przykład spróbujesz za ich pomocą wygenerować bardzo duże obrazy, nierzaz uzyskasz rezultat, w którym poszczególne, lokalne elementy będą wiarygodne, ale cały obraz będzie zawierał mnóstwo niespójności (np. koszulki, których jeden rękaw jest znacznie dłuższy od drugiego). Jak możemy rozwiązać ten problem?

Rozrost progresywny sieci GAN

Tero Karras wraz z innymi badaczami z firmy Nvidia przedstawił ważną technikę w publikacji z 2018 roku (<https://arxiv.org/abs/1710.10196>)¹⁶: zaproponował, aby na początku uczenia generować małe obrazy, a następnie stopniowo dodawać kolejne warstwy splotowe zarówno w generatorze, jak i w dyskryminatorze po to, aby uzyskiwać coraz większe obrazy (4×4 , 8×8 , 16×16 , ..., 512×512 , 1024×1024). Metoda ta przypomina zachłanne uczenie warstwowe autokoderów stosowych. Dodatkowe warstwy są umieszczane na końcu generatora i na początku dyskryminatora, natomiast wcześniej wytrenowane warstwy pozostają modyfikowalne.

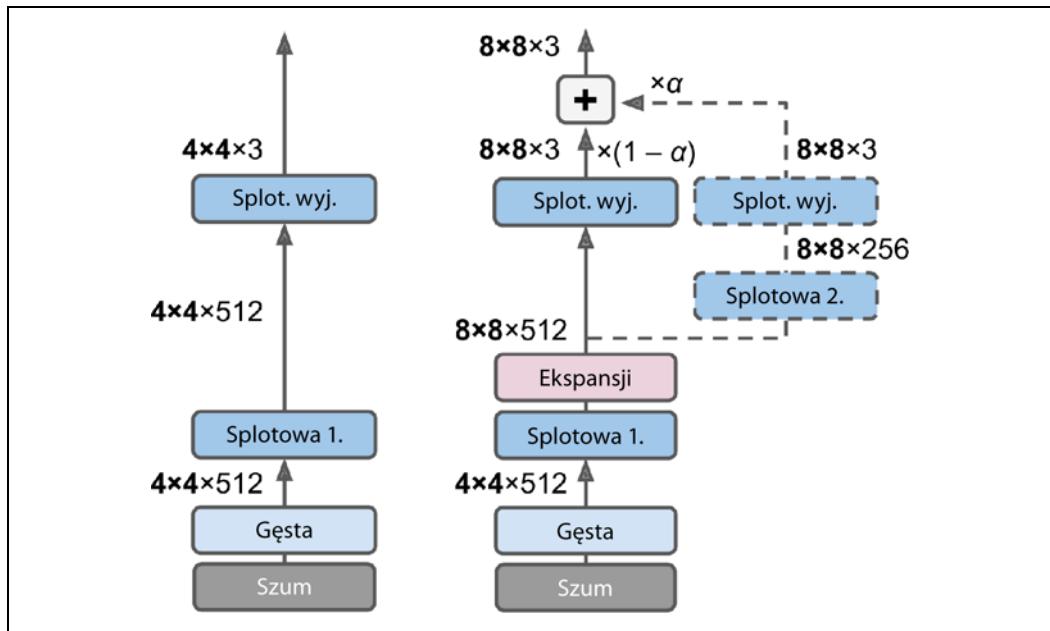
Na przykład podczas powiększania wyników generatora z rozmiaru 4×4 na 8×8 (rysunek 17.19) warstwa ekspansji (za pomocą filtrowania najbliższego sąsiada) zostaje dodana do już istniejącej

¹⁴ Rysunek ten został umieszczony w mniejszej książce za zgodą autorów.

¹⁵ Mehdi Mirza i Simon Osindero, *Conditional Generative Adversarial Nets*, arXiv preprint arXiv: 1411.1784 (2014).

¹⁶ Tero Karras i in., *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, „Proceedings of the International Conference on Learning Representations” (2018).

warstwy splotowej, zatem generuje mapy cech 8×8 , które następnie są przekazywane do następnej warstwy splotowej (zawierającej uzupełnianie zerami typu "same" i krok o wartości równej 1, dzięki czemu jej wyniki mają również rozmiar 8×8). Po tej nowej warstwie występuje nowa, wyjściowa warstwa splotowa: jest to tradycyjna warstwa splotowa o jądrze równym 1, która rzutuje wyniki na wymaganą liczbę kanałów kolorów (np. 3). Aby nie „popsuć” wyuczonych wag pierwszej warstwy splotowej w czasie dołączania nowej warstwy konwolucyjnej, wynik końcowy stanowi sumę ważoną pierwotnej warstwy wyjściowej (która generuje teraz mapy cech 8×8) i nowej warstwy wyjściowej. Waga nowych sygnałów wyjściowych ma wartość α , w przypadku pierwotnych sygnałów wyjściowych jest to $1 - \alpha$, natomiast α stopniowo rośnie od 0 do 1. Innymi słowy, nowe warstwy splotowe (symbolizowane linią przerywaną na rysunku 17.19) są stopniowo wzmacniane, a pierwotna warstwa wyjściowa stopniowo zanika. Podobna technika wzmacniania/osłabiania jest stosowana podczas dołączania nowej warstwy splotowej w dyskryminatorze (po niej zaś występuje uśredniająca warstwa łącząca, która służy do zmniejszania obrazu wyjściowego).



Rysunek 17.19. Rozrost progresywny sieci GAN: generator w sieci GAN tworzy obrazy kolorowe o rozmiarach 4×4 (po lewej), a my powiększamy je do rozmiarów 8×8 (po prawej)

We wspomnianej publikacji zaproponowano również kilka innych technik zwiększających zróżnicowanie generowanych wyników (w celu uniknięcia załamania modu) i stabilizujących proces uczenia:

Warstwa odchylenia standardowego miniwsadów (ang. minibatch standard deviation layer)

Jest dodawana pod koniec dyskryminatora. Dla każdej pozycji w danych wejściowych oblicza ona odchylenie standardowe wzduż wszystkich kanałów i przykładów w grupie danych ($S = \text{tf.math.reduce_std}(\text{inputs}, \text{axis}=[0, -1])$). Odchylenia te są następnie uśredniane dla wszystkich punktów, co pozwala uzyskać pojedynczą wartość ($v = \text{tf.reduce_mean}(S)$).

Na koniec do każdego przykładu w grupie zostaje dołączona dodatkowa mapa cech, która zostaje wypełniona wynikiem operacji `tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)`. W jaki sposób ma to nam pomóc? Jeżeli generator produkuje mało zróżnicowane obrazy, to mapy cech w dyskryminatorze będą cechować się małym zróżnicowaniem. Dzięki tej warstwie dyskryminator będzie miał łatwy dostęp do tej statystyki i nie da się łatwo oszukać generatorowi generującemu obrazy o zbyt małym zróżnicowaniu. Z tego powodu generator będzie produkował bardziej różnorodne obrazy, co zapobiegnie załamaniu modu.

Wyrównany współczynnik uczenia (ang. *equalized learning rate*)

Inicjalizuje wszystkie wagi za pomocą prostego rozkładu gaussowskiego o średniej równej 0 i odchyleniu standardowym równym 1 (czyli rezygnujemy z inicjalizacji He). Wagi są jednak skalowane w czasie działania modelu (za każdym razem, gdy ta warstwa jest wykorzystywana) o taki sam współczynnik jak w inicjalizacji He: dzielone są przez wartość $\sqrt{2 / n_{wejścia}}$, gdzie $n_{wejścia}$ określa liczbę wejść do warstwy. Udowodniono, że technika ta znaczowo poprawia wydajność sieci GAN podczas korzystania z optymalizatora RMSProp, Adam lub innego adaptacyjnego optymalizatora gradientów. Istotnie, optymalizatory te normalizują aktualizacje gradientów o ich oszacowane odchylenie standardowe (zob. rozdział 11.), zatem parametry o większym zakresie dynamicznym¹⁷ będą potrzebowaly więcej czasu na naukę, natomiast parametry o małym zakresie dynamicznym mogą być aktualizowane zbyt szybko, co prowadzi do niestabilności. Poprzez włączenie skalowania wag do samego modelu, a nie skalowania ich wyłącznie w fazie inicjalizacji zapewniamy taki sam zakres dynamiczny dla wszystkich parametrów w czasie uczenia, dzięki czemu będą się one uczyć z taką samą szybkością. Rozwiążanie to zarówno przyspiesza, jak i stabilizuje proces uczenia.

Warstwa normalizacji piksel po pikselu (ang. *pixelwise normalization layer*)

Dodawana jest po każdej warstwie splotowej generatora. Normalizuje każdą funkcję aktywacji na podstawie wszystkich aktywacji tego samego obrazu i w tym samym miejscu, ale wzduż wszystkich kanałów (wykonujemy operację dzielenia przez pierwiastek kwadratowy z pobudzenia średniokwadratowego). W module TensorFlow zapisujemy to jako `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (człon wygładzający 1e-8 służy zapobieganiu dzielenia przez 0). Dzięki tej technice unikamy eksplozji pobudzeń wynikającej z nadmiernego współzawodnictwa pomiędzy generatorem i dyskryminatorem.

Kombinacja tych technik pozwoliła autorom wygenerować niezwykle wiarygodne obrazy twarzy w wysokiej rozdzielcości (<https://www.youtube.com/watch?v=G06dEcZ-QTg>). Co jednak dokładnie rozumiemy przez „wiarygodne”? Ocenianie modelu jest jednym z większych wyzwań w pracy z sieciami GAN: można automatycznie ocenić zróżnicowanie generowanych obrazów, ale określenie ich jakości to znacznie bardziej podchwytliwe i subiektywne zadanie. Jednym ze sposobów jest zatrudnienie ludzkich recenzentów, ale jest ono kosztowne i czasochłonne. Z tego powodu autorzy zaproponowali wskaźnik podobieństwa struktury lokalnej wygenerowanych obrazów i obrazów

¹⁷ Zakresem dynamicznym zmiennej nazywamy stosunek pomiędzy największą a najmniejszą wartością, jaką ta zmienna może przyjąć.

uczących przy uwzględnieniu każdej skali. Koncepcja ta doprowadziła do kolejnej przełomowej innowacji: sieci StyleGAN.

Sieci StyleGAN

Poprzeczka w postępach w generowaniu obrazów o dużej rozdzielczości została ponownie podniesiona przez ten sam zespół z firmy Nvidia w 2018 roku (<https://arxiv.org/abs/1812.04948>)¹⁸, w publikacji, w której opisano popularną architekturę StyleGAN. Autorzy wykorzystali techniki **przenoszenia stylów** (ang. *style transfer*) w generatorze, dzięki którym generowane obrazy mają taką samą strukturę lokalną (w każdej skali) jak obrazy uczące, co znacznie poprawia jakość generowanych obrazów. Dyskryminator i funkcja straty nie są modyfikowane, jedynie generator. Przyjrzymy się modelowi StyleGAN. Składa się on z dwóch sieci (rysunek 17.20):

Sieć mapująca (ang. *mapping network*)

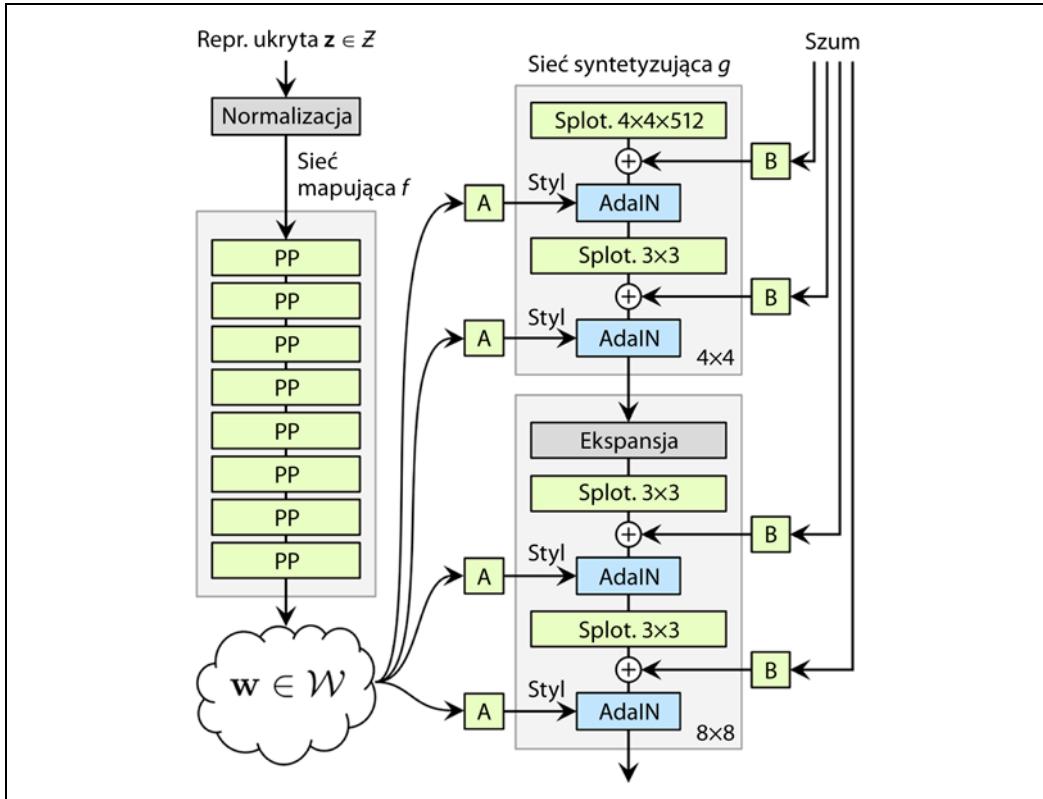
Ośmiowarstwowy perceptron wielowarstwowy mapujący reprezentacje ukryte z (tzn. kodowania) na wektor w . Wektor ten jest następnie przesyłany przez wiele **przekształceń afinicznych** (ang. *affine transformations*; są to warstwy Dense pozbawione funkcji aktywacji, oznaczone polami A na rysunku 17.20), które tworzą wiele wektorów. Wektory te określają styl generowanego obrazu na różnych poziomach, od szczegółowych struktur (np. kolor włosów) aż do ogólnych cech (np. osoba dorosła lub dziecko). Mówiąc krótko, sieć mapująca przenosi kodowania na wiele wektorów stylów.

Sieć syntetyzująca (ang. *synthesis network*)

Jest odpowiedzialna za generowanie obrazów. Zawiera wyuczone, stałe wejście (mówiąc dokładniej, wejście to pozostaje stałe po fazie uczenia, ale w *trakcie* uczenia jest aktualizowane przez algorytm propagacji wstecznej). Podobnie jak w poprzednim modelu przetwarza ona sygnał wejściowy poprzez wiele warstw splotowych i ekspansji, ale zawiera dwie modyfikacje. Po pierwsze, do danych wejściowych zostaje dodzony pewien szum, jak również do wszystkich wyjść warstw splotowych (przed funkcją aktywacji). Po drugie, po każdej warstwie zaszumienia zostaje dodana warstwa **AdaIN** (ang. *Adaptive Instance Normalization* — warstwa adaptacyjnej normalizacji przykładów), która standaryzuje niezależnie od pozostałych każdą mapę cech (poprzez odjęcie średniej wartości mapy cech i podzielenie przez jej odchylenie standarde), a następnie wykorzystuje wektor stylu do wyznaczenia skali i przesunięcia każdej mapy cech (wektor stylu zawiera po jednej skali i jednym członie obciążenia na każdą mapę cech).

Bardzo istotna jest koncepcja dodania szumu niezależnie od kodowań. Niektóre elementy obrazu, takie jak dokładne położenia każdego piega lub włosia, są losowe. W starszych modelach GAN losowość ta musiała pochodzić albo z kodowań, albo jakiegoś szumu pseudolosowego produkowanego przez sam generator. Jeżeli szum ten był zawarty w kodowaniach, generator musiał poświęcić znaczną ilość mocy reprezentacyjnej kodowań na przechowanie tych informacji o szumie, co jest sporym marnotrawstwem. Ponadto szum ten musi przepływać przez sieć i dotrzeć do końcowych

¹⁸ Tero Karras i in., *A Style-Based Generator Architecture for Generative Adversarial Networks*, arXiv preprint arXiv:1812.04948 (2018).



Rysunek 17.20. Architektura generatora StyleGAN (część rysunku 1. z publikacji omawiającej sieć StyleGAN)¹⁹

warstw generatora, to zaś wydaje się niepotrzebnym ograniczeniem, które prawdopodobnie spowalniało proces uczenia. Nie zapominajmy również, że jeżeli ten sam szum był używany na różnych poziomach, to mogło się to skończyć pojawiением się jakichś artefaktów graficznych. Z kolei jeżeli generator starał się produkować własny szum pseudolosowy, to mógł być on niezbyt przekonujący, co również prowadziłoby do artefaktów graficznych, a do tego trzeba byłoby przeznaczyć część wag generatora na tworzenie szumu pseudolosowego, co też jest marnotrawstwem. Wszystkie te problemy zostają rozwiązane po wprowadzeniu dodatkowych sygnałów przenoszących szum; sieć GAN może wykorzystać taki dołączony szum do zwiększenia losowości każdej części obrazu.

Dołączony szum jest odmienny na każdym poziomie. Każdy sygnał wejściowy szumu zawiera pojedynczą mapę cech wypełnioną szumem gaussowskim, który jest przesyłany do wszystkich map cech (na danym poziomie) i przed dołączeniem zostaje przeskalowany za pomocą wyuczonych współczynników skalowania (pola B na rysunku 17.20).

Ostatnią nowością w sieci StyleGAN jest technika zwana **regularyzacją mieszającą** (ang. *mixing regularization*) lub **mieszaniem stylów** (ang. *style mixing*), w której pewien odsetek generowanych obrazów jest uzyskiwany za pomocą dwóch różnych kodowań. Mówiąc dokładniej, kodowania c_1 i c_2

¹⁹ Rysunek ten został umieszczony w niniejszej książce za zgodą autorów.

są przesyłane przez sieć mapującą, co pozwala uzyskać wektory stylu w_1 i w_2 . Następnie sieć syntetyzująca generuje obraz na podstawie stylów w_1 na pierwszych poziomach i stylów w_2 na pozostałych poziomach. Stosunek poziomów wyznaczanych poszczególnym wektorom dobierany jest losowo. W ten sposób sieć nie będzie zakładała, że style na sąsiadujących poziomach są ze sobą skorelowane, co z kolei sprzyja lokalności w sieciach GAN, a to oznacza, że każdy wektor wpływa jedynie na ograniczoną liczbę cech generowanego obrazu.

Istnieje tak wiele odmian sieci GAN, że można na ich temat napisać osobną książkę. Mam nadzieję, że dzięki niniejszemu wprowadzeniu rozumiesz jej główne założenia, a co ważniejsze, być może zaszczepięm w Tobie chęć jej dokładniejszego zrozumienia. Jeżeli aparat matematyczny sprawia Ci problem, na pewno istnieją blogi ułatwiające zrozumienie wzorów. Zaimplementuj śmiało swoją własną sieć GAN i nie zrażaj się, jeżeli na początku będzie miała problemy z nauką — niestety jest to normalne i wymaga odrobiny cierpliwości, ale warto poczekać na rezultaty. Jeśli nie radzisz sobie ze szczegółami implementacji, istnieje mnóstwo implementacji w module TensorFlow i interfejsie Keras, które możesz dogłębiście przeanalizować. Jeżeli zależy Ci wyłącznie na szybkim uzyskaniu zdumiewających wyników, możesz po prostu skorzystać z jakiegoś gotowego modelu (np. istnieją gotowe modele StyleGAN wykorzystujące interfejs Keras).

W następnym rozdziale zajmiemy się całkowicie odmiennym działem uczenia głębokiego: głębokim uczeniem przez wzmacnianie.

Ćwiczenia

1. Do jakich zadań są przede wszystkim wykorzystywane autokodery?
2. Załóżmy, że chcesz wytrenować klasyfikator i masz wiele nieoznakowanych danych, ale zaledwie kilka tysięcy próbek zawierających etykiety. W jaki sposób mogą pomóc autokodery? Jakie czynności należy wykonać?
3. Jeżeli autokoder idealnie rekonstruuje dane wejściowe, to czy jest przydatny? W jaki sposób można zmierzyć wydajność autokodera?
4. Czym są autokodery niedopełnione i przepelnione? Jakie ryzyko wiąże się z nadmiernie niedopełnionym autokoderem? Dlaczego nie należy tworzyć zbyt przepelnionego autokodera?
5. Jak wiążemy wagi w autokoderze stosowym? Po co to robimy?
6. Czym jest model generatywny? Czy znasz jakiś rodzaj autokodera generatywnego?
7. Czym jest sieć GAN? Wymień kilka zadań, w których wyróżniają się modele GAN.
8. Jakie są podstawowe trudności związane z uczeniem sieci GAN?
9. Spróbuj wyuczyć wstępnie klasyfikator obrazów za pomocą autokodera odszumiającego. Możesz skorzystać ze zbioru obrazów MNIST (najprostsze rozwiązanie) lub, jeśli szukasz wyzwania, z takiego zbioru jak CIFAR10 (<https://www.cs.toronto.edu/~kriz/cifar.html>). Bez względu na wybrany zestaw danych wykonaj następujące czynności:
 - a. Podziel zestaw danych na podzbiory uczący i testowy. Wyucz głęboki autokoder odszumiający na pełnym zbiorze uczącym.

- b.** Sprawdź, czy obrazy zostały w miarę wiernie zrekonstruowane. Zwizualizuj obrazy najbardziej pobudzające każdy neuron w warstwie kodowania.
 - c.** Utwórz klasyfikującą głęboką sieć neuronową, ponownie wykorzystując niższe warstwy autokodera. Wytrenuj je za pomocą zaledwie 500 przykładów z zestawu uczącego. Kiedy model uzyskuje lepszą wydajność: po zastosowaniu uczenia wstępnego czy bez tej techniki?
- 10.** Wyucz autokoder wariacyjny na dowolnym zestawie danych i spraw, żeby generował obrazy. Ewentualnie możesz znaleźć interesujący Cię nieoznakowany zbiór danych i sprawdzić, czy autokoder będzie w stanie generować z niego nowe przykłady.
- 11.** Wytrenuj sieć DCGAN przetwarzającą dowolnie wybrany zestaw danych i użyj jej do wygenerowania obrazów. Dodaj funkcję odtwarzania doświadczenia i sprawdź, czy uzyskasz dzięki niej lepsze rezultaty. Przekształć model w warunkową sieć GAN, w której możesz kontrolować generowaną klasę.

Rozwiązań ćwiczeń znajdziesz w dodatku A.

Uczenie przez wzmacnianie

Uczenie przez wzmacnianie (ang. *Reinforcement Learning* — RL) stanowi obecnie jedną z najbardziej fascynujących, a zarazem najstarszych dziedzin uczenia maszynowego. Znana jest ona już od lat 50. ubiegłego wieku i znalazła wiele przydatnych zastosowań¹, zwłaszcza w grach (np. TD-Gammon — programie grającym w tryktrak) i sterowaniu urządzeniami, rzadko jednak pojawia się na pierwszych stronach gazet. Przełom nastąpił w 2013 roku, gdy twórcy projektu DeepMind zaprezentowali system potrafiący przejść niemal każdą grę stworzoną na komputer Atari (<https://arxiv.org/abs/1312.5602>)², a nawet w większości przypadków osiągać lepsze wyniki od człowieka (<https://homl.info/dqn2>)³ — danymi wejściowymi w tym modelu są zwykłe piksele, a ponadto musi on samodzielnie odkryć reguły rządzące każdą z gier⁴. Było to pierwsze z szeregu zdumiewających dokonań zespołu, których kulminacja miała miejsce w marcu 2016 roku, gdy system AlphaGo pokonał Lee Sedola, legendarnego gracza zawodowego w grę Go, a w maju 2017 roku odniósł zwycięstwo nad Ke Jie, mistrzem świata w tej grze. Żadnemu innemu programowi nie udało się wcześniej nawet zbliżyć do podobnego rezultatu, zwłaszcza przeciwko najlepszemu ludzkiemu graczowi. Dzisiaj cały sektor uczenia przez wzmacnianie wprost buzuje nowymi pomysłami i znajduje zastosowanie w coraz to nowszych dziedzinach. System DeepMind został w 2014 roku wykupiony przez firmę Google za ponad 500 milionów dolarów.

Jak udało się więc tego dokonać? Teraz może wydawać się to banalne: twórcy wykorzystali potęgę uczenia głębokiego w dziedzinie uczenia przez wzmacnianie, a wyniki przerosły ich najśmiesze oczekiwania. W tym rozdziale najpierw wyjaśnię, czym jest uczenie przez wzmacnianie i do czego się przydaje, a następnie omówię dwie najważniejsze techniki używane w głębokim uczeniu przez wzmacnianie: **gradienty strategii** (ang. *policy gradients*) i **głębokie sieci Q** (ang. *Deep Q-networks* — DQN), przy czym poruszę również zagadnienie **procesów decyzyjnych Markowa** (ang. *Markov Decision Processes* — MDP). Zastosujemy te metody do utrzymywania masztu w równowadze na pędzącym wózku.

¹ Więcej szczegółów znajdziesz w książce poświęconej uczeniu przez wzmacnianie: Richard Sutton i Andrew Barto, *Reinforcement Learning: An Introduction*, MIT Press (<http://www.incompleteideas.net/book/the-book-2nd.html>).

² Volodymyr Mnih i in., *Playing Atari with Deep Reinforcement Learning*, arXiv preprint arXiv:1312.5602 (2013).

³ Volodymyr Mnih i in., *Human-Level Control Through Deep Reinforcement Learning*, „Nature” 518 (2015), s. 529 – 533.

⁴ Na stronie <https://homl.info/dqn3> znajdziesz filmik pokazujący system DeepMind uczący się takich gier jak Space Invaders, Breakout i in.

Następnie zaprezentuję bibliotekę TF-Agents, która wykorzystuje najnowocześniejsze algorytmy, znacznie upraszczające budowanie potężnych systemów RL (wytrenujemy za jej pomocą agenta grającego w słynną grę na Atari, Breakout). Na zakończenie rozdziału przybliżę najnowsze osiągnięcia w omawianej dziedzinie.

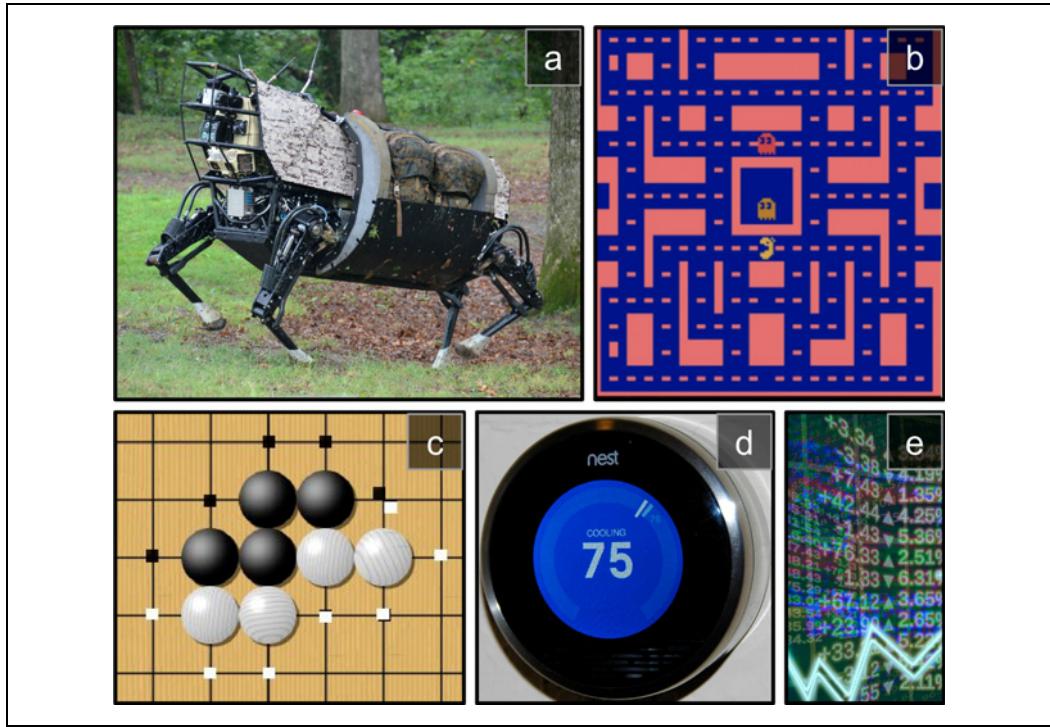
Uczenie się optymalizowania nagród

W uczeniu przez wzmacnianie oprogramowanie zwane **agentem** przeprowadza **obserwacje** i w obrębie **środowiska** wykonuje **czynności**, dzięki którym otrzymuje **nagrody**. Jego celem jest uczenie się zachowania maksymalizującego oczekiwane nagrody w dłuższym okresie. Jeżeli nie masz nic przeciwko odrobinie personifikacji, to dodatnie nagrody możemy interpretować jako przyjemność, a ujemne jako ból (w tym przypadku słowo „nagroda” jest nieco mylące). Krótko mówiąc, agent działa w środowisku i uczy się metodą prób i błędów maksymalizowania przyjemności i minimalizowania bólu.

Definicja ta jest całkiem obszerna i znajduje zastosowanie w wielu odmianach zadań. Oto kilka przykładów (rysunek 18.1):

- a. Agent może być programem sterującym chodem robota. Wówczas środowiskiem jest świat, agent obserwuje środowisko za pomocą szeregu **czujników**, takich jak kamery czy sensory dotyku, a czynnością jest przesyłanie sygnałów do części ruchowych. Możemy tak zaprogramować tego robota, żeby nagrodą dodatnią dla niego było dotarcie do celu, a ujemna była otrzymywana w sytuacji, gdy marnuje on czas lub idzie w złym kierunku.
- b. Agent może być programem sterującym Panią Pac-Manową (Ms. Pac-Man). W takiej sytuacji środowisko stanowi symulacja gry na komputerze Atari, czynności są definiowane przez dziesięć kierunków wychylenia joysticka (lewa góra, dolna, środkowa itd.), obserwacje przyjmują postać zrzutów ekranów, a nagrodą są punkty w grze.
- c. W drodze analogii agentem może być program grający w grę planszową, np. Go.
- d. Agent nie steruje fizycznie (lub wirtualnie) poruszającym się bytem. Możemy go znaleźć na przykład w termostacie inteligentnym, gdzie nagrodą dodatnią jest utrzymywanie docelowej temperatury przy jak najmniejszym nakładzie energii, natomiast nagrodę ujemną otrzymuje, gdy człowiek musi własnoręcznie wpływać na temperaturę, co sprawia, że agent musi nauczyć się przewidywać potrzeby człowieka.
- e. Agent może obserwować ceny akcji giełdowych i w każdej sekundzie decydować, ile z nich warto kupić, a ile sprzedać. Oczywiście nagrodami są zyski i straty.

Zwróci uwagę, że wcale nie muszą występować dodatnie nagrody — przykładowo agent może się poruszać po labiryncie i dostawać nagrodę ujemną za każdy wykonany ruch, dlatego stara się jak najszybciej z niego wydostać! Istnieje wiele innych zadań, do których uczenie przez wzmacnianie nadaje się idealnie, np. uczestnictwo samochodu autonomicznego w ruchu ulicznym, systemy rekomendacji, umieszczanie reklam na stronach internetowych lub kontrolowanie obszarów, na których powinien skoncentrować się system klasyfikowania obrazów.



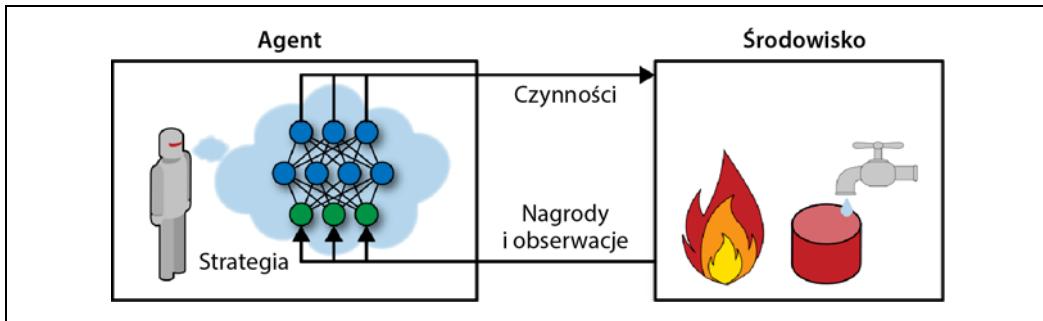
Rysunek 18.1. Przykłady zastosowań uczenia przez wzmacnianie: (a) robotyka, (b) Pani Pac-Manowa, (c) gracz w Go, (d) termostat, (e) gracz gieldowy⁵

Wyszukiwanie strategii

Algorytm używany przez agenta do określania czynności jest nazywany **strategią** (ang. *policy*). Przykładowo strategią może być sieć neuronowa przyjmująca na wejściach obserwacje i wypuszczająca na wyjściach rodzaj wykonywanej czynności (rysunek 18.2).

Strategią może być dowolny algorytm, nie musi być on nawet deterministyczny. W rzeczywistości w pewnych sytuacjach nie musi nawet obserwować środowiska! Dobrym przykładem jest inteligentny odkurzacz, dla którego nagrodą jest ilość kurzu zebranego w pół godziny. Jego strategię może stanowić ruch do przodu co sekundę z pewnym prawdopodobieństwem p albo losowe obracanie się w lewo lub prawo z prawdopodobieństwem $1-p$. Kąt obrotu byłby losowy i mieściłby się w zakresie od $-r$ do r .

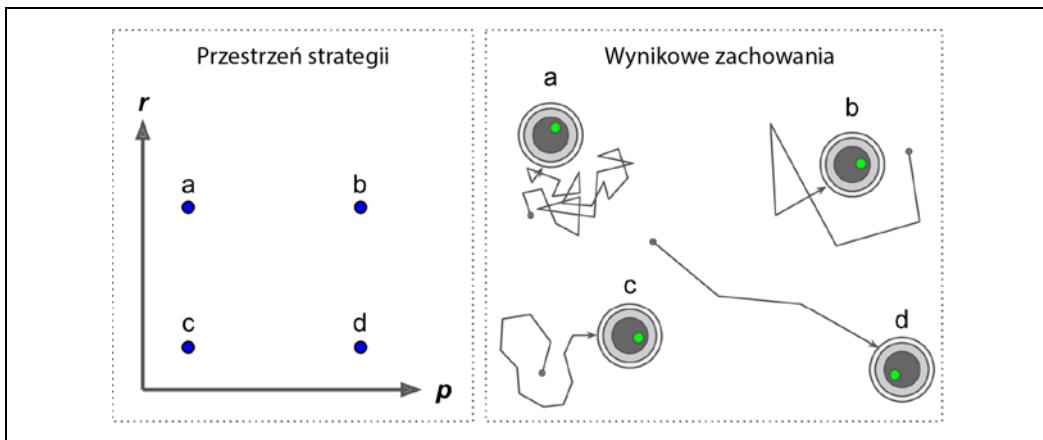
⁵ Rysunek (a) pochodzi z serwisu NASA (własność publiczna). Rysunek (b) stanowi zrzut ekranu z gry Ms. Pac-Man, której prawa autorskie należą do firmy Atari (uzasadnione użycie zrzutu ekranu w tym rozdziale). Obrazy (c) i (d) pochodzą z Wikipedii. Rysunek (c) został stworzony przez użytkownika Stevertigo i objęty licencją Uznanie autorstwa — na tych samych warunkach 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/deed.pl>). Rysunek (d) stanowi własność publiczną. Rysunek (e) pochodzi z serwisu Pixabay i jest objęty licencją Przekazanie do domeny publicznej CC0 (<https://creativecommons.org/publicdomain/zero/1.0/deed.pl>).



Rysunek 18.2. Uczenie przez wzmacnianie za pomocą strategii w postaci sieci neuronowej

Tego typu strategia wprowadza trochę losowości, dlatego jest nazywana **strategią stochastyczną** (ang. *stochastic policy*). Odkurzacz będzie poruszał się w chaotyczny sposób, dzięki czemu dotrze w każde dostępne miejsce i zbierze całą warstwę kurzu. Pytanie brzmi: jak wiele kurzu zdoła zebrać w pół godziny?

W jaki sposób można wytrenować taki odkurzacz? Możemy dostroić tylko dwa **parametry strategii** (ang. *policy parameters*): prawdopodobieństwo p i zakres kąta obrotu r . Jednym z dostępnych rozwiązań jest własnoręczne wypróbowanie wszystkich możliwych kombinacji tych parametrów i wybranie gwarantującej największą wydajność (rysunek 18.3). Jest to przykład **wyszukiwania strategii** (ang. *policy search*), w tym przypadku metodą „siłową”. Jednak w przypadku rozległej **przestrzeni strategii** (ang. *policy space*), a taka jest w większości przypadków, znalezienie optymalnej kombinacji parametrów można porównać do szukania igły w olbrzymim stogu siana.



Rysunek 18.3. Cztery punkty w przestrzeni strategii (po lewej) i odpowiadające im zachowanie agenta (po prawej)

Innym sposobem sprawdzania przestrzeni strategii jest zastosowanie **algorytmów genetycznych** (ang. *genetic algorithms*). Przykładowo możemy najpierw losowo stworzyć pierwsze pokolenie 100 strategii i wypróbować je, następnie „zniszczyć” 80 najmniej wydajnych strategii⁶, a z pozosta-

⁶ Często lepiej dać niezbyt wydajnym strategiom cień szansy na przetrwanie w celu zachowania zróżnicowanej „puli genetycznej”.

łych 20 pozwolić każdej wygenerować cztery strategie potomne. Taka strategia potomna stanowi kopię rodzica⁷ wraz z domieszką losowej wariancji. Tych 20 ocalałych strategii wraz z potomstwem stanowi drugie pokolenie. Możesz tak generować kolejne pokolenia strategii aż do znalezienia optymalnej kombinacji parametrów⁸.

Jeszcze innym rozwiązaniem jest użycie technik optymalizacyjnych poprzez ewaluację gradientów nagród w odniesieniu do parametrów strategii, a następnie zmodyfikowanie tych parametrów w sposób pozwalający na podążanie za gradientami w kierunku wyższych nagród⁹. Jest to tak zwana metoda **gradientów strategii** (ang. *policy gradients* — PG), którą omówię dokładniej w dalszej części rozdziału. Użyjmy jeszcze raz przykładu z odkurzaczem: moglibyśmy nieznacznie zwiększyć wartość p i sprawdzić, czy w konsekwencji robot zbierze więcej kurzu w ciągu 30 minut; jeśli tak, możemy spróbować jeszcze zwiększyć wartość p , w przeciwnym wypadku zmniejszyć ją. Zaimplementujemy w module TensorFlow popularny algorytm PG, zanim jednak do tego przejdziemy, musimy najpierw stworzyć środowisko, w którym będzie przebywał agent. W tym celu skorzystamy z narzędzia OpenAI Gym.

Wprowadzenie do narzędzia OpenAI Gym

Jednym z wyzwań uczenia przez wzmacnianie jest konieczność stworzenia środowiska roboczego w celu wyuczenia agenta. Jeśli chcesz zaprogramować agenta do przechodzenia gier stworzonych na komputer Atari, potrzebny byłby symulator tejże platformy. Aby zaprogramować chodzącego robota, środowiskiem okazuje się świat rzeczywisty i możemy bezpośrednio w nim wytrenować maszynę, ale rozwiązanie to ma swoje ograniczenia: jeśli robot spadnie z klifu, nie będziemy mogli tego cofnąć ani zresetować. Nie jesteśmy też w stanie przyspieszyć czasu; dodanie mocy obliczeniowej nie sprawi, że robot zacznie poruszać się szybciej. Ponadto jednokrotnie wytrenowanie tysiąca maszyn zazwyczaj jest również bardzo kosztowne. Krótko mówiąc, uczenie robotów w rzeczywistym otoczeniu jest zadaniem trudnym i powolnym, dlatego najczęściej potrzebujemy **środowiska symulowanego** (ang. *simulated environment*), przynajmniej w początkowej fazie trenowania. Możesz na przykład wykorzystać bibliotekę PyBullet (<https://pybullet.org/wordpress/>) lub MuJoCo (<http://www.mujoco.org/>), symulujące trójwymiarowe środowiska fizyczne.

Narzędzie OpenAI Gym (<https://gym.openai.com/>)¹⁰ zawiera różne rodzaje środowisk symulowanych (gry na komputer Atari, gry planszowe, dwu- i trójwymiarowe środowiska fizyczne itd.), dzięki czemu jesteśmy w stanie uczyć agenty, porównywać je lub projektować nowe algorytmy RL.

⁷ W przypadku obecności tylko jednego rodzica mamy do czynienia z **rozmnażaniem bezpłciowym** (ang. *asexual reproduction*). Jeżeli dostępne są przynajmniej dwie strategie nadrzędne, proces ten jest nazywany **rozmnażaniem płciowym** (ang. *sexual reproduction*). Genom potomka (w tym przypadku zestaw parametrów strategii) składa się z losowo dobranych części genomów poszczególnych rodziców.

⁸ Interesującym przykładem użycia algorytmu genetycznego w uczeniu przez wzmacnianie jest **algorytm NEAT** (ang. *NeuroEvolution of Augmented Topologies*; <https://www.cs.ucf.edu/~kstanley/neat.html>).

⁹ Jest to tak zwany algorytm **wzrostu gradientu** (ang. *gradient ascent*). Stanowi on przeciwieństwo algorytmu gradientu prostego — tutaj celem jest nie minimalizowanie funkcji, lecz znalezienie jej maksimum.

¹⁰ OpenAI jest firmą badawczą typu non profit zajmującą się opracowywaniem sztucznych inteligencji. Została założona przez Elona Muska. Jej zadaniem jest promowanie i projektowanie sztucznych inteligencji, które będą przyjazne ludzkości i będą ją wspierać (a nie dążyć do jej zniszczenia).

Zanim zainstalujemy to narzędzie, uaktywnijmy środowisko izolowane virtualenv (jeżeli je utworzyliśmy):

```
$ cd $ML_PATH          # Twój katalog roboczy uczenia maszynowego (np. $HOME/um)
$ source moje_srod/bin/activate # Linux/MacOS
$ .\moje_srod\Scripts\activate # Windows
```

Zainstalujmy narzędzie OpenAI Gym¹¹(jeżeli nie korzystasz ze środowiska wirtualnego, musisz dodać opcję --user lub skorzystać z uprawnień administratora):

```
$ python3 -m pip install -U gym
```

W zależności od systemu operacyjnego może być konieczne zainstalowanie biblioteki Mesa OpenGL Utility (GLU; na przykład w dystrybucji Ubuntu 18.04 należy wpisać polecenie apt install libglu1-mesa). Biblioteka ta posłuży do wyświetlenia pierwszego środowiska. Następnie otwórz powłokę Pythona lub notatnik Jupyter i stwórz środowiska za pomocą funkcji make():

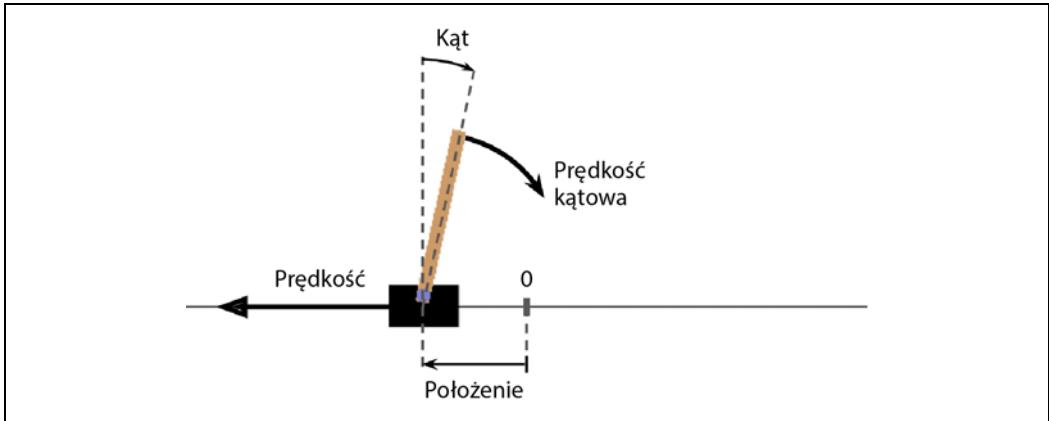
```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614, 0.04207708, -0.00180545])
```

Stworzyliśmy tu środowisko o nazwie CartPole. Jest to dwuwymiarowa symulacja, w której możemy przespieszać wózek w lewo lub w prawo w celu utrzymania w równowadze umieszczonego na nim masztu (rysunek 18.4). Listę wszystkich dostępnych środowisk uzyskasz za pomocą funkcji gym.envs.registry.all(). Po stworzeniu środowiska musimy je zainicjalizować za pomocą metody reset(). Zostaje w ten sposób zwrocona pierwsza obserwacja. Rodzaj obserwacji zależy od typu środowiska. W przypadku środowiska CartPole każda obserwacja stanowi jednowymiarową tablicę NumPy zawierającą cztery wartości zmiennoprzecinkowe: położenie wózka w osi poziomej (0,0 = środek), jego prędkość (wartość dodatnia oznacza kierunek w prawo), kąt nachylenia masztu (0,0 = pionowo) oraz jego prędkość kątową (wartość dodatnia oznacza ruch zgodny z kierunkiem wskazówek zegara).



Jeżeli korzystasz z serwera bezobsługowego (tzn. niezawierającego monitora), np. z wirtualnej maszyny umieszczonej w chmurze, to proces wyświetlania obrazu zostanie zakończony niepowodzeniem. Jednym sposobem uniknięcia tego problemu jest użycie fałszywego serwera X, takiego jak Xvfb czy Xdummy. Możesz na przykład zainstalować serwer Xvfb (polecenie apt install xvfb w dystrybucji Ubuntu lub Debian) i uruchomić środowisko Pythona za pomocą polecenia xvfb-run -s -screen 0 1400x900x24" python3. Ewentualnie możesz zainstalować serwer Xvfb i bibliotekę pyvirtualdisplay (<https://github.com/pony/pyvirtualdisplay>), która „opakowuje” serwer Xvfb, a na początku programu umieścić wiersz pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start().

¹¹ Obecnie pakiet OpenAI Gym jest dostępny wyłącznie na dystrybucje Linuksa i system Mac OS. Istnieje jednak sposób jego instalacji w systemie Windows, instrukcję (w języku angielskim) znajdziesz na stronie <https://towardsdatascience.com/how-to-install-openai-gym-in-a-windows-environment-338969e24d30> — przyp. tłum.



Rysunek 18.4. Środowisko CartPole

Wyświetlmy teraz to środowisko za pomocą metody render() (rysunek 18.4). W systemie Windows wymagane jest do tego zainstalowanie środowiska X Server, np. VcXsrv lub Xming:

```
>>> env.render()
True
```

Jeśli chcesz, aby metoda render() zwracała obraz środowiska w postaci tablicy NumPy, możesz wyznaczyć mode="rgb_array" (co ciekawe, środowisko to mimo wszystko będzie wyświetlane na ekranie):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # Wysokość, szerokość, kanały (3=czerwony, zielony, niebieski)
(800, 1200, 3)
```

Dowiedzmy się, jakie mamy dostępne czynności:

```
>>> env.action_space
Discrete(2)
```

Wynik Discrete(2) oznacza, że dostępnymi czynnościami są wartości 0 i 1, odpowiadające przyspieszeniu w lewo (0) lub w prawo (1). Inne środowiska mogą umożliwiać większą liczbę dyskretnych czynności lub zupełnie inne ich rodzaje (np. ciągłe czynności). Maszt przechyla się prawo ($obs[2] > 0$), dlatego przyspieszmy ruch wózka w prawą stronę:

```
>>> action = 1 # Przyspieszamy w prawo
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699, 0.19292789, 0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

Metoda `step()` wykonuje określoną czynność i zwraca cztery wartości:

obs

Jest to nowa obserwacja. Wózek porusza się teraz w prawo ($obs[1] > 0$). Maszt jest ciągle przechylony w prawo ($obs[2] > 0$), ale jego prędkość kątowa ma teraz ujemną wartość ($obs[3] < 0$), zatem po następnym kroku będzie prawdopodobnie przechylony w lewo.

reward

W tym środowisku agent będzie otrzymywał nagrodę o wartości 1,0 w każdym etapie bez względu na to, co będzie się działało, zatem celem jest jak najdłuższe utrzymywanie wózka w ruchu.

done

Atrybut ten będzie miał wartość `True`, gdy dany epizod zostanie zakończony. Sytuacja taka wystąpi, jeśli maszt przechyli się za bardzo, wysunie się poza obszar ekranu lub po 200 krokach (ostatnia sytuacja oznacza wygraną). W takim przypadku należy zresetować środowisko, żeby można było z niego ponownie skorzystać.

info

Ten specyficzny dla każdego środowiska słownik może dostarczać dodatkowe informacje pomocne w usuwaniu błędów lub w uczeniu. W pewnych grach może na przykład wyznaczać liczbę życia agenta.



Gdy skończysz korzystać ze środowiska, wywołaj metodę `close()`, aby zwolnić zasoby.

Napiszmy kod prostej strategii przyspieszającej wózek w lewo, gdy maszt zaczyna przechylać się w lewo (to samo dla prawej strony). Uruchomimy tę strategię i sprawdzimy, jaką średnią nagrodę będzie otrzymywać po 500 epizodach:

```
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(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

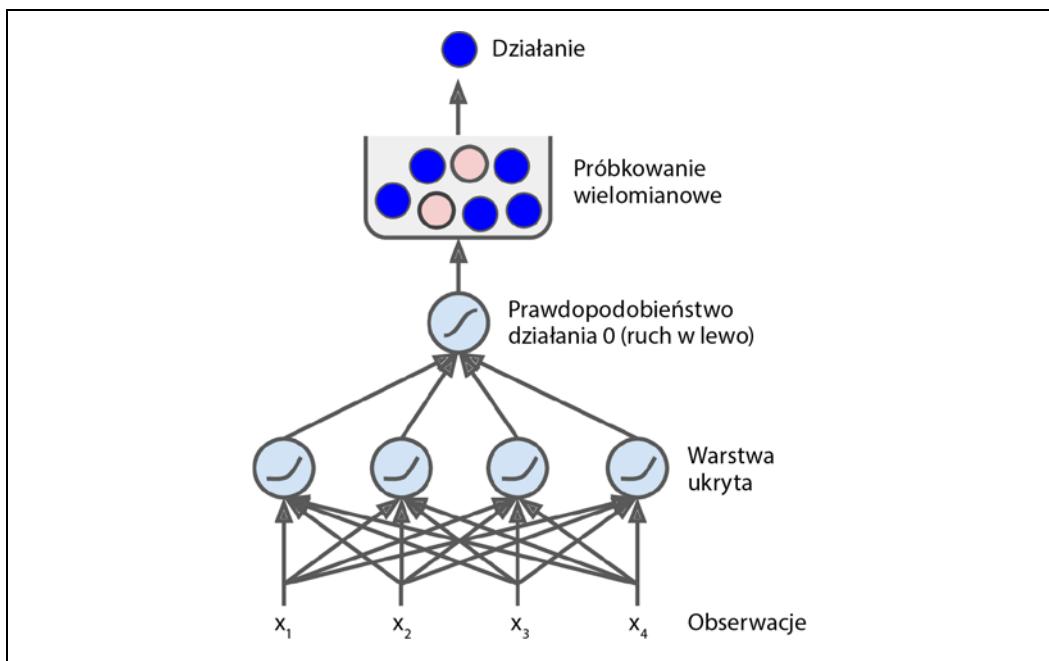
Mam nadzieję, że powyższy kod nie wymaga wyjaśnień. Spójrzmy na wyniki:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Nawet mając do dyspozycji 500 epizodów, strategia nie była w stanie ani razu utrzymać masztu pionowo dłużej niż przez 68 kroków. Nie za dobrze. Jeśli przyjrzyisz się symulacji umieszczonej w notatniku Jupyter, zauważysz, że wagon oscyluje coraz mocniej w lewo i w prawo aż do momentu nadmiernego przechylenia masztu. Sprawdźmy, czy sieć neuronowa będzie w stanie zdefiniować lepszą strategię.

Sieci neuronowe jako strategie

Stwórzmy teraz strategię będącą w istocie siecią neuronową. Podobnie jak w przypadku poprzedniego przykładu, ta sieć neuronowa będzie na wejściu przyjmowała obserwacje, a na wyjściu będzie wstawać działanie, jakie ma zostać wykonane. Mówiąc precyzyjniej, sieć ta będzie szacować prawdopodobieństwo wystąpienia każdego działania, a następnie losowo dobierze działanie na podstawie oszacowanych prawdopodobieństw (rysunek 18.5). W przypadku środowiska CartPole istnieją tylko dwie dostępne czynności (ruch w lewo lub w prawo), dlatego będziemy potrzebować tylko jednego neuronu wyjściowego. Będzie on umieszczony na wyjściu prawdopodobieństwo p czynności 0 (ruch w lewo), jest więc oczywiste, że prawdopodobieństwo $1-p$ będzie symbolizować działanie 1 (ruch w prawo). Przykładowo jeśli wynikiem neuronu wyjściowego będzie wartość 0,7, znaczy to, że istnieje 70% szans na wybór działania 0 i 30% dla czynności 1.



Rysunek 18.5. Sieć neuronowa jako strategia

Pewnie się zastanawiasz, dlaczego wybieramy losowe działanie, bazując na prawdopodobieństwie wyliczonym przez sieć neuronową, zamiast po prostu dobrać działanie, które uzyskało najwyższy wynik. Dzięki takiemu podejściu agent jest w stanie zachować właściwą równowagę pomiędzy **poszukiwaniem** nowych działań a **wykorzystywaniem** już znanych i sprawdzonych. Posłużymy się analogią:

założmy, że idziesz pierwszy raz w życiu do restauracji i wszystkie dania wyglądają równie apetycznie, dlatego zamawiasz losowo jedno z nich. Jeśli okaże się smaczne, to zwiększa się prawdopodobieństwo, że wybierzesz je znowu, ale wartość tego prawdopodobieństwa nigdy nie osiąga 100%, gdyż w przeciwnym razie nigdy nie zaryzykowałbyś zamówienia innej potrawy, która mogłaby okazać się jeszcze smaczniejsza.

Zwróć również uwagę, że w tym konkretnym środowisku możemy spokojnie ignorować wcześniejsze działania i obserwacje, ponieważ każda obserwacja zawiera pełne informacje na temat stanu środowiska. Gdyby występował tu jakiś ukryty stan, musielibyśmy brać pod uwagę także poprzednie działania i obserwacje. Na przykład jeśli środowisko przekazało informację jedynie na temat położenia wózka, ale nie jego prędkości, musielibyśmy uwzględnić też poprzednią obserwację w celu ustalenia prędkości. Spostrzeżenie to dotyczy również zaszumionych obserwacji — w tej sytuacji zazwyczaj należy wykorzystać kilka wcześniejszych obserwacji w celu oszacowania najbardziej prawdopodobnego stanu środowiska. Z tego wynika, że problem określony w środowisku CartPole jest najprostszy z możliwych; obserwacje nie zawierają szumu, za to przechowują pełny stan środowiska.

Oto kod definiujący sieć neuronową jako strategię w interfejsie tf.keras:

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])

```

Po zimportowaniu potrzebnych klas definiujemy sieć neuronową jako strategię za pomocą standartowego modelu Sequential. Liczba wejść jest równa rozmiarowi przestrzeni obserwacji (w przypadku środowiska CartPole ma ona wartość 4) i wprowadzamy jedynie pięć jednostek ukrytych, gdyż mamy do czynienia z nieskomplikowanym problemem. Chcemy także, aby na wyjściu było umieszczone pojedyncze prawdopodobieństwo (prawdopodobieństwo wykonania ruchu w lewo), dlatego umieszczać w warstwie wyjściowej jeden neuron, zawierający sigmoidalną funkcję aktywacji. Gdybyśmy mieli do czynienia z ponad dwiema możliwościami, to na każdą z nich przydzieliłyby osobny neuron, a funkcję sigmoidalną zastąpilibyśmy funkcją softmax.

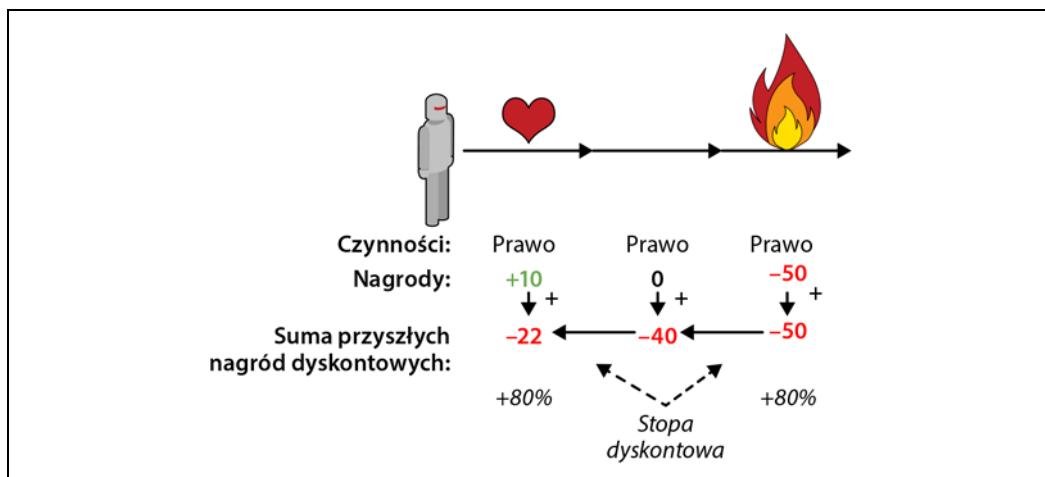
No dobrze, przygotowaliśmy strategię w postaci sieci neuronowej, która będzie pobierała obserwacje i wybierała czynności. Jak jednak wygląda proces jej uczenia?

Ocenianie czynności: problem przypisania zasługi

Gdybyśmy wiedzieli, jaka czynność będzie najlepsza w danym kroku, moglibyśmy wytrenować sieć neuronową w standardowy sposób — minimalizując entropię krzyżową pomiędzy rozkładami szacowanego i docelowego prawdopodobieństwa. Mielibyśmy do czynienia z tradycyjnym uczeniem nadzorowanym. Jednak w przypadku uczenia przez wzmacnianie jedyne wskazówki, na jakich może bazować agent, kryją się w nagrodach, które są zazwyczaj rzadkie i opóźnione. Przykładowo jeśli agentowi uda się utrzymać maszt pionowo przez sto etapów, skąd może wiedzieć, które z czynności

podjętych w tym czasie były dobre, a które złe? Wie jedynie, że maszt przewrócił się po ostatniej czynności, jednak nie można zwalać na nią całą winy. Jest to tak zwany **problem przypisania zasługi** (ang. *credit assignment problem*): gdy agent otrzymuje nagrodę, nie jest w stanie stwierdzić, którym czynnościom należy przypisać zasługę (lub winę). Podobnie jest w przypadku psa, który otrzymuje nagrodę kilka godzin po czynności zasługującej na pochwałę: czy będzie rozumiał, za co został nagrodzony?

Często stosowaną strategią pozwalającą na rozwiązywanie tego problemu jest ocenienie czynności na podstawie sumy wszystkich nagród otrzymanych po jej wykonaniu, najczęściej przy zastosowaniu **stopy dyskontowej γ** (ang. *discount factor*) na każdym etapie. Suma takich dyskontowych nagród stanowi **zwrot** (ang. *return*) czynności. Przykład został pokazany na rysunku 18.6: jeżeli agent postanowi trzykrotnie wykonać ruch w prawo i otrzyma nagrodę +10 za pierwszy krok, 0 za drugi i -50 za trzeci, to przy założeniu, że wartość stopy dyskontowej wynosi $\gamma = 0,8$, okazuje się, że pierwsze działanie będzie miało całkowitą wartość $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. Jeżeli stopa dyskontowa będzie bliska零, to przyszłe nagrody nie będą liczyć się znacząco w porównaniu do bieżących nagród. I odwrotnie: w przypadku wartości stopy dyskontowej zbliżonej do 1 przyszłe nagrody będą miały niemal takie samo znaczenie jak nagrody bieżące. Typowe wartości tego parametru mieścią się w zakresie od 0,9 do 0,99. Przy stopie dyskontowej o wartości 0,95 nagrody czekające po 13 krokach będą miały wagę niemal równą połowie wagi bieżącej nagrody ($0,95^{13} \approx 0,5$), natomiast w przypadku stopy o wartości 0,99 podobny wynik osiągniemy po 69 krokach. W środowisku CartPole czynności mają w miarę krótkotrwały wpływ, zatem rozsądny jest wybór stopy dyskontowej o wartości 0,95.



Rysunek 18.6. Obliczanie zwrotu czynności: suma przyszłych nagród dyskontowych

Oczywiście po dobrym działaniu może nastąpić kilka złych działań, powodujących przewrócenie masztu, przez co dobre działanie uzyskuje niski zwrot (nawet świetny aktor może zagrać od czasu do czasu w kiepskim filmie). Jeśli jednak zagramy w grę wystarczająco wiele razy, po uśrednieniu dobre czynności będą miały lepszy zwrot od złych czynności. Chcemy oszacować, w jakim stopniu dana czynność jest lepsza lub gorsza (po uśrednieniu) od pozostałych czynności. Jest to tak zwana **przewaga czynności** (ang. *action advantage*). Aby ją obliczyć, musimy pozwolić działać agentowi przez wiele epizodów i znormalizować zwroty czynności (poprzez odjęcie od nich średniej i podzielenie

ich przez odchylenie standardowe). Następnie możemy rozsądnie założyć, że czynności mające zwroty ujemne są złe, w przeciwieństwie do czynności mających zwroty dodatnie. Znakomicie — znamy już sposób oceniania każdej czynności i jesteśmy gotowi wyuczyć pierwszego agenta za pomocą gradientów strategii. Zobaczmy, na czym to polega.

Gradienty strategii

Jak już wiesz, algorytmy PG optymalizują parametry strategii, dążąc do wyższych nagród wzduż gradientów. W 1992 roku Ronald Williams zaproponował popularną klasę algorytmów PG — **algorytmy REINFORCE** (<https://homl.info/132>)¹². Oto jeden z bardziej znanych wariantów:

1. Najpierw pozwalamy strategii w postaci sieci neuronowej na kilkukrotne zagranie w grę i w każdym kroku obliczamy gradienty, dzięki którym dana czynność byłaby wykonana z jeszcze większym prawdopodobieństwem, jednak jeszcze ich nie stosujemy.
2. Po kilku epizodach obliczamy wynik każdej czynności (za pomocą metody opisanej w poprzednim podrozdziale).
3. Jeżeli wynik czynności jest dodatni, oznacza to, że jest ona dobra, i chcemy zastosować wcześniej wyliczone gradienty po to, aby zwiększyć prawdopodobieństwo wybrania jej w przyszłości. Jeśli jednak wynik danej czynności jest ujemny, jest ona zła, i chcemy wprowadzić przeciwnie gradienty, żeby nieznacznie zmniejszyć prawdopodobieństwo jej wyboru w przyszłości. Sama operacja polega na przemnożeniu każdego wektora gradientów przez przewagę danej czynności.
4. Na koniec obliczamy średnią uzyskanych wektorów gradientów i używamy ich w algorytmie gradientu prostego.

Do zaimplementowania tego algorytmu wykorzystajmy interfejs tf.keras. Wytrenujemy utworzoną chwilę wcześniej sieć neuronową tak, że nauczy się utrzymywać maszt pionowo na wózku. Potrzebna jest nam najpierw funkcja wykonująca jeden krok uczenia. Będziemy na razie udawać, że każda wykonana czynność jest właściwa, dzięki czemu będziemy w stanie obliczyć funkcję straty i gradienty (gradienty zostaną zapisane tylko na pewien czas; będziemy je później modyfikować w zależności od tego, jak dobra okazała się dana czynność):

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```

Przeanalizujmy tę funkcję:

- W bloku GradientTape (zob. rozdział 12.) najpierw wywołujemy model i przekazujemy mu jedną obserwację (tak przekształcamy obserwację, że staje się grupą zawierającą jeden przykład,

¹² Ronald J. Williams, *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning*, „Machine Learning” 8 (1992), s. 229 – 256.

ponieważ model oczekuje grupy na wejściu). W ten sposób uzyskamy prawdopodobieństwo wykonania ruchu w lewo.

- Następnie losujemy wartość zmiennoprzecinkową w zakresie od 0 do 1 i sprawdzamy, czy jest ona większa od `left_proba`. Czynność `action` będzie miała wartość `False` z prawdopodobieństwem `left_proba` lub wartość `True` z prawdopodobieństwem `1 - left_proba`. Po przekształceniu wartości logicznej w wartość liczbową otrzymamy czynności wykonania ruchu w lewo (0) lub w prawo (1) z wyznaczonymi odpowiednimi prawdopodobieństwami.
- Teraz definiujemy prawdopodobieństwo docelowe ruchu w lewo: wynosi ono 1 minus czynność (przekształcona w liczbę zmiennoprzecinkową). Jeżeli wartość czynności wynosi 0 (ruch w lewo), to prawdopodobieństwo docelowe wyniesie 1. Z kolei jeżeli wartość czynności jest równa 1 (ruch w prawo), to prawdopodobieństwo docelowe będzie miało wartość 0.
- Obliczamy funkcję straty, a następnie za pomocą taśmy wyznaczamy gradient tej funkcji w odniesieniu do zmiennych modyfikowalnych modelu. Pamiętaj, że przed ich zastosowaniem gradienty te będą później precyzowane, w zależności od tego, czy dana czynność okazała się dobra czy zła.
- Na koniec odgrywamy daną czynność i zwracamy nową obserwację, nagrodę, informację o tym, czy dany epizod został zakończony, oraz oczywiście obliczone gradienty.

Utwórzmy teraz kolejną funkcję, opartą na funkcji `play_one_step()` i odtwarzającą wiele epizodów; będzie ona zwracała wszystkie wartości nagród i gradienty dla każdego epizodu i przebiegu:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):  
    all_rewards = []  
    all_grads = []  
    for episode in range(n_episodes):  
        current_rewards = []  
        current_grads = []  
        obs = env.reset()  
        for step in range(n_max_steps):  
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)  
            current_rewards.append(reward)  
            current_grads.append(grads)  
            if done:  
                break  
        all_rewards.append(current_rewards)  
        all_grads.append(current_grads)  
    return all_rewards, all_grads
```

Kod ten zwraca listę z listami nagród (po jednej liście na epizod, po jednej nagrodzie na każdy krok), a także listę z listami gradientów (po jednej liście na epizod, każda z nich zawiera po jednej krotce gradientów na każdy krok, a każda krotka zawiera po jednym tensorze gradientów na każdą zmienną modyfikowaną).

Algorytm będzie wykorzystywał funkcję `play_multiple_episodes()` do rozegrania kilku partii (np. dziesięciu), a następnie przeanalizuje uzyskane nagrody, zdyskontuje ich wartość i znormalizuje ją. Do tego będą nam potrzebne dwie dodatkowe funkcje: pierwsza będzie obliczać sumę przyszłych nagród dyskontowych w każdym kroku, druga zaś będzie normalizować wszystkie te nagrody dyskontowe (zwroty) w wielu epizodach poprzez odjęcie średniej i podzielenie przez odchylenie standardowe:

```

def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              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]

```

Zobaczmy, czy działa to rozwiązanie:

```

>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.07277777])]

```

Wywołanie funkcji `discount_rewards()` zwraca dokładnie to, czego się spodziewamy (rysunek 18.6). Możemy sprawdzić, że funkcja `discount_and_normalize_rewards()` oczywiście zwraca znormalizowane wyniki dla każdej czynności w obydwu epizodach. Zwróć uwagę, że w pierwszym epizodzie model uzyskuje znacznie gorsze rezultaty niż w drugim, dlatego obliczone w nim wszystkie znormalizowane przewagi są ujemne; wszystkie czynności z pierwszego epizodu zostały uznane za złe, z kolei czynności z drugiego epizodu byłyby określone jako dobre.

Jesteśmy już niemal gotowi do uruchomienia algorytmu! Zdefiniujmy teraz hiperparametry. Przeprowadzimy 150 iteracji uczących i na każdą iterację poświęcimy 10 epizodów, a każdy epizodów będzie się składał maksymalnie z 200 kroków. Wyznaczmy stopę dyskontową o wartości 0,95:

```

n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95

```

Potrzebne są nam również optymalizator i funkcja straty. Powinien nam całkowicie wystarczyć standardowy optymalizator Adam o współczynniku uczenia 0,01, a jako funkcję straty wyznaczmy binarną entropię krzyżową, ponieważ trenujemy klasyfikator binarny (występują dwie możliwe czynności: ruch w lewo albo w prawo):

```

optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy

```

Utwórzmy i zrealizujmy w końcu pętlę uczenia!

```

for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []

```

```

for var_index in range(len(model.trainable_variables)):
    mean_grads = tf.reduce_mean(
        [final_reward * all_grads[episode_index][step][var_index]
         for episode_index, final_rewards in enumerate(all_final_rewards)
             for step, final_reward in enumerate(final_rewards)], axis=0)
    all_mean_grads.append(mean_grads)
optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))

```

Przeanalizujmy powyższy listing:

- W każdej iteracji pętla uczenia wywołuje funkcję `play_multiple_episodes()`, która rozgrywa 10 partii i zwraca wszystkie nagrody oraz gradienty z każdego epizodu i kroku.
- Następnie wywołujemy funkcję `discount_and_normalize_rewards()` po to, aby obliczyć znormalizowaną przewagę każdej czynności (w listingu jest ona nazwana `final_reward`). Dzięki temu dowiadujemy się z perspektywy czasu, czy poszczególne czynności były dobre czy złe.
- Przechodzimy następnie do każdej zmiennej modyfikowej i obliczamy dla każdej z nich średnią ważoną gradientów ze wszystkich epizodów i kroków (jest ona ważona względem `final_reward`).
- Na koniec wprowadzamy te uśrednione gradienty za pomocą optymalizatora: zostaną zaktualizowane zmienne modyfikowe modelu, co, miejmy nadzieję, udoskonali nieco strategię.

I zrobione! Powyższy kod wytrenuje strategię w postaci sieci neuronowej i nauczy się skutecznie utrzymywać maszt w równowadze (działający przykład znajdziesz w sekcji „Gradienty strategii” w notatniku Jupyter). Średnia nagroda na każdy epizod będzie zbliżona do wartości 200 (domyślnie jest to maksymalna nagroda w tym środowisku). Sukces!



Naukowcy próbują opracować algorytm działający wydajnie nawet w sytuacji, gdy agent początkowo nic nie wie o środowisku. W praktyce jednak, jeżeli nie piszesz publikacji naukowej, dostarczaj agentowi jak najwięcej informacji, ponieważ taka wiedza znacznie przyspiesza proces uczenia. Wiemy na przykład, że maszt powinien być ustawiony maksymalnie pionowo, dlatego moglibyśmy dodać ujemne nagrody proporcjonalne do jego przechylenia. Nagrody przestaną być takie rzadkie, a proces uczenia zostanie przyspieszony. Poza tym w przypadku korzystania z wystarczająco dobrej strategii (np. własnoręcznie napisanej) możesz chcieć wyuczyć imitującą ją sieć neuronową, zanim wykorzystasz gradienty strategii do jej usprawnienia.

Wyuczony przez nas prosty algorytm gradientów strategii wykonał swoje zadanie w środowisku CartPole, ale niezbyt nadaje się do większych i bardziej skomplikowanych wyzwań. To dlatego, że wykazuje bardzo **małą wydolność próbkowania** (ang. *sample inefficiency*), co oznacza, że musi poświęcić bardzo dużo czasu na poznawanie gry, zanim zostaną poczynione znaczne postępy. Wynika to z faktu, że w celu oszacowania przewagi każdej czynności należy rozegrać wiele epizodów, o czym już zdążyliśmy się przekonać. Rozwiążanie to jednak stanowi podwaliny pod znacznie późniejsze algorytmy, takie jak algorytm **aktor – krytyk** (przyjrzymy mu się pobiieżnie pod koniec rozdziału).

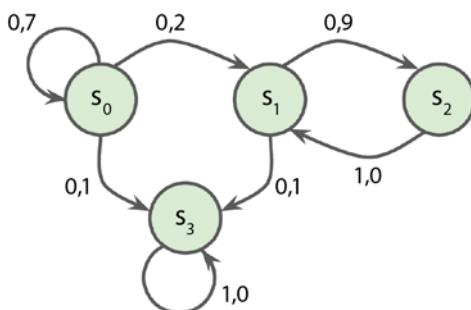
Przyjrzymy się teraz innej popularnej rodzinie algorytmów. Tam, gdzie algorytmy PG bezpośrednio próbują optymalizować strategię w celu zwiększenia nagród, omawiana klasa algorytmów działa w bardziej niejawnny sposób: agent uczy się szacować oczekiwanyą sumę zmniejszonych przyszłych nagród dla każdego stanu lub oczekiwanyą sumę zmniejszonych przyszłych nagród dla każdej czynności w każdym stanie, po czym wykorzystuje tę wiedzę do wybrania czynności. Aby zrozumieć

mechanizm działania tych algorytmów, musimy najpierw omówić **procesy decyzyjne Markowa** (ang. *Markov Decision Processes* — MDP).

Procesy decyzyjne Markowa

Na początku XX wieku matematyk Andriej Markow badał procesy stochastyczne niezawierające pamięci, tzw. **łańcuchy Markowa**. Taki proces ma stałą liczbę stanów i w każdym kroku losowo przechodzi z jednego stanu do innego. Prawdopodobieństwo przejścia procesu ze stanu s do stanu s' jest niezmienne i zależy wyłącznie od pary (s, s') , a nie od przeszłych stanów (dlatego mówimy, że system nie ma pamięci).

Rysunek 18.7 przedstawia przykład łańcucha Markowa zawierającego cztery stany.



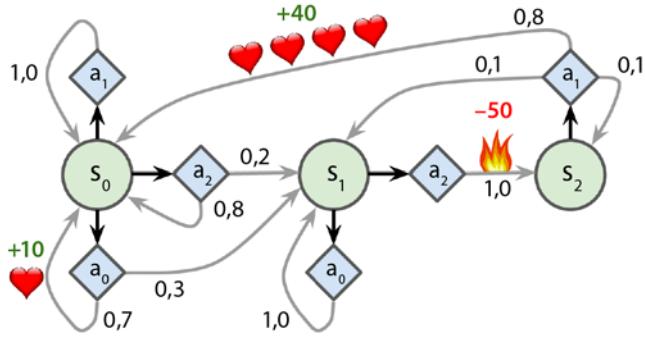
Rysunek 18.7. Przykład łańcucha Markowa

Załóżmy, że proces zaczyna się w stanie s_0 i prawdopodobieństwo jego pozostania w tym stanie w następnym kroku wynosi 70%. W końcu jednak proces ten opuści stan s_0 i nigdy już do niego nie wróci, ponieważ żaden stan nie wskazuje na niego. Jeśli proces przejdzie do stanu s_1 , to stamtąd najprawdopodobniej przejdzie do stanu s_2 (prawdopodobieństwo wynosi 90%), a następnie natychmiast wróci do stanu s_1 (100% prawdopodobieństwa). Proces może oscylować jakiś czas pomiędzy tymi dwoma stanami, ale w końcu przejdzie do stanu s_3 i pozostałe w nim już na zawsze (jest to **stan końcowy** — ang. *terminal state*). łańcuchy Markowa cechują się zróżnicowaną dynamiką i są często wykorzystywane w termodynamice, chemii, statystyce itd.

Procesy decyzyjne Markowa zostały po raz pierwszy opisane w latach 50. ubiegłego wieku przez Richarda Bellmana (<https://homl.info/133>)¹³. Przypominają one łańcuchy Markowa, istnieje jednak pewna różnica: w każdym kroku agent może wybrać jedną z kilku dostępnych czynności, od których zależy prawdopodobieństwo przejścia do określonego stanu. Ponadto przejścia do pewnych stanów zwracają jakąś nagrodę (dodatnią lub ujemną), a zadaniem agenta jest znalezienie strategii maksymalizującej uzyskiwanie nagród wraz z upływem czasu.

Na przykład proces MDP zaprezentowany na rysunku 18.8 ma trzy stany (kółka) oraz maksymalnie trzy możliwe dyskretnie czynności w każdym kroku (romby).

¹³ Richard Bellman, *A Markovian Decision Process*, „Journal of Mathematics and Mechanics” 6, no. 5 (1957), s. 679 – 684.



Rysunek 18.8. Przykładowy proces decyzyjny Markowa

Jeżeli proces rozpoczyna się od stanu s_0 , agent może wybrać jedną z czynności: a_0 , a_1 lub a_2 . W przypadku czynności a_1 proces pozostaje w stanie s_0 z całkowitą pewnością i bez jakiejkolwiek nagrody, może więc pozostać tam na zawsze, jeśli agent tak zadecyduje. Jeśli jednak zostanie wybrana czynność a_0 , istnieje 70% prawdopodobieństwa, że proces otrzyma nagrodę +10 i pozostanie w stanie s_0 . Agent może zatem ciągle próbować zdobywać jak największą nagrodę. Jednak w pewnym momencie wyląduje w końcu w stanie s_1 . Tutaj dostępne są tylko dwie czynności: a_0 lub a_2 . Agent może tu zostać, wybierając ciągle czynność a_0 , lub przejść do stanu s_2 i otrzymać ujemną nagrodę -50 (auć). W tym stanie może wybrać jedynie czynność a_1 , która najprawdopodobniej przeniesie proces z powrotem do stanu s_0 , przy czym zberzie po drodze nagrodę +40. Wiesz już, jak to wygląda. Czy analizując proces MDP, jesteś w stanie przewidzieć, jaka strategia pozwala uzyskać największą nagrodę wraz z upływem czasu? Jest oczywiste, że w stanie s_0 najlepszym rozwiązaniem okazuje się czynność a_0 , a w stanie s_2 agent może wybrać wyłącznie czynność a_1 , jednak w stanie s_1 nie jest wcale takie oczywiste, czy agent powinien pozostać w miejscu (a_0), czy przejść przez płomienie (a_2).

Bellman odkrył sposób oszacowania **optymalnej wartości stanu** s , określonej wyrażeniem $V^*(s)$, stanowiącej uśrednioną sumę wszystkich zmniejszonych przyszłych nagród, jakiej agent może się spodziewać po osiągnięciu stanu s , przy założeniu, że wykonuje optymalne czynności. Udowodnił, że jeśli agent zachowuje się optymalnie, to można zastosować **równanie optymalności Bellmana** (równanie 18.1). Ten rekurencyjny wzór mówi nam, że jeśli agent zachowuje się optymalnie, to optymalna wartość bieżącego stanu jest równa sumie średniej nagrody po wybraniu optymalnej czynności i oczekiwanej optymalnej nagrody dla wszystkich możliwych stanów, do których ta czynność może prowadzić.

Równanie 18.1. Równanie optymalności Bellmana

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad \text{dla wszystkich } s$$

W tym równaniu:

- $T(s, a, s')$ jest prawdopodobieństwem przejścia ze stanu s do stanu s' przy założeniu, że agent wybrał czynność a . Na przykład na rysunku 18.8 $T(s_2, a_1, s_0) = 0,8$.
- $R(s, a, s')$ to nagroda, jaką agent otrzyma po przejściu ze stanu s do stanu s' przy założeniu, że wybrał czynność a . Na przykład na rysunku 18.8 $R(s_2, a_1, s_0) = +40$.
- γ wyznacza stopę dyskontową.

Równanie to prowadzi bezpośrednio do algorytmu mogącego precyjnie oszacować optymalną wartość stanu dla każdego możliwego stanu: najpierw inicjalizujemy oszacowania wartości stanu z wartością zero, a następnie iteracyjnie je aktualizujemy za pomocą **algorytmu iteracji wartości** (ang. *value iteration*) (równanie 18.2). Godny uwagi jest fakt, że po upływie czasu oszacowania te uzyskają zbieżność z optymalnymi wartościami stanu, równoznacznymi z optymalną strategią.

Równanie 18.2. Algorytm iteracji wartości

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{dla wszystkich } s$$

W tym równaniu $V_k(s)$ jest szacowaną wartością stanu s w k -tym przebiegu algorytmu.



Algorytm ten stanowi przykład **programowania dynamicznego**, dzięki któremu złożony problem zostaje przekształcony w rozwiązywalne problemy składowe, które mogą być rozwiązywane iteracyjnie.

Znajomość optymalnych wartości stanu może być przydatna, zwłaszcza do oceniania strategii, ale nie mówią one bezpośrednio agentowi, co powinien robić. Na szczęście Bellman odkrył bardzo podobny algorytm oszacowujący optymalne **wartości stanu – czynności** (ang. *state-action values*), znane jako **Q-wartości** (ang. *Q-Values*; skrót Q pochodzi od angielskiego słowa *quality*, czyli „jakość”). Optymalną Q-wartością pary stan – czynność (s, a) (zapisywana jako $Q^*(s, a)$) jest średnia suma zmniejszonych przyszłych nagród agenta, jakiej może się spodziewać po osiągnięciu stanu s i wybraniu czynności a , jednak przed wybraniem danej czynności widzi jej konsekwencje, przy założeniu, że będzie działać optymalnie po jej wybraniu.

Mechanizm działania tego algorytmu jest następujący: również w tym przypadku inicjalizujemy oszacowania Q-wartości z wartością 0, a następnie aktualizujemy je za pomocą algorytmu **iteracji Q-wartości** (ang. *Q-Value iteration*) (równanie 18.3).

Równanie 18.3. Algorytm iteracji Q-wartości

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a')] \quad \text{dla wszystkich } (s, a)$$

Po otrzymaniu optymalnych Q-wartości zdefiniowanie optymalnej strategii (określonej jako $\pi^*(s)$) okazuje się banalne: gdy agent znajduje się w stanie s , powinien wybrać czynność mającą największą Q-wartość dla tego stanu: $\pi^*(s) = \arg \max_a Q^*(s, a)$.

Wstawmy ten algorytm do procesu MDP zaprezentowanego na rysunku 18.8. Najpierw musimy zdefiniować ten proces:

```
transition_probabilities = [ # Wymiary=[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], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [ # Wymiary=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Jeżeli na przykład chcemy poznać prawdopodobieństwo przejścia ze stanu s_2 do stanu s_0 po wykonaniu czynności a_1 , zwróćmy uwagę na `transition_probabilities[2][1][0]` (wartość wynosi 0,8). Jeśli natomiast interesuje nas związana z tą czynnością nagroda, poszukamy `rewards[2][1][0]` (wartość wynosi +40). A jeżeli chcemy sprawdzić listę dostępnych czynności w stanie s_2 , znajdziemy je w `possible_actions[2]` (w tym przypadku występuje tu tylko czynność a_1). Następnie musimy zainicjalizować wszystkie Q-wartości z wartością 0 (oprócz czynności niemożliwych do wykonania, którym wyznaczamy Q-wartości równe $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf dla niemożliwych do wykonania czynności
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # Dla wszystkich możliwych czynności
```

Pozwólmy w końcu działać algorytmowi iteracji Q-wartości. Będzie on systematycznie stosował wzór zawarty w równaniu 18.3 we wszystkich Q-wartościach, dla każdego stanu i dla każdej możliwej czynności:

```
gamma = 0.90 # Stopa dyskontowa

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
            for sp in range(3)])
```

To wszystko! Uzyskujemy następujące Q-wartości:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])
```

Na przykład gdy agent znajduje się w stanie s_0 i wybierze czynność a_1 , to oczekiwana suma przyszłych nagród dyskontowych wyniesie w przybliżeniu 17,0.

Sprawdźmy, która czynność ma największą Q-wartość dla każdego stanu:

```
>>> np.argmax(Q_values, axis=1) # Optymalna czynność dla każdego stanu
array([0, 0, 1])
```

Uzyskujemy w ten sposób optymalną strategię dla tego procesu MDP przy stopie dyskontowej o wartości 0,90: w stanie s_0 jest wybierana czynność a_0 , w stanie s_1 czynność a_0 (agent pozostaje w tym samym miejscu), a w stanie s_2 czynność a_1 (jedyna dostępna). Co ciekawe, po zwiększeniu stopy dyskontowej do wartości 0,95 zmianie ulega optymalna strategia: w stanie s_1 najlepszą czynnością staje się a_2 (agent przechodzi przez płomień!). Jest to logiczne, ponieważ jeżeli cenimy bardziej przyszłość od teraźniejszości, to lepiej teraz trochę pocierpieć, żeby później dostąpić rozkoszy.

Uczenie metodą różnic czasowych

Problemy analizowane za pomocą uczenia przez wzmacnianie wykorzystujące czynności dyskretne często można przedstawić w postaci procesów decyzyjnych Markowa, ale agent początkowo nie zna prawdopodobieństw przejścia pomiędzy poszczególnymi stanami ($T(s, a, s')$) ani wartości nagród

$(R(s, a, s))$. Musi on przynajmniej raz znaleźć się w każdym stanie i wykonać każde możliwe przejście, aby poznać nagrody, a w celu optymalnego oszacowania prawdopodobieństw przejść musi wykonać wiele takich przebiegów.

Algorytm uczenia **metodą różnic czasowych** (ang. *temporal difference* — TD) bardzo przypomina algorytm iteracji wartości, ale brany jest w nim pod uwagę fakt, że agent ma jedynie częściową wiedzę na temat procesu MDP. Ogólnie zakładamy, że agent początkowo zna jedynie możliwe stany i czynności, nic więcej. Agent wykorzystuje **strategię poszukiwania** (ang. *exploration policy*) — może być ona całkowicie losowa — do poznawania procesu MDP, a w miarę postępów algorytm TD aktualizuje oszacowania wartości stanów na podstawie zaobserwowanych przejść i nagród (równanie 18.4).

Równanie 18.4. Algorytm uczenia TD

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

lub równoważne:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

$$\text{dla } \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

W tym równaniu:

- α jest współczynnikiem uczenia (np. 0,01);
- $r + \gamma \cdot V_k(s')$ to **cel TD** (ang. *TD target*);
- $\delta_k(s, r, s')$ stanowi **błąd TD** (ang. *TD error*).

Można stosować zwięzlszy zapis pierwszej formy tego równania, gdzie wprowadzamy notację $a \xleftarrow{\alpha} b$, która oznacza $a_{k+1} \leftarrow (1-\alpha) \cdot a_k + \alpha \cdot b_k$. Zatem pierwszy wiersz w równaniu 18.4 możemy uprościć do następującej postaci: $V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$.



Uczenie metodą różnic czasowych pod wieloma względami przypomina stochastyczny spadek wzduż gradientu, zwłaszcza w sposobie przyjmowania danych wejściowych — każdorazowo przetwarza tylko jeden przykład. Podobnie jak w przypadku algorytmu SGD, możemy uzyskać tu zbieżność jedynie wtedy, gdy będziemy stopniowo zmniejszać wartość współczynnika uczenia (w przeciwnym razie model będzie ciągle pomijał optymalne Q-wartości).

Algorytm ten dla każdego stanu s śledzi średnią kroczącą nagród otrzymywanych przez agenta tuż po opuszczeniu tego stanu, a także wartości nagród oczekiwanych w przyszłości (przy założeniu, że agent będzie wybierał optymalne czynności).

Q-uczenie

W analogiczny sposób algorytm Q-uczenia (ang. *Q-Learning*) stanowi adaptację algorytmu iteracji Q-wartości do sytuacji, w której prawdopodobieństwa przejść i wartości nagród są początkowo nieznane (równanie 18.5). Algorytm Q-uczenia działa poprzez obserwowanie gry agenta (np. losowej) i stopniowe poprawianie jego oszacowań Q-wartości. Po uzyskaniu dokładnych (choćby w przybliżeniu) oszacowań Q-wartości zadaniem optymalnej strategii jest wybór czynności o największej Q-wartości (strategia zachłanna).

Równanie 18.5. Algorytm Q-uczenia

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} Q(s', a')$$

Dla każdej pary stan – czynność (s, a) algorytm ten śledzi średnią kroczącą nagrodę r , które otrzymuje agent, gdy opuści stan s po wybraniu czynności a , a także nagrody spodziewane w przyszłości. Strategia docelowa działałaby optymalnie, dlatego aby oszacować tę sumę, przyjmujemy maksymalne oszacowania Q-wartości dla następnego stanu s' , ponieważ zakładamy, że od tego momentu strategia docelowa będzie działać optymalnie.

Zaimplementujmy algorytm Q-uczenia. Musimy najpierw sprawić, aby agent poznawał środowisko. W tym celu wprowadzimy funkcję skokową, dzięki czemu agent po wykonaniu jednej czynności uzyska informacje o stanie i nagrodzie:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Zaimplementujmy teraz strategię poszukiwania. Przestrzeń stanów nie jest zbyt rozbudowana, dlatego wystarczy nam tu prosta strategia losowa. Jeżeli algorytm będzie działał wystarczająco długo, agent wielokrotnie odwiedzi każdy stan i również spróbuje wiele razy wykonać każdą czynność:

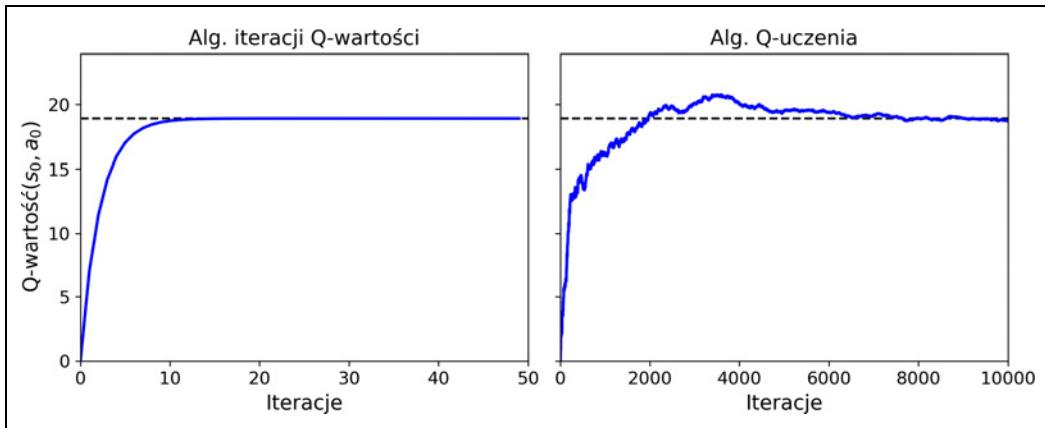
```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Następnie po zainicjalizowaniu już Q-wartości w znany nam sposób jesteśmy gotowi uruchomić algorytm Q-uczenia z wprowadzonym zanikiem współczynnika uczenia (za pomocą omówionego w rozdziale 11. harmonogramowania potęgowego):

```
alpha0 = 0.05 # Początkowy współczynnik uczenia
decay = 0.005 # Parametr zaniku współczynnika uczenia
gamma = 0.90 # Stopa dyskontowa
state = 0 # Stan początkowy

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

Algorytm uzyska zbieżność z optymalnymi Q-wartościami, ale zajmie mu to mnóstwo iteracji i, prawdopodobnie, strojenia hiperparametrów. Jak widać na rysunku 18.9, algorytm iteracji Q-wartości (po lewej) uzyskuje zbieżność bardzo szybko, w mniej niż 20 przebiegach, natomiast algorytm Q-uczenia (po prawej) potrzebuje na to zadanie mniej więcej 8000 iteracji. Jest oczywiste, że brak znajomości prawdopodobieństw przejścia lub nagród znacznie utrudnia znalezienie optymalnej strategii!



Rysunek 18.9. Porównanie algorytmu iteracji Q-wartości (po lewej) z algorytmem Q-uczenia (po prawej)

Algorytm Q-uczenia jest nazywany algorytmem **bez strategii** (ang. *off-policy*), gdyż wyuczona strategia niekoniecznie musi być tą, która będzie używana — w poprzednim przykładzie realizowana strategia (poszukiwania) jest zupełnie losowa, natomiast uczona strategia będzie zawsze wybierała czynności o największych Q-wartościach. Z kolei algorytm strategii gradientów jest algorytmem **ze strategią** (ang. *on-policy*), ponieważ poznaje świat za pomocą wyuczonej strategii. Jest dość zaskakujące, że algorytm ten jest w stanie nauczyć się optymalnej strategii poprzez obserwację losowego zachowania agenta (wyobraź sobie naukę gry w golfa, gdy Twoim nauczycielem jest pijana małpa). Czy jesteśmy w stanie osiągać lepsze wyniki?

Strategie poszukiwania

Oczywiście algorytm Q-uczenia może działać tylko wtedy, jeżeli strategia poszukiwania może odpowiednio dogłębiście przeanalizować proces MDP. Całkowicie losowa strategia może wielokrotnie odwiedzić każdy stan i wykonać każde możliwe przejście, może to jednak zająć mnóstwo czasu. Dlatego lepszym rozwiązaniem jest skorzystanie ze **strategii ϵ -zachłannej** (ang. *ϵ -greedy policy*): w każdym kroku zachowuje się ona losowo z prawdopodobieństwem ϵ lub zachłannie (w przypadku wyboru czynności o największej Q-wartości) z prawdopodobieństwem $1-\epsilon$. Zaletą tego rodzaju strategii (w porównaniu do strategii całkowicie losowej) jest fakt, że poświęca ona coraz więcej czasu na przeszukiwanie interesujących obszarów środowiska wraz z coraz lepszymi oszacowaniami Q-wartości, a jednocześnie znajduje czas na zwiedzanie nieznanych regionów procesu MDP. Dość powszechnie jest rozpoczęcie pracy z dużą wartością ϵ (np. 1,0), a następnie jej stopniowe zmniejszanie (np. do 0,05).

Nie musimy jednak polegać na prawdopodobieństwie poszukiwania — alternatywną metodą jest zatęcenie strategii poszukiwania do sprawdzania rzadko wykonywanych czynności. Możemy to zrobić, wstawiając dodatkowy człon do oszacowań Q-wartości (równanie 18.6).

Równanie 18.6. Algorytm Q-uczenia wykorzystujący funkcję poszukiwania

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a'))$$

W tym równaniu:

- $N(s, a)$ zlicza, ile razy czynność a została wybrana w stanie s ;
- $f(Q, N)$ to **funkcja poszukiwania** (ang. *exploration function*), np. $f(Q, N) = Q + \kappa/(1+N)$, gdzie κ stanowi hiperparametr ciekawości określający stopień zainteresowania agenta nieznanymi obszarami.

Przybliżający algorytm Q-uczenia i Q-uczenie głębokie

Algorytm Q-uczenia ma jeden spory problem: nie skaluje się zbyt dobrze do dużych (a nawet średnich) procesów MDP składających się z wielu stanów i czynności. Założymy, że chcemy wykorzystać go do uczenia agenta gry w Ms. Pac-Man (rysunek 18.1). Istnieje ponad 150 kulek, które może zjeść Pani Pac-Manowa, a każda z nich może być dostępna lub niedostępna (już zjedzona). Zatem liczba możliwych stanów jest większa niż $2^{150} \approx 10^{45}$. A jeżeli dodamy do tego wszystkie możliwe kombinacje położeń duszków w Pani Pac-Manowej, to liczba ta znacznie przekroczy liczbę atomów tworzących naszą planetę, zatem nie mamy najmniejszej możliwości śledzenia oszacowań każdej Q-wartości.

Rozwiązańiem okazuje się znalezienie funkcji $Q_\theta(s, a)$ aproksymującej Q-wartość dowolnej pary stan – czynność (s, a) przy użyciu zarządzalnej liczby parametrów (przy danym wektorze parametrów θ). Jest to tzw. **przybliżający algorytm Q-uczenia** (ang. *approximate Q-learning*). Przez wiele lat zalecano stosowanie liniowych kombinacji ręcznie generowanych cech wydobywanych ze stanu (np. odległość do najbliższych duszków, kierunek ich ruchu itd.) do szacowania Q-wartości, ale w 2013 roku system DeepMind (<https://arxiv.org/abs/1312.5602>) udowodnił, że można osiągnąć o wiele lepsze wyniki za pomocą głębokich sieci neuronowych, zwłaszcza w przypadku skomplikowanych problemów. Dodatkową zaletą tego rozwiązania jest brak konieczności sztucznego tworzenia cech. Sieć DNN wykorzystywana do szacowania Q-wartości jest nazywana **głęboką Q-siecią** (ang. *deep Q-network* — DQN), natomiast stosowanie sieci DQN wraz z przybliżającym algorytmem Q-uczenia to **głębokie Q-uczenie** (ang. *deep Q-learning*).

Jak możemy uczyć sieć DQN? Weźmy pod uwagę przybliżoną Q-wartość obliczoną przez sieć DQN dla danej pary stan – czynność (s, a) . Wiemy dzięki Bellmanowi, że ta przybliżona Q-wartość będzie maksymalnie zbliżona do sumy nagrody r , którą rzeczywiście obserwujemy po wykonaniu czynności a w stanie s , i wartości dyskontowej uzyskanej poprzez optymalną rozgrywkę od tego momentu. Aby oszacować tę sumę przyszłych nagród dyskontowych, wystarczy po prostu dostarczyć sieci DQN przyszły stan s i wszystkie możliwe czynności a . Otrzymamy w ten sposób przybliżoną przyszłą Q-wartość dla każdej możliwej czynności. Wybieramy następnie największą Q-wartość (gdyż zakładamy, że będziemy grać optymalnie) i obniżamy ją, co da nam oszacowanie sumy

przyszłych nagród dyskontowych. Jeżeli zsumujemy nagrodę r i oszacowanie przyszłej wartości dyskontowej, otrzymamy Q-wartość docelową $y(s, a)$ dla pary stan – czynność (s, a) (równanie 18.7).

Równanie 18.7. Q-wartość docelowa

$$Q_{\text{docelowa}}(s, a) = r + \gamma \cdot \max_{a'} Q_\theta(s', a')$$

Z taką Q-wartością docelową możemy wykonać krok uczący za pomocą dowolnego algorytmu gradientu prostego. Zasadniczo staramy się minimalizować błąd kwadratowy pomiędzy szacowaną Q-wartością $Q(s, a)$ a Q-wartością docelową (lub funkcję straty Hubera w celu zmniejszenia czułości algorytmu na duże wartości błędów). I to tyle, jeśli chodzi o podstawowy algorytm uczenia Q-głębokiego! Zaimplementujmy go w naszym środowisku CartPole.

Implementacja modelu Q-uczenia głębokiego

W pierwszej kolejności musimy utworzyć głęboką Q-sieć. Teoretycznie potrzebujemy sieci neuronowej, która przyjmuje na wejściu parę stan – czynność, a na wyjściu generuje przybliżoną Q-wartość, w praktyce jednak znacznie skuteczniejszym rozwiązańiem okazuje się użycie sieci neuronowej, która przyjmuje stan i generuje po jednej przybliżonej Q-wartości na każdą możliwą czynność. Do rozwiązania problemu zdefiniowanego w środowisku CartPole nie potrzebujemy bardzo skomplikowanej sieci neuronowej — wystarczą nam dwie warstwy ukryte:

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

Sieć ta będzie wybierać czynność o największej prognozowanej Q-wartości. Dzięki algorytmowi ε-zachłannemu sprawi, że agent zwiedzi całe środowisko (czyli będziemy dobierać losową czynność z prawdopodobieństwem ε):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Sieć DQN będzie uczona nie tylko na podstawie najnowszych doświadczeń, lecz wprowadzimy także **bufor odtwarzania** (ang. *replay buffer*), zwany również **pamięcią odtwarzania** (ang. *replay memory*), z którego w każdej iteracji będziemy losować grupę danych uczących. Rozwiązanie to pomaga zmniejszyć korelację pomiędzy poszczególnymi doświadczeniami w grupie, co znacząco usprawnia proces uczenia. W tym celu wystarczy wykorzystać listę dwukierunkową (deque):

```
from collections import deque
replay_buffer = deque(maxlen=2000)
```



Lista/kolejka dwukierunkowa (ang. *deque*) jest listą wiązaną, w której każdy element wskazuje element go poprzedzający i po nim następujący. Dzięki temu wstawianie i usuwanie elementów przebiega bardzo szybko, ale im lista dwukierunkowa jest dłuższa, tym więcej czasu trzeba poświęcić na losowy dostęp. Jeżeli potrzebujesz bardzo dużego bufora odtwarzania, skorzystaj z bufora cyklicznego — implementację znajdziesz w sekcji „Lista dwukierunkowa a lista cykliczna” w notatniku Jupyter.

Każde doświadczenie będzie się składać z pięciu elementów: stanu, czynności wybranej przez agenta, otrzymanej nagrody, następnego stanu oraz wartości logicznej wskazującej, czy epizod w tym momencie został zakończony (done). Musimy zdefiniować małą funkcję losującą grupę doświadczeń z bufora odtwarzania. Będzie ona zwracać pięć tablic NumPy zawierających po jednym z wymienionych elementów doświadczenia:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Przygotujmy także funkcję rozgrywającą jeden krok za pomocą strategii ϵ -zachłannej i umieszczającą uzyskanie doświadczenie w buforze odtwarzania:

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

Utwórzmy jeszcze jedną funkcję, która będzie losować grupę doświadczeń z bufora odtwarzania i uczyć się DQN poprzez realizację algorytmu gradientu prostego na tej grupie:

```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Przeanalizujmy ten kod:

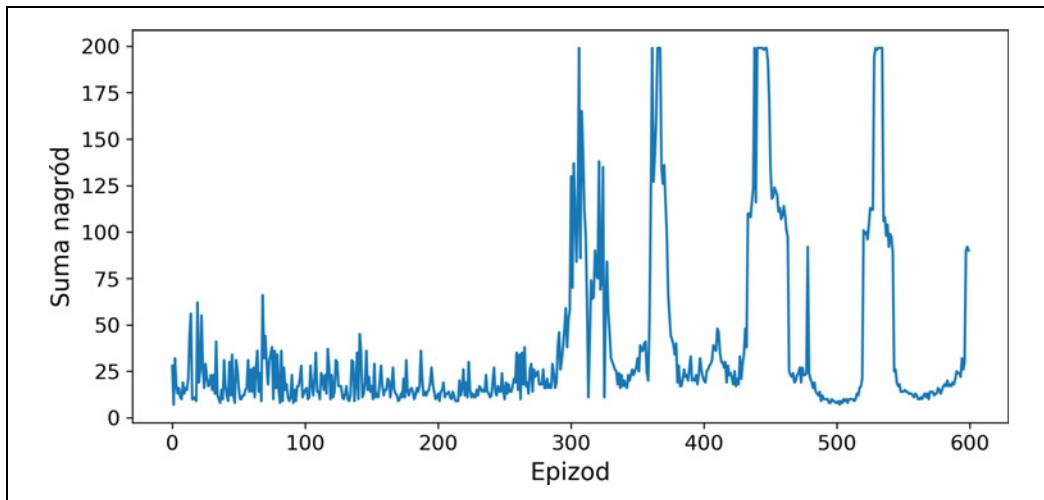
- Najpierw definiujemy hiperparametry oraz tworzymy optymalizator i funkcję straty.
- Potem tworzymy funkcję `training_step()`. Pobiera ona grupę doświadczeń, następnie wykorzystuje sieć DQN do przewidywania Q-wartości każdej możliwej czynności w następnym stanie każdego doświadczenia. Zakładamy, że agent będzie grał optymalnie, zatem zachowujemy jedynie maksymalną Q-wartość każdego następnego stanu. Teraz możemy użyć równania 18.7 do obliczenia Q-wartości docelowej dla pary stan – czynność każdego doświadczenia.
- Chcemy następnie użyć sieci DQN do obliczenia Q-wartości dla każdej dotychczas zrealizowanej pary stan – czynność. Jednocześnie jednak głęboka sieć Q-uczenia będzie generowała również Q-wartości dla pozostałych możliwych czynności, a nie wyłącznie dla tych, które zostały wybrane przez agenta. Zatem musimy ukryć wszystkie niepotrzebne Q-wartości. Funkcja `tf.one_hot()` ułatwia przekształcanie tablicy z indeksami czynności w taką maskę. Na przykład jeżeli trzy pierwsze doświadczenia zawierają czynności, odpowiednio, 1, 1, 0, to początek maski będzie wyglądał następująco: `[[0, 1], [0, 1], [1, 0], ...]`. Możemy następnie przemnożyć wynik sieci DQN przez tę maskę, dzięki czemu zostaną wyzerowane wszystkie niepotrzebne Q-wartości. Potem sumujemy wartości wzduż osi 1 po to, aby usunąć wszystkie zera i pozostawić tylko Q-wartości dla zrealizowanych par stan – czynność. Otrzymujemy tensor `Q_values`, zawierający po jednej prognozowanej Q-wartości dla każdego doświadczenia w grupie.
- Obliczamy teraz funkcję straty: błąd średniokwadratowy pomiędzy Q-wartością docelową a przewidywaną dla zrealizowanych par stan – czynność.
- W końcu przeprowadzamy fazę gradientu prostego, która służy do minimalizowania funkcji straty w odniesieniu do zmiennych modyfikowalnych modelu.

To była najtrudniejsza część. Samo uczenie modelu nie powinno stanowić problemu:

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

Rozgrywamy 600 epizodów, każdy składający się maksymalnie z 200 kroków. W każdym kroku obliczamy najpierw wartość `epsilon` ze strategii ϵ -zachłannej — będzie ona stopniowo maleć od 1 do 0,01 w nieco mniej niż 500 epizodów. Następnie wywołujemy funkcję `play_one_step()`, która za pomocą strategii ϵ -zachłannej będzie wybierała czynność, którą następnie wykona i zapisze doświadczenie w buforze odtwarzania. Wychodzimy z pętli, jeżeli epizod zostanie zakończony. Po rozegraniu 50. epizodu wywołujemy funkcję `training_step()`, która ma za zadanie uczyć model za pomocą jednej grupy wylosowanej z bufora odtwarzania. Nie uczymy modelu przez 50 epizodów po to, aby dać czas na wypełnienie bufora odtwarzania (jeżeli nie poczekamy wystarczająco dugo, to bufor odtwarzania będzie zawierał zbyt mało zróżnicowanych doświadczeń). I to tyle, właśnie zaimplementowaliśmy algorytm Q-uczenia głębokiego!

Rysunek 18.10 ukazuje całkowitą sumę nagród, jakie uzyskał agent w poszczególnych epizodach.



Rysunek 18.10. Krzywa uczenia algorytmu DQN

Jak widać, algorytm nie poczynił wyraźnych postępów przez niemal 300 epizodów (częściowo z powodu dużej początkowej wartości ϵ), po czym jego wydajność gwałtownie podskoczyła do 200 (maksymalna wydajność w tym środowisku). Są to znakomite wieści: algorytm działa bardzo dobrze i w istocie jest znacznie szybszy od algorytmu gradientów strategii! Ale chwila... zaledwie kilka epizodów później model zapomniał wszystkiego, czego się nauczył, i jego wydajność spadła poniżej 25! Jest to tzw. **katastrofalne zapominanie** (ang. *catastrophic forgetting*) i stanowi jeden z największych problemów praktycznie wszystkich algorytmów uczenia przez wzmacnianie: w miarę poznawania środowiska przez agenta jego strategia jest aktualizowana, ale to, czego nauczył się w jednym obszarze środowiska, może zniweczyć wszelkie doświadczenia, jakie nabył w innych rejonach. Doświadczenia są ze sobą w pewnym stopniu skorelowane, a środowisko uczenia ciągle podlega zmianom, co nie sprzyja algorytmowi gradientów prostych! Jeżeli zwiększasz rozmiar bufora odtwarzania, model stanie się nieco mniej podatny na ten problem. Pomóc może także zmniejszenie współczynnika uczenia. Prawda jest jednak taka, że uczenie przez wzmacnianie nie jest łatwym zadaniem: proces uczenia jest często niestabilny i być może będziesz musiał/musiać wypróbować wiele różnych wartości hiperparametrów i załączników losowości, zanim znajdziesz stabilną kombinację. Na przykład jeżeli w powyższym przykładzie spróbowujesz zmienić liczbę neuronów tworzących każdą warstwę z 32 na 30 lub 34, to wydajność nigdy nie przekroczy wyniku 100 (głęboka sieć Q-uczenia może być stabilniejsza przy jednej warstwie ukrytej, a nie dwóch).

Być może się zastanawiasz, dlaczego nie stworzyliśmy wykresu funkcji straty. Okazuje się, że w tym przypadku funkcja straty stanowi kiepski wskaźnik wydajności modelu. Wartość funkcji straty może zmaleć, a mimo to agent może sobie radzić jeszcze gorzej (bywa tak wtedy, gdy agent utknie w jakimś małym obszarze środowiska, a sieć ulega przetrenowaniu w tym rejonie). I odwrotnie, wartość funkcji straty może się zwiększyć, ale agent zacznie sobie lepiej radzić (np. sieć DQN zbyt nisko szacowała Q-wartości i zaczyna prawidłowo zwiększać predykcje, a zatem agent zacznie używać lepsze rezultaty i zdobywać większe nagrody, ale funkcja straty będzie rosnąć, ponieważ głęboka sieć Q-uczenia wyznacza również zmienne docelowe, które też będą większe).



Uczenie przez wzmacnianie notorycznie sprawia problemy, w dużej mierze z powodu niestabilności uczenia oraz olbrzymiej wrażliwości na dobór wartości hiperparametrów i załączków losowości¹⁴. Badacz Andrej Karpathy ujął to w następujący sposób: „[Uczenie nadzorowane] chce pracować. [...] Uczenie przez wzmacnianie należy zmusić do pracy”. Będziesz potrzebować czasu, cierpliwości, wytrwałości i być może także odrobiny szczęścia. Jest to główna przyczyna względnie małej popularności uczenia maszynowego w porównaniu do standardowych modeli uczenia głębokiego (np. sieci splotowych). Sieci te znalazły jednak kilka zastosowań w świecie rzeczywistym oprócz modeli AlphaGo i przechodzenia gier na Atari: na przykład firma Google wykorzystuje uczenie przez wzmacnianie do optymalizowania kosztów centrów danych; stosowane są także w pewnych dziedzinach robotyki, do strojenia hiperparametrów i w systemach rekommendacji.

Podstawowy algorytm Q-uczenia głębokiego jest zbyt niestabilny, aby uczyć się przechodzenia gier na Atari. Jak więc dokonał tego model DeepMind? To proste: twórcy usprawnili algorytm!

Odmiany Q-uczenia głębokiego

Przyjrzyjmy się kilku wariantom algorytmu Q-uczenia głębokiego, które mają na celu ustabilizowanie i przyspieszenie procesu uczenia.

Ustalone Q-wartości docelowe

W podstawowym algorytmie Q-uczenia głębokiego model jest używany jednocześnie do obliczania prognoz i do wyznaczania własnych wartości docelowych. Może to doprowadzić do sytuacji przypominającej gajanie psa za własnym ogonem. Taka pętla sprzężenia zwrotnego może zdestabilizować sieć: może stawać się ona rozbieżna z wyznaczonym celem, oscylować, zatrzymywać się w miejscu itd. W celu rozwiązania tego problemu naukowcy z firmy DeepMind w publikacji z 2013 roku wykorzystali nie jedną, lecz dwie sieci DQN: pierwsza, **model trwający** (ang. *online model*), uczy się przy każdym kroku i służy do przemieszczania agenta, natomiast druga, **model docelowy** (ang. *target model*), używana jest wyłącznie do definiowania celów. Model docelowy stanowi jedynie kopię modelu trwającego:

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

W funkcji `training_step()` musimy zmienić jeden wiersz, aby w czasie obliczania Q-wartości następnych stanów skorzystać z modelu docelowego zamiast trwającego:

```
next_Q_values = target.predict(next_states)
```

W końcu w samej pętli uczenia musimy kopiować wagę modelu trwającego do modelu docelowego w regularnych odstępach czasu (np. co 50 epizodów):

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

¹⁴ Największe bolączki i ograniczenia modeli RL zostały znakomicie omówione we wpisie Alexa Irpana z 2018 roku (<https://www.alexirpan.com/2018/02/14/rl-hard.html>).

Model docelowy jest aktualizowany znacznie rzadziej od modelu trwającego, więc Q-wartości stają się stabilniejsze, wspomniana pętla sprzężenia zwrotnego zostaje zredukowana, a jej skutki przestają być tak doniosłe. Rozwiążanie to stanowiło jeden z najważniejszych elementów artykułu z 2013 roku i pozwoliło agentom przechodzić gry na Atari jedynie na podstawie nieprzetworzonych pikseli. W celu ustabilizowania treningu autorzy wprowadzili bardzo mały współczynnik uczenia, rzędu 0,00025, aktualizowali model docelowy zaledwie co 10 000 kroków (a nie co 50) i wprowadzili olbrzymi bufor odtwarzania o rozmiarze miliona doświadczeń. Bardzo powoli zmniejszały wartość ϵ i α , od 1 do 0,1 w ciągu miliona kroków, sam algorytm zaś działał przez 50 milionów kroków.

W dalszej części rozdziału wykorzystamy bibliotekę TF-Agents do nauki agenta DQN gry w Breakout za pomocą tych parametrów, zanim jednak do tego przejdziemy, przyjrzyjmy się innej odmianie sieci DQN, która jeszcze wyżej podniosła poprzeczkę wydajności modeli uczenia przez wzmacnianie.

Podwójna sieć DQN

W publikacji z 2015 roku (<https://arxiv.org/abs/1509.06461>)¹⁵ naukowcy z firmy DeepMind usprawnili algorytm DQN, zwiększyli jego wydajność i niejako ustabilizowali również proces uczenia. Wariant ten został nazwany **podwójną siecią DQN** (ang. *Double DQN*). Usprawnienie bazowało na obserwacji, że sieć docelowa jest podatna na szacowanie zbyt dużych Q-wartości. Rzeczywiście, założymy, że wszystkie czynności są równie dobre: Q-wartości szacowane przez model docelowy powinny być identyczne, ale mamy do czynienia z przybliżeniami, dlatego niektóre z nich mogą być (całkowicie przypadkowo) większe od pozostałych. Model docelowy będzie zawsze wybierał największą Q-wartość, która będzie nieco większa od średniej Q-wartości, co najprawdopodobniej będzie prowadziło do oszacowania zbyt dużej Q-wartości (trocę przypomina to uwzględnienie wysokości najwyższej przypadkowej fali przy mierzeniu głębokości basenu). W celu rozwiążania tego problemu zaproponowano użycie modelu trwającego zamiast docelowego podczas wybierania najlepszych czynności dla następnych stanów, przez co model docelowy służyłby wyłącznie do szacowania Q-wartości dla tych najlepszych czynności. Oto zaktualizowana funkcja `training_step()`:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # Pozostała część wygląda tak samo jak w poprzednim przykładzie
```

Zaledwie kilka miesięcy później pojawiło się kolejne udoskonalenie algorytmu DQN.

¹⁵ Hado van Hasselt i in., *Deep Reinforcement Learning with Double Q-Learning*, „Proceedings of the 30th AAAI Conference on Artificial Intelligence” (2015), s. 2094 – 2100.

Odtwarzanie priorytetowych doświadczeń

Zamiast losować doświadczenia równomiernie z bufora odtwarzania, dlaczego by nie losować częściej ważnych doświadczeń? Koncepcja ta nazywana jest **próbkowaniem ważonym** (ang. *importance sampling* — IS) lub **odtwarzaniem priorytetowych doświadczeń** (ang. *prioritized experience replay* — PER). Została zaproponowana w artykule opublikowanym przez badaczy z firmy DeepMind (znowu!) w 2015 roku (<https://arxiv.org/abs/1511.05952>)¹⁶.

Mówiąc dokładniej, doświadczenia są uznawane za „ważne”, jeżeli prowadzą do szybkich postępów w uczeniu. Ale jak to oszacować? Jedną z rozsądnych metod jest pomiar wielkości błędu TD: $\delta = r + \gamma \cdot V(s') - V(s)$. Duża wartość błędu TD wskazuje, że przejście (s, r, s') jest bardzo zaskakujące, a zatem może warto wyciągnąć z niego wnioski¹⁷. Gdy doświadczenie jest rejestrowane w buforze odtwarzania, zostaje mu wyznaczona bardzo duża wartość priorytetu, dzięki czemu mamy pewność, że zostanie wylosowane przynajmniej raz. Jednak po jego każdorazowym wylosowaniu zostaje obliczony błąd TD δ i wyznaczony priorytet tego doświadczenia równy $p = |\delta|$ (plus mała wartość stała, aby zagwarantować każdemu doświadczeniu niezerowe prawdopodobieństwo wylosowania). Prawdopodobieństwo P wylosowania doświadczenia o ważności p jest proporcjonalne do p^ζ , gdzie ζ jest hiperparametrem kontrolującym zachłanność próbkowania ważonego: gdy $\zeta = 0$, to mamy do czynienia z losowaniem równomiernym, natomiast $\zeta = 0$ wyznacza pełne próbkowanie ważone. Autorzy użyli w artykule wartości $\zeta = 0,6$, ale wartość optymalna zależy od zadania.

Jest tu jednak pewien haczyk: przykłady będą obciążone w kierunku istotnych doświadczeń, dlatego musimy to zrównoważyć w fazie uczenia poprzez zmniejszanie wag doświadczeń zgodnie z ich istotnością, gdyż w przeciwnym razie model po prostu ulegnie przetrenowaniu zgodnie z ważnymi doświadczeniami. Należy jasno stwierdzić, że chcemy, aby istotne doświadczenia były losowane częściej, co oznacza to jednak, że musimy wyznaczać im niższe wagi podczas uczenia. W tym celu definiujemy wagę każdego doświadczenia w fazie uczenia jako $w = (nP)^{-\beta}$, gdzie n oznacza liczbę doświadczeń w buforze odtwarzania, natomiast β stanowi hiperparametr regulujący stopień równoważenia obciążenia próbkowania ważonego (0 oznacza brak równoważenia, 1 określa równoważenie całkowite). We wspomnianym artykule autorzy wprowadzili na początku uczenia wartość $\beta = 0,4$ i stopniowo zwiększały ją do $\beta = 1$ na koniec fazy uczenia. Również w tym przypadku wartość optymalna zależy od zadania, ale jeżeli zwiększasz jeden hiperparametr, zazwyczaj oznacza to konieczność zwiększenia również drugiego hiperparametru.

Pozostał nam do omówienia jeszcze jeden istotny wariant algorytmu DQN.

Walcząca sieć DQN

Algorytm walczącej sieci DQN (ang. *Dueling DQN* — DDQN; nie należy jej mylić z siecią *Double DQN*, chociaż obydwie techniki można łatwo ze sobą łączyć) został opisany w 2015 roku w innej publikacji naukowców z DeepMind (<https://arxiv.org/abs/1511.06581>)¹⁸. Aby zrozumieć mechanizm

¹⁶ Tom Schaul i in., *Prioritized Experience Replay*, arXiv preprint arXiv:1511.05952 (2015).

¹⁷ Równie dobrze jednak nagrody mogą być po prostu zaszumione — w takim przypadku istnieją lepsze metody oszacowania ważności doświadczenia (szczególnie znajdziesz we wspomnianym artykule).

¹⁸ Ziyu Wang i in., *Dueling Network Architectures for Deep Reinforcement Learning*, arXiv preprint arXiv: 1511.06581 (2015).

jego działania, musimy pamiętać, że Q-wartość pary stan – czynność (s, a) można zapisać jako $Q(s, a) = V(s) + A(s, a)$, gdzie $V(s)$ jest wartością stanu s , natomiast $A(s, a)$ stanowi **korzyść** (ang. *advantage*) wynikającą z podjęcia czynności a w stanie s w porównaniu do wszystkich pozostałych czynności w tym stanie. Co więcej, wartość stanu jest równa Q-wartości najlepszej czynności a^* w tym stanie (gdyż zakładamy, że optymalna strategia wybierze najlepszą czynność), zatem $V(s) = Q(s, a^*)$, z czego wynika, że $A(s, a^*) = 0$. W walczącej sieci DQN model oszacowuje zarówno wartość stanu, jak i korzyść wynikającą z każdej możliwej czynności. Najlepsza możliwa czynność powinna mieć korzyść równą 0, dlatego model odejmuje maksymalną przewidywaną korzyść od wszystkich prognozowanych korzyści. Oto prosty model walczącej sieci DQN zaimplementowanej za pomocą interfejsu funkcyjnego:

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

Pozostała część algorytmu jest niezmieniona. W rzeczywistości istnieje możliwość utworzenia podwójnej walczącej sieci DQN i połączenia jej z odtwarzaniem priorytetowych doświadczeń! Zasadniczo można łączyć wiele technik uczenia przez wzmacnianie, co zespół DeepMind zademonstrował w artykule z 2017 roku (<https://arxiv.org/abs/1710.02298>)¹⁹. Autorzy utworzyli z sześciu różnych technik agenta nazwanego **Rainbow**, który znacznie przewyższył skutecznością najnowocześniejsze modele.

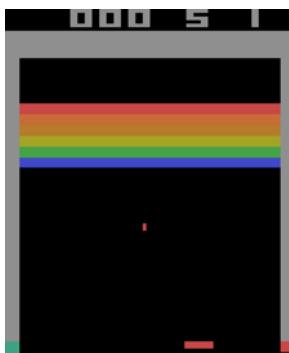
Niestety implementowanie, poprawianie, dostrajanie i oczywiście uczenie wszystkich tych modeli bywa bardzo pracochłonne. Dlatego wymyślanie koła na nowo nie ma sensu i lepiej korzystać z gotowych skalowalnych i przetestowanych bibliotek, takich jak TF-Agents.

Biblioteka TF-Agents

Biblioteka TF-Agents (<https://github.com/tensorflow/agents>) jest opartą na module TensorFlow biblioteką uczenia przez wzmacnianie. Została zaprojektowana przez firmę Google i opublikowana w 2018 roku. Podobnie jak narzędzie OpenAI Gym zawiera ona wiele gotowych środowisk (w tym funkcje umożliwiające dołączanie środowisk OpenAI Gym), a także obsługę bibliotek PyBullet (symuluje fizykę w przestrzeniach trójwymiarowych), DM Control z DeepMind (oparta na silniku fizyki MuJoCo) oraz ML-Agents ze środowiskiem Unity (symuluje wiele środowisk trójwymiarowych). Implementuje ona również wiele algorytmów uczenia przez wzmacnianie, takich jak REINFORCE, DQN czy DDQN, oraz różne składniki RL, na przykład wydajne bufory odtwarzania i wskaźniki. Jest ona szybka, skalowalna, przystępna i modyfikowalna — możesz tworzyć własne środowiska i sieci neuronowe oraz konfigurować zasadniczo każdy element. W tym podrozdziale wykorzystamy

¹⁹ Matteo Hessel i in., *Rainbow: Combining Improvements in Deep Reinforcement Learning*, arXiv preprint arXiv:1710.02298 (2017), s. 3215 – 3222.

bibliotekę TF-Agents do nauczenia agenta gry w Breakout, słynną grę na Atari (rysunek 18.11)²⁰ za pomocą algorytmu DQN (jeśli chcesz, możesz z łatwością zmienić algorytm).



Rysunek 18.11. Słynna gra Breakout

Instalacja biblioteki TF-Agents

Zacznijmy od zainstalowania biblioteki TF-Agents. Możesz użyć w tym celu menedżera pip (jak zawsze nie zapomnij aktywować środowiska wirtualnego, jeśli z niego korzystasz; w przeciwnym razie musisz użyć opcji --user lub wykorzystać uprawnienia administratora):

```
$ python3 -m pip install -U tf-agents
```



Gdy pisałem tę książkę, biblioteka była jeszcze ciągle usprawnianą nowością, dlatego API mogło zostać zmodyfikowane, ale ogólne koncepcje oraz większość kodu powinny pozostać niezmienione. Jeżeli coś przestanie działać, będę aktualizował notatnik Jupyter.

Utwórzmy teraz środowisko TF-Agents, w którym umieścimy środowisko Breakout z pakietu OpenAI Gym. Aby to zrobić, musimy najpierw zainstalować biblioteki zależne Atari:

```
$ python3 -m pip install -U 'gym[atari]'
```

Zostanie zainstalowana między innymi biblioteka atari-py, czyli interfejs środowiska ALE (ang. *Arcade Learning Environment* — środowisko uczenia się przechodzenia gier zręcznościowych), opartego na emulatorze Stella (emulującym komputer Atari 2600).

Środowiska TF-Agents

Jeżeli instalacja przebiegła pomyślnie, możesz zimportować bibliotekę TF-Agents i utworzyć środowisko gry Breakout:

```
>>> from tf_agents.environments import suite_gym  
>>> env = suite_gym.load("Breakout-v4")
```

²⁰ Jeżeli nie znasz jeszcze tej gry, jej zasady są proste: widoczne na środku ekranu cegielki rozbijasz pileczką, odbijając ją znajdującą się na dole ekranu paletką, która może się poruszać w prawo lub w lewo, przy czym musisz tak odbijać pileczkę, aby zniszczyć wszystkie cegielki, a jednocześnie pilnować, żeby nie wypadła poza obszar pod paletką.

```
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

Jest to jedynie funkcja otaczająca środowisko OpenAI Gym, do którego możesz uzyskać dostęp za pomocą atrybutu `gym`:

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

Szrodowiska TF-Agents bardzo przypominają szrodowiska OpenAI Gym, wystepują jednak pomiędzy nimi pewne różnice. Przede wszystkim metoda `reset()` nie zwraca obserwacji, lecz obiekt `TimeStep` otaczający tę obserwację, a także zawierający dodatkowe informacje:

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

Również metoda `step()` zwraca obiekt `TimeStep`:

```
>>> env.step(1) # Przycisk Fire
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

Znaczenia atrybutów `reward` i `observation` nie trzeba chyba wyjaśniać, gdyż są one takie same jak w szrodowisku OpenAI Gym (tutaj jednak `reward` ma postać tablicy NumPy). Atrybut `step_type` jest równy 0 w pierwszym taktcie epizodu, ma wartość 1 dla pośrednich taktów i 2 dla taktu końcowego. Możesz wywołać dla taktu metodę `is_last()`, aby sprawdzić, czy jest on ostatni. Z kolei atrybut `discount` określa stopę dyskontową użytą w danym taktcie. W omawianym przykładzie jest ona równa 1, zatem nagroda nie będzie wcale obniżona. Możesz wyznaczyć stopę dyskontową poprzez zdefiniowanie parametru `discount` w czasie wczytywania szrodowiska.



Możesz uzyskać dostęp do bieżącego taktu szrodowiska w dowolnym momencie za pomocą wywołania metody `current_time_step()`.

Specyfikacja szrodowiska

Dla naszej wygody szrodowisko TF-Agents zawiera specyfikację obserwacji, czynności i taktów, w tym takie parametry jak wymiary, typy danych i nazwy, a także wartości minimalne i maksymalne:

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                  minimum=[[0. 0. 0.], [0. 0. 0.], ...]],
                  maximum=[[255., 255., 255.], [255., 255., 255.], ...]])

>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                  minimum=0, maximum=3)

>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
         discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

Jak widać, obserwacje to zwykłe zrzuty ekranu z komputera Atari, reprezentowane jako tablice NumPy o wymiarach [210, 160, 3]. W celu wyświetlenia tego środowiska możesz wywołać metodę `env.render(mode="human")`, a jeżeli chcesz wrócić do obrazu w postaci tablicy NumPy, wywołaj metodę `env.render(mode="rgb_array")` (w przeciwnieństwie do środowiska OpenAI Gym tutaj jest to tryb domyślny).

Dostępne są cztery czynności. Środowiska Atari zawierają dodatkową metodę, która po wywołaniu ukazuje kolejność poszczególnych typów czynności:

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```



Parametry te mogą być wystąpieniami klasy specyfikacji, listami zagnieżdzonymi lub słownikami. Jeżeli specyfikacja jest zagnieżdzona, to dany obiekt musi być zgodny z zagnieżdzoną strukturą tej specyfikacji. Na przykład jeżeli specyfikacja obserwacji ma postać `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, to właściwa obserwacja wyglądałaby następująco: `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`. Pakiet `tf.nest` zawiera narzędzia pozwalające na obsługę takich struktur zagnieżdzonych (tzw. **gniazd** — ang. *nests*).

Obserwacje są dość duże, dlatego je zmniejszymy i zredukujemy do jednego kanału (czarno-białe). Przyspieszymy w ten sposób proces uczenia i zaoszczędzimy pamięć operacyjną. Posłużymy się w tym celu **funkcją opakowującą środowisko** (ang. *environment wrapper*).

Funkcje opakowujące środowisko i wstępne przetwarzanie środowiska Atari

Biblioteka TF-Agents zawiera kilka funkcji opakowujących środowisko w pakiecie `tf_agents.environments.wrappers`. Jak sama nazwa wskazuje, opakowują one środowisko i przekierowują do niego każde wywołanie, ale dodają także pewne możliwości. Oto niektóre z dostępnych funkcji opakowujących:

ActionClipWrapper

Przycina czynność do specyfikacji czynności.

ActionDiscretizeWrapper

Przekształca ciągłą przestrzeń czynności w przestrzeń dyskretną. Na przykład jeżeli przestrzeń czynności w pierwotnym środowisku ma ciągły zakres od -1,0 do 1,0, ale chcesz skorzystać z algorytmu obsługującego wyłącznie wartości dyskretne (np. sieć DQN), możesz otoczyć to środowisko za pomocą funkcji `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)`, a nowy obiekt `discrete_env` będzie miał dyskretną przestrzeń czynności wyznaczającą pięć możliwych czynności: 0, 1, 2, 3, 4. Czynności te stanowią odpowiedniki czynności -1,0, -0,5, 0, 0,5 i 1,0 w pierwotnym środowisku.

ActionRepeat

Powtarza każdą czynność przez n kroków i gromadzi nagrody. Proces ten może znacząco przyspieszyć uczenie w wielu środowiskach.

RunStats

Rejestruje statystyki środowiska, takie jak liczba kroków czy liczba epizodów.

TimeLimit

Przerywa działanie, jeżeli algorytm będzie działał dłużej niż wyznaczona maksymalna liczba kroków.

VideoWrapper

Nagrywa dane środowisko jako plik wideo.

Aby utworzyć środowisko opakowane, musisz utworzyć funkcję opakowującą i przekazać środowisko do konstruktora. I to tyle! Na przykład ten oto listing opakowuje środowisko w funkcji ActionRepeat, dzięki czemu każda czynność będzie powtarzana czterokrotnie:

```
from tf_agents.environments.wrappers import ActionRepeat  
  
repeating_env = ActionRepeat(env, times=4)
```

Narzędzie OpenAI Gym zawiera własne funkcje opakowujące w pakiecie gym.wrappers. Służą one jednak do otaczania środowisk Gym, nie TF-Agents, zatem jeśli chcesz z nich korzystać, musisz najpierw opakować środowisko Gym za pomocą funkcji opakowującej Gym, a następnie tak uzyskane środowisko opakować za pomocą funkcji opakowującej TF-Agents. Zadanie to wykona za Ciebie funkcja suite_gym.wrap_env() pod warunkiem, że dostarczysz jej środowisko Gym, a także listę funkcji opakowujących Gym i/lub listę funkcji opakowujących TF-Agents. Ewentualnie funkcja suite_gym.load() utworzy środowisko Gym i opakuje je za Ciebie, jeżeli podasz jej funkcje opakowujące. Każda funkcja opakowująca zostanie utworzona bez argumentów, jeżeli więc chcesz jakieś zdefiniować, musisz przekazać parametr lambda. Na przykład w tym listingu tworzymy środowisko Breakout, które w każdym epizodzie będzie działało maksymalnie przez 10 000 kroków, a każda czynność będzie powtarzana czterokrotnie:

```
from gym.wrappers import TimeLimit  
  
limited_repeating_env = suite_gym.load(  
    "Breakout-v4",  
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],  
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

W większości publikacji wykorzystujących środowiska Atari przeprowadzane są operacje wstępnego przetwarzania, dlatego biblioteka TF-Agents zawiera przydatną funkcję opakowującą Atari →Preprocessing, która ułatwia ich przeprowadzanie. Oto lista dostępnych operacji wstępnego przetwarzania:

Odcienie szarości i zmniejszanie rozmiaru

Obserwacje są przekształcane do obrazu czarno-białego i zmniejszane (domyślnie do rozmiaru 84×84).

Łączenie maksymalizujące

Ostatnie dwie klatki gry są przetwarzane za pomocą maksymalizującej warstwy łączącej z filtrem 1×1. Ma to na celu zapobieganie migotaniu obrazu, które występuje w niektórych grach

na Atari i jest spowodowane ograniczoną liczbą obiektów typu sprite, jaką Atari 2600 mógł wyświetlać w jednej klatce.

Pomijanie klatek

Agent otrzymuje jedynie co n -tą klatkę gry (domyślnie $n = 4$) i jego czynności są powtarzane w każdej klatce, a uzyskane nagrody są dodawane. Z perspektywy agenta rozwiążanie to znacznie przyspiesza grę, a także proces uczenia, ponieważ występuje mniejsze opóźnienie dostarczania nagród.

Koniec gry w wyniku utraty życia

W niektórych grach nagrody bazują wyłącznie na wyniku, zatem agent nie zostaje natychmiast ukarany za utratę życia. Jednym z rozwiązań jest natychmiastowe przerwanie gry w momencie utraty życia. Nie ma jednoznacznosci w kwestii zalet tej strategii, dlatego mechanizm ten jest domyślnie wyłączony.

Środowisko Atari samo w sobie domyślnie stosuje losowe pomijanie klatek i łączenie maksymalizujące, dlatego musimy wczytać pierwotny, niezmodyfikowany wariant o nazwie "BreakoutNoFrameskip-v4". Ponadto jedna klatka z gry Breakout nie wystarczy do określenia kierunku i szybkości kulki, przez co agentowi byłoby bardzo trudno odpowiednio przechodzić grę (chyba że byłby to agent rekurencyjny, który przechowywałby pewien stan wewnętrzny pomiędzy poszczególnymi krokami). Jednym z rozwiązań jest użycie funkcji opakowującej środowisko, która będzie generowała obserwacje składające się z wielu ułożonych na siebie klatek wzduż wymiaru kanałów. Strategia ta jest dostępna w funkcji FrameStack4, która zwraca stosy czteroklatkowe. Utwórzmy opakowane środowisko Atari!

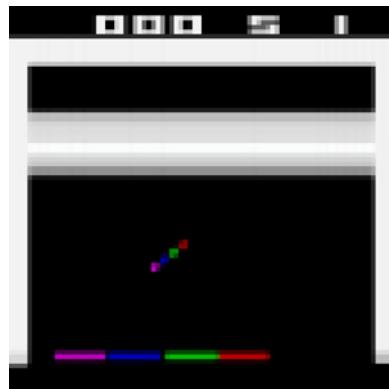
```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108 tysięcy klatek środowiska ALE, gdyż 1 krok = 4 klatki
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

Rezultat tego wstępniego przetwarzania został pokazany na rysunku 18.12. Jak widać, rozdzielcość jest znacznie mniejsza, ale wystarczająca do grania. Ponadto klatki są układane wzduż osi kanałów, więc kolor czerwony reprezentuje klatkę przed trzema, zielony przed dwoma krokami, niebieski z poprzedniego kroku, a różowy z bieżącego²¹. Z takiej pojedynczej obserwacji agent może wywnioskować, że piłeczka leci w stronę lewego dolnego rogu i powinien kontynuować przesuwanie paletki w lewo (kontynuować dotyczczącową czynność).

²¹ Istnieją tylko trzy kolory podstawowe, dlatego nie możemy wyświetlać obrazu z czterema kanałami barw. Z tego powodu połączylem ostatni kanał z pierwszymi trzema, aby otrzymać widoczny na rysunku kolorowy obraz. Kolor różowy stanowi połączenie barw niebieskiej i czerwonej, ale agent widzi cztery niezależne kanały.



Rysunek 18.12. Wstępnie przetworzona obserwacja z gry Breakout

Na koniec możemy jeszcze umieścić to środowisko wewnątrz klasy TFPyEnvironment:

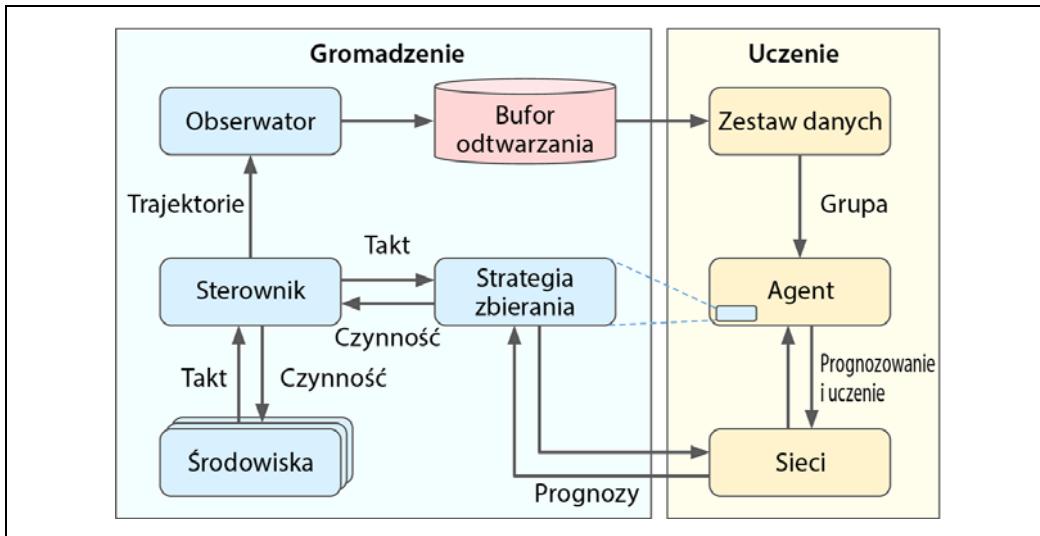
```
from tf_agents.environments.tf_py_environment import TFPyEnvironment  
  
tf_env = TFPyEnvironment(env)
```

W ten sposób środowisko będzie używane z poziomu grafu TensorFlow (klasa ta jest oparta na funkcji `tf.py_funtion()`, która umożliwia grafowi wywoływanie niestandardowego kodu Pythona). Dzięki klasie `TFPyEnvironment` biblioteka TF-Agents obsługuje zarówno wyłącznie środowiska Pythona, jak i środowiska oparte na module TensorFlow. Mówiąc ogólnie, biblioteka ta obsługuje i dostarcza zarówno składniki pochodzące z języka Python, jak i elementy modułu TensorFlow (agenty, bufore odtwarzania, wskaźniki itd.).

Skoro mamy już ładne środowisko gry Breakout, poddane odpowiednim operacjom wstępnego przetwarzania i zawierające obsługę modułu TensorFlow, musimy teraz utworzyć agenta DQN i inne składniki, za pomocą których będziemy go uczyć. Spójrzmy na strukturę tworzonego przez nas systemu.

Architektura ucząca

Tworzony za pomocą biblioteki TF-Agents program uczący składa się zazwyczaj z dwóch równolegle działających składników, co widać na rysunku 18.13: po lewej stronie **sterownik** (ang. *driver*) przeszukuje **środowisko** za pomocą **strategii zbierania** (ang. *collect policy*) i gromadzi **trajektorie** (ang. *trajectories*), np. doświadczenia, które przesyła do **obserwatora** (ang. *observer*) zachowującego je w **buforze odtwarzania**; po prawej stronie **agent** pobiera grupy trajektorii z bufora odtwarzania i trenuje niektóre **sieci** wykorzystywane przez strategię zbierania. Mówiąc krótko, moduł po lewej stronie poznaje środowisko i gromadzi trajektorie, natomiast moduł po prawej uczy się i aktualizuje strategię zbierania.



Rysunek 18.13. Typowa architektura uczenia utworzona za pomocą biblioteki TF-Agents

Pojawia się tutaj kilka pytań, na które postaram się odpowiedzieć:

- Dlaczego występuje tu wiele środowisk? Zazwyczaj chcemy, żeby sterownik przeszukiwał nie jedno środowisko, lecz wiele jego kopii równocześnie, gdyż chcemy maksymalnie wykorzystać możliwości wielu rdzeni procesora i kart graficznych i uzyskiwać mniej skorelowane trajektorie, dostarczane algorytmowi uczącemu.
- Czym jest **trajektoria**? Jest to zwięzła reprezentacja **przejścia** z jednego taktu do następnego lub sekwencja kolejnych przejść z taktu n do taktu $n+t$. Trajektorie zebrane przez sterownik są przekazywane do obserwatora, który zapisuje je w buforze odtwarzania, następnie zaś są losowane przez agenta i zostają użyte w fazie uczenia.
- Do czego służy obserwator? Czy sterownik nie może bezpośrednio zapisywać trajektorii? Móglby, ale zmniejszyłoby to elastyczność architektury. Możesz na przykład nie chcieć korzystać z bufora odtwarzania. Możesz przeznaczyć trajektorie do innego zadania, takiego jak obliczanie wskaźników. W rzeczywistości obserwatorem może być dowolna funkcja przyjmująca trajektorie jako argument. Możesz za pomocą obserwatora zapisywać trajektorie w buforze odtwarzania lub do pliku TFRecord (zob. rozdział 13.), obliczać wskaźniki itd. Ponadto jesteś w stanie przekazywać sterownikowi wiele obserwatorów, a on z kolei będzie przesyłał trajektorie do każdego z tych obserwatorów.



Architektura ta jest najpowszechniejsza, ale możesz ją dowolnie modyfikować, a nawet zastępować niektóre składniki własnymi. W rzeczywistości, jeśli nie opracowujesz nowego algorytmu RL, to najprawdopodobniej chcesz realizować zadanie w niestandardowym środowisku. W tym celu musisz utworzyć niestandardową klasę, która dziedziczy z klasy PyEnvironment należącej do pakietu `tf_agents.environments`.
`→py_environment`, i przesłonić odpowiednie metody, takie jak `action_spec()`, `observation_spec()`, `_reset()` czy `_step()` (przykład znajdziesz w sekcji „Tworzenie niestandardowego środowiska TF-Agents” w notatniku Jupyter).

Utworzmy teraz te wszystkie składniki: najpierw Q-sieć głęboką, potem agenta DQN (będzie odpowiadał za tworzenie strategii zbierania), bufor odtwarzania i związanego z nim obserwatora, sterownik, a na końcu zestaw danych. Po przygotowaniu tych elementów umieścimy w buforze odtwarzania jakieś początkowe trajektorie i uruchomimy główną pętlę uczenia. Zaczniemy więc od przygotowania Q-sieci głębokiej.

Tworzenie Q-sieci głębokiej

Biblioteka TF-Agents zawiera wiele różnych sieci w pakiecie `tf_agents.networks` i jego podpakietach. Skorzystamy z klasy `tf_agents.networks.q_network.QNetwork`:

```
from tf_agents.networks.q_network import QNetwork

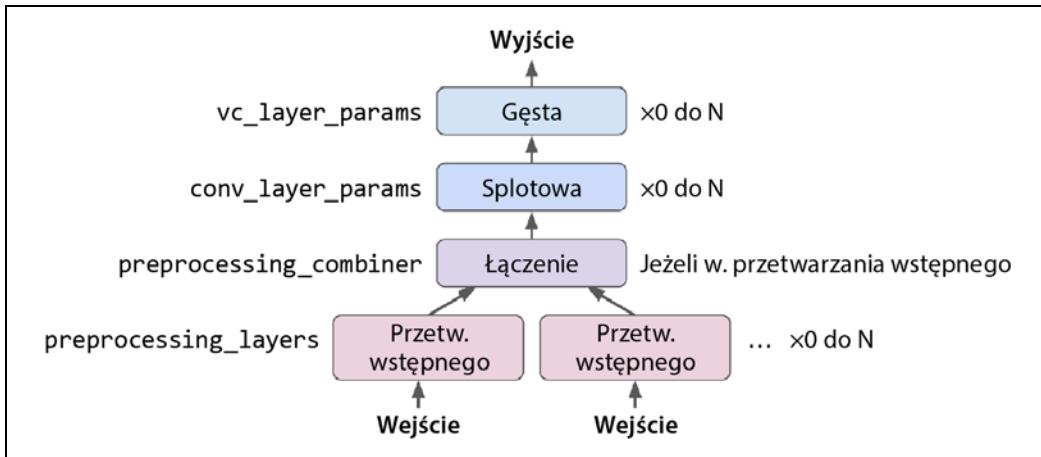
preprocessing_layer = keras.layers.Lambda(
    lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8, 4), (64, (4, 4), 2), (64, (3, 3), 1))]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

Ta klasa `QNetwork` przyjmuje na wejściu obserwację, a na wyjściu generuje po jednej Q-wartości na każdą czynność, musimy więc podać jej specyfikację obserwacji i czynności. Rozpoczyna się ona od warstwy wstępniego przetwarzania: prostej warstwy `Lambda`, która przetwarza obserwacje w 32-bitowe dane zmiennoprzecinkowe i normalizuje je (wartości będą mieściły się w zakresie od 0,0 do 1,0). Obserwacje zawierają bajty bezznakowe, które zajmują czterokrotnie mniej miejsca niż 32-bitowe wartości zmiennoprzecinkowe, co stanowi główną przyczynę tego, że nie przekształcaliśmy wcześniej obserwacji w 32-bitowe dane zmiennoprzecinkowe — chcemy zaoszczędzić pamięć operacyjną w buforze odtwarzania. Po warstwie przetwarzania wstępniego występują trzy warstwy splotowe: pierwsza zawiera 32 filtry o rozmiarach 8×8 i krok równy 4, druga ma 64 filtry o rozmiarach 4×4 i krok o wartości 2, trzecia natomiast przechowuje 64 filtry o rozmiarach 3×3 i krok równy 1. Na koniec wprowadzamy warstwę gęstą składającą się z 512 jednostek, a po niej cztero-jednostkową gęstą warstwę wyjściową, po jednej jednostce na każdą generowaną Q-wartość (tzn. po jednej Q-wartości na czynność). Oprócz warstwy wyjściowej wszystkie pozostałe warstwy splotowe i gęsta domyślnie wykorzystują funkcję aktywacji `ReLU` (możesz ją zmienić za pomocą argumentu `activation_fn`). Warstwa wyjściowa nie zawiera żadnej funkcji aktywacji.

Poza naszym wzrokiem klasa `QNetwork` składa się z dwóch części: sieci kodowania, która przetwarza obserwacje, i gęstej sieci wyjściowej, która generuje po jednej Q-wartości na każdą czynność. Klasa `EncodingNetwork` implementuje strukturę sieci neuronowej spotykaną w różnych agentach (rysunek 18.14).

Może ona zawierać więcej niż jedno wejście. Na przykład jeżeli obserwacja składa się z danych uzyskiwanych przez czujnik i z obrazu uzyskanego przez kamerę, będziesz mieć do dyspozycji dwa sygnały wejściowe. Każdy sygnał wejściowy może wymagać odmiennego przetwarzania wstępnego — w takim przypadku możesz wyznaczyć listę warstw Keras za pomocą argumentu



Rysunek 18.14. Architektura sieci kodowania

preprocessing_layers, po jednej warstwie na sygnał wejściowy, a sieć przeznaczy odpowiednią warstwę dla tego sygnału (jeżeli dane wejściowe wymagają wielu warstw wstępnego przetwarzania, możesz przekazać cały model, gdyż model Keras może zawsze zostać użyty jako warstwa). Jeżeli występują co najmniej dwa sygnały wejściowe, musisz także przekazać dodatkową warstwę za pomocą argumentu preprocessing_combiner po to, aby połączyć dane wyjściowe z warstw wstępnego przetwarzania w jeden sygnał wyjściowy.

Następnie sieć kodowania może ewentualnie wprowadzać sekwencyjnie listę warstw splotowych, pod warunkiem że wyznaczysz ich parametry za pomocą argumentu conv_layer_params. Lista ta musi składać się z trzykrotek (po jednej na każdą warstwę splotową) i określać liczbę filtrów, rozmiar jądra i krok. Po warstwach splotowych sieć kodowania może opcjonalnie wprowadzić sekwencję warstw gęstych, jeżeli zdefiniujesz argument fc_layer_params: musi to być lista definiująca liczbę neuronów w każdej warstwie gęstej. Dodatkowo możesz przekazać listę współczynników porzucania (po jednym na każdą warstwę gęstą) za pomocą argumentu dropout_layer_params, jeżeli zależy Ci na wprowadzeniu techniki porzucania po każdej warstwie gęstej. Sieć QNetwork przyjmuje dane wyjściowe z tej sieci kodowania i przekazuje je do gęstej warstwy wyjściowej (zawierającej po jednej jednostce na każdą czynność).



Klasa QNetwork jest wystarczająco elastyczna, abyśmy mogli dzięki niej tworzyć wiele różnych struktur, jeżeli jednak potrzebujesz jeszcze więcej możliwości, zawsze możesz utworzyć własną klasę niestandardową: rozszerz klasę tf_agents.networks.Network i zaimplementuj ją jak niestandardową warstwę Keras. Klasa tf_agents.networks.[→Network](#) stanowi podklasę klasy keras.layers.Layer zawierającą dodatkowe funkcje wymagane przez niektóre agenty, na przykład możliwość łatwego tworzenia płytkich kopii sieci (tzn. kopiowanie architektury sieci, ale z pominięciem wag). W ten sposób na przykład model DQNAgent tworzy kopię modelu trwającego.

Skoro mamy już gotową sieć DQN, możemy zająć się tworzeniem agenta.

Tworzenie agenta DQN

Biblioteka TF-Agents implementuje wiele typów agentów w pakietach `tf_agents.agents` i jej podpakietach. Skorzystamy z klasy `tf_agents.agents.dqn.dqn_agent.DqnAgent`:

```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

train_step = tf.Variable(0)
update_period = 4 # Trenuje model co cztery kroki
optimizer = keras.optimizers.RMSprop( lr=2.5e-4, rho=0.95, momentum=0.0,
                                      epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # Początkowa wartość ε
    decay_steps=250000 // update_period, # <=> 1 000 000 klatek ALE
    end_learning_rate=0.01) # Końcowa wartość ε
agent = DqnAgent(tf_env.time_step_spec(),
                  tf_env.action_spec(),
                  q_network=q_net,
                  optimizer=optimizer,
                  target_update_period=2000, # <=> 32 000 klatki ALE
                  td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                  gamma=0.99, # Stopa dyskontowa
                  train_step_counter=train_step,
                  epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()
```

Przeanalizujmy ten kod:

- Najpierw tworzymy zmienną zliczającą kroki uczenia.
- Następnie budujemy optymalizator i wykorzystujemy przy tym hiperparametry użyté w publikacji z 2015 roku poświęconej sieci DQN.
- Teraz tworzymy obiekt `PolynomialDecay`, który będzie obliczał wartość ϵ dla ϵ -zachłannej strategii zbierania dla bieżącego kroku uczenia (zazwyczaj jest on stosowany do zmniejszania wartości współczynnika uczenia, stąd nazwy argumentów, ale można za jego pomocą doprowadzić do zaniku dowolnej wartości). Wartość ta będzie maleć od 1,0 do 0,01 (zgodnie z parametrami użytymi w publikacji z 2015 roku) w trakcie jednego miliona klatek ALE, co odpowiada 250 000 kroków, ponieważ korzystamy z co czwartej klatki. Ponadto będzie uczyć agenta co cztery kroki (czyli co 16 klatek ALE), zatem wartość ϵ będzie w rzeczywistości zanikać przez ponad 62 500 kroków **uczenia**.
- Kolejnym etapem jest zbudowanie obiektu `DQNAgent`, któremu przekazujemy specyfikacje taktów i czynności, obiekt `QNetwork`, optymalizator, liczbę kroków uczących pomiędzy aktualizacjami modelu docelowego, funkcję straty, stopę dyskontową, zmienną `train_step`, a także funkcję zwracającą wartość ϵ (nie może ona przyjmować żadnych argumentów, dlatego do jej przekazania potrzebujemy zmiennej `lambda`).

Zwrócić uwagę, że funkcja straty musi zwracać wartość błędu dla każdego przykładu, a nie średni błąd, dlatego właśnie wyznaczamy `reduction="none"`.

- Na koniec inicjalizujemy agenta.

Czas zająć się buforem odtwarzania i zapisującym w nim trajektorie obserwatorem.

Tworzenie bufora odtwarzania i związanego z nim obserwatora

Biblioteka TF-Agents zawiera wiele różnych implementacji bufora odtwarzania w pakiecie `tf_agents.replay_buffers`. Część z nich została napisana wyłącznie w Pythonie (nazwy ich modułów zaczynają się przedrostkiem `py_`), inne zaś bazują na module TensorFlow (przedrostek `tf_`). Skorzystamy z klasy `TFUniformReplayBuffer`, umieszczonej w pakiecie `tf_agents.replay_buffers.tf_uniform_replay_buffer`. Uzyskujemy w ten sposób wydajną implementację bufora odtwarzania z losowaniem równomiernym²²:

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

Przeanalizujmy te argumenty:

`data_spec`

Specyfikacja danych, które będą przechowywane w buforze odtwarzania. Agent DQN wie, jak będą wyglądać zgromadzone dane, i udostępnia ich specyfikację poprzez argument `collect_data_spec`, zatem przekazujemy właśnie te informacje buforowi odtwarzania.

`batch_size`

Liczba trajektorii, które będą dodawane w każdym kroku. W naszym przypadku będzie to tylko jedna trajektoria, gdyż sterownik będzie realizował wyłącznie jedną czynność w każdym kroku. Gdybyśmy mieli do czynienia ze **środowiskiem wsadowym** (ang. *batched environment*), czyli środowiskiem pobierającym w każdym kroku grupę czynności i zwracającym grupę obserwacji, to sterownik musiałby zapisywać w każdym kroku grupę trajektorii. Wykorzystujemy bufor odtwarzania TensorFlow, więc musi on znać rozmiar obsługiwanej grupy (po to, aby móc zbudować graf obliczeniowy). Przykład środowiska wsadowego znajdziemy w klasie `ParallelPyEnvironment` (z pakietu `tf_agents.environments.parallel_py_environment`): przetwarza równolegle wiele środowisk w osobnych procesach (środowiska te mogą być różne pod warunkiem, że będą mieć takie specyfikacje czynności i obserwacji), a w każdym kroku przyjmuje grupę czynności i wykonuje je w poszczególnych środowiskach (po jednej czynności na środowisko), po czym zwraca wszystkie uzyskane obserwacje.

`max_length`

Maksymalny rozmiar bufora odtwarzania. Utworzyliśmy duży bufor, przechowujący do miliona trajektorii (tak samo jak w publikacji z 2015 roku). Z tego powodu będziemy potrzebować sporo pamięci RAM.



Gdy przechowujemy dwie następujące po sobie trajektorie, zawierają one dwie kolejne, czteroklatkowe obserwacje (ponieważ korzystamy z funkcji `FrameStack4`) i niestety trzy z czterech klatek w drugiej obserwacji są nadmiarowe (występują już w pierwszej

²² W czasie pisania książki nie istniała jeszcze implementacja bufora odtwarzania priorytetowych doświadczeń, ale przedżej czy później ktoś ją opublikuje.

obserwacji). Innymi słowy, obciążamy czterokrotnie więcej pamięci RAM, niż to potrzebne. Możemy uniknąć tego problemu, jeśli użyjemy bufora PyHashedReplayBuffer z pakietu `tf_agents.replay_buffers.py_hashed_replay_buffer` — deduplikuje on dane w przechowywanych trajektoriach wzduż ostatniej osi obserwacji.

Możemy teraz utworzyć obserwatora zapisującego trajektorie w buforze odtwarzania. Jest to zwykła funkcja (lub wywoływalny obiekt), która przyjmuje argument trajektorii, dlatego możemy bezpośrednio wykorzystać metodę `add_method()` (powiązaną z obiektem `replay_buffer`) jako naszego obserwatora:

```
replay_buffer_observer = replay_buffer.add_batch
```

Gdybyśmy chcieli utworzyć własnego obserwatora, moglibyśmy napisać dowolną funkcję zawierającą argument `trajectory`. Jeżeli musi ona zawierać stan, możesz napisać klasę zawierającą metodę `_call_(self, trajectory)`. Prosty obserwator z tego przykładu zwiększa wartość licznika po każdym wywołaniu (z wyjątkiem sytuacji, gdy trajektoria reprezentuje granicę pomiędzy dwoma epizodami, gdyż nie liczy się to jako krok) i co 100 przyrostów zostaje wyświetlony pasek postępów (dzięki powrotowi karetki `\r` wraz z zapisem `end=""` wyświetlany licznik pozostaje w tym samym wierszu):

```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Czas przygotować kilka wskaźników procesu uczenia.

Tworzenie wskaźników procesu uczenia

Biblioteka TF-Agents implementuje kilka wskaźników RL w pakiecie `tf_agents.metrics`: niektóre są napisane wyłącznie w Pythonie, inne bazują na module TensorFlow. Utwórzmy wskaźniki zliczające epizody i wykonane kroki oraz, co najważniejsze, obliczające średni zwrot na epizod i średnią długość trwania epizodu:

```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```



Zmniejszanie nagród ma sens w procesie uczenia lub implementowania strategii, ponieważ umożliwia ono równoważenie znaczenia nagród natychmiastowych i przyszłych. Jednak po zakończeniu epizodu możemy go ocenić poprzez zsumowanie nagród **nieobniżonych**. Z tego powodu klasa `AverageReturnMetric` oblicza w każdym epizodzie sumę nagród nieobniżonych i śledzi średnią tych sum ze wszystkich epizodów.

Możesz uzyskać w dowolnym momencie wartość każdego z tych wskaźników poprzez wywołanie ich metody `result()` (np. `train_metrics[0].result()`). Ewentualnie możesz zapisać wszystkie wskaźniki w dzienniku zdarzeń za pomocą funkcji `log_metrics(train_metrics)` (stanowi ona część pakietu `tf_agents.eval.metric_utils`):

```
>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.get_logger().set_level(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0
```

Utwórzmy teraz sterownik.

Tworzenie sterownika

Jak już wiesz z rysunku 18.13, sterownik jest obiektem, który poznaje środowisko za pomocą danej strategii, gromadzi doświadczenia i przekazuje je do niektórych obserwatorów. W każdym kroku dzieją się następujące zjawiska:

- Sterownik przekazuje bieżący takt do strategii zbierania, która wybiera czynność i zwraca obiekt **kroku czynności** (ang. *action step*) zawierającego tę czynność.
- Następnie sterownik przekazuje tę czynność do środowiska, które zwraca kolejny takt.
- Na koniec sterownik tworzy obiekt trajektorii reprezentujący to przejście i przesyła go do wszystkich obserwatorów.

Niektóre strategie, takie jak RNN, są stanowe: czynność jest w nich wybierana na podstawie zarówno danego taktu, jak i własnego stanu wewnętrznego. Strategie stanowe zwracają własny stan w ramach kroku czynności wraz z wybraną czynnością. Następnie sterownik w kolejnym taktie przekaże ten stan z powrotem do strategii. Co więcej, sterownik zapisuje ten stan strategii w trajektorii (w polu `policy_info`), zatem trafia on do bufora odśwarzania. Zjawisko to jest kluczowe w procesie uczenia strategii stanowej: gdy agent losuje trajektorię, musi wyznaczyć stan strategii na taki sam, jaki występował w wylosowanych takciach.

Jak już również wiesz, środowisko może być wsadowe, co oznacza, że sterownik przekazuje strategii **takt wsadowy** (ang. *batched time step*; jest to obiekt taktu zawierający grupę obserwacji, grupę typów kroków, grupę nagród i grupę wartości dyskontowych, gdzie wszystkie grupy mają taki sam rozmiar). Sterownik przekazuje także grupę poprzednich stanów strategii. Strategia następnie zwraca

wsadowy krok czynności, przechowujący grupę czynności i grupę stanów strategii. Na koniec sterownik tworzy **trajektorię wsadową** (czyli trajektorię zawierającą grupę typów kroków, grupę obserwacji, grupę czynności, grupę nagród i, ogólniej, grupę każdego atrybutu trajektorii, gdzie wszystkie grupy mają taki sam rozmiar).

Istnieją dwie główne klasy sterowników: `DynamicStepDriver` i `DynamicEpisodeDriver`. Pierwsza gromadzi doświadczenia przez określoną liczbę kroków, druga zbiera je przez wyznaczoną liczbę epizodów. Chcemy gromadzić doświadczenia dla czterech kroków w każdej iteracji uczenia (zgodnie z konfiguracją opisaną w publikacji z 2015 roku), dlatego utworzymy klasę `DynamicStepDriver`:

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # Gromadzi cztery kroki dla każdej iteracji uczenia
```

Przekazujemy sterownikowi środowisko, strategię zbierania, listę obserwatorów (w tym obserwatora wyznaczonego do bufora odtwarzania i wskaźnika uczenia), a także liczbę kroków, jakie mają zostać wykonane (w tym przypadku cztery). Moglibyśmy teraz uruchomić sterownik za pomocą metody `run()`, lepiej jednak przygotować bufor odtwarzania za pomocą doświadczeń zgromadzonych przy użyciu całkiem losowej strategii. W tym celu możemy wykorzystać klasę `RandomTFPolicy` i utworzyć drugi sterownik, który będzie eksplotował tę strategię przez 20 000 kroków (jest to równoważne 80 000 klatek symulatora, zgodnie z wytycznymi publikacji z 2015 roku). Możemy wyświetlać postępy za pomocą naszego obserwatora `ShowProgress`:

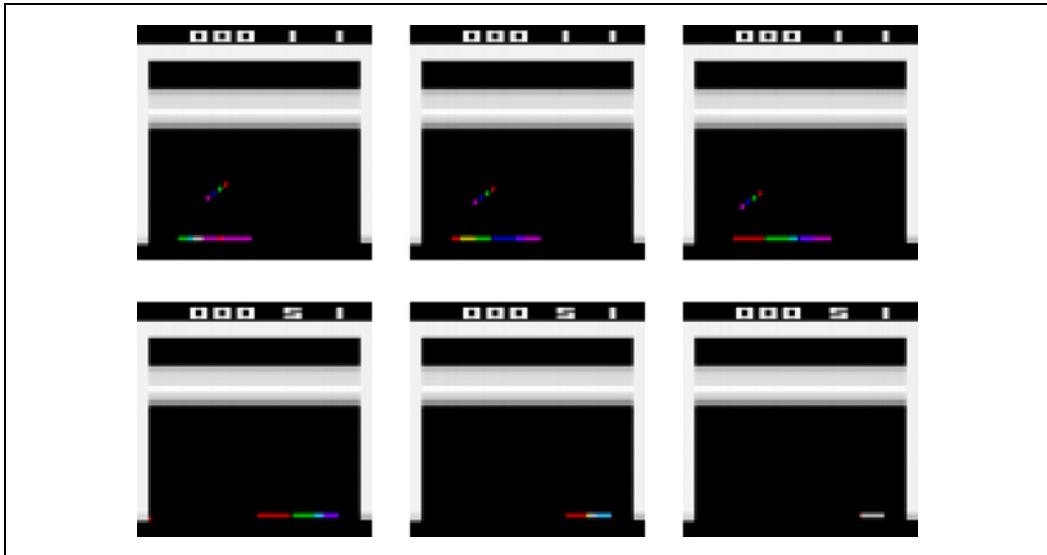
```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                         tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80 000 klatek ALE
final_time_step, final_policy_state = init_driver.run()
```

Jesteśmy już niemal gotowi do uruchomienia pętli uczenia! Potrzebujemy jeszcze tylko jednego składnika: zestawu danych.

Tworzenie zestawu danych

Aby wylosować grupę trajektorii z bufora odtwarzania, wywołaj jego metodę `get_next()`. Zostanie zwrócona grupa trajektorii, a także obiekt `BufferInfo`, zawierający identyfikatory przykładów i prawdopodobieństwa ich losowania (dane te przydają się w pewnych algorytmach, np. PER). Na przykład kod w tym listingu będzie losował małą grupę składającą się z dwóch trajektorii (podepizodów), z których każda określa trzy kolejne kroki (rysunek 18.15) (każdy wiersz zawiera trzy kolejne kroki z epizodu):

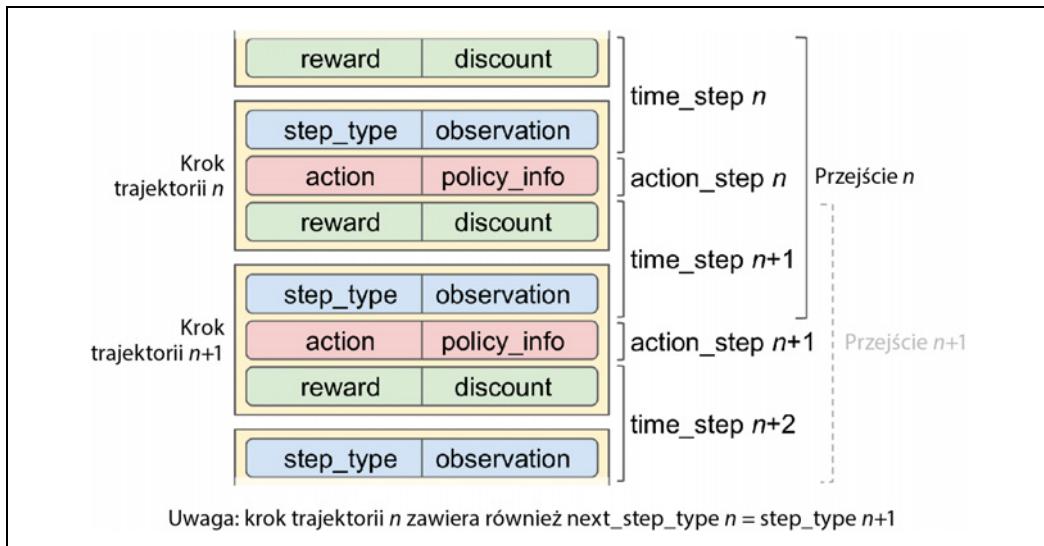


Rysunek 18.15. Dwie trajektorie zawierające po trzy kolejne kroki

```
>>> trajectories, buffer_info = replay_buffer.get_next(
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
 'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

Obiekt `trajectories` jest krotką nazwaną, zawierającą siedem pól. Każde pole przechowuje tensor, którego dwa pierwsze wymiary to 2 i 3 (gdyż mamy do czynienia z dwiema trajektoriami zawierającymi po trzy kroki). Teraz już wiemy, dlaczego pole `observation` ma wymiary [2, 3, 84, 84, 4]: określa dwie trajektorie zawierające po trzy kroki, a obserwacja każdego kroku ma rozmiar $84 \times 84 \times 4$. Podobnie w przypadku tensora `step_type` o wymiarach [2, 3]: w tym przykładzie obydwie trajektorie zawierają po trzy kolejne kroki ze środka epizodu (typy 1, 1, 1). W drugiej trajektorii możemy z trudnością zauważać piłeczkę w lewym dolnym rogu ekranu, a w dwóch kolejnych obserwacjach już wcale jej nie widać, co oznacza, że agent straci życie, ale epizod nie zostanie zakończony, ponieważ agentowi pozostało jeszcze kilka życia.

Każda trajektoria stanowi zwięzłą reprezentację sekwencji kolejnych taktów i kroków czynności, zaprojektowaną tak, aby uniknąć nadmiarowości. W jaki sposób? Jak widać na rysunku 18.16, przejście n składa się z taktu n , kroku czynności n oraz taktu $n+1$, natomiast przejście $n+1$ składa się taktu $n+1$, kroku czynności $n+1$ i taktu $n+2$. Gdybyśmy przechowywali te dwa przejścia bezpośrednio w buforze odtwarzania, to takt $n+1$ byłby powielony. Aby uniknąć tego powielenia, n -ty krok trajektorii zawiera wyłącznie typ i obserwację z taktu n (bez nagrody i wartości dyskontowej) oraz nie zawiera obserwacji z taktu $n+1$ (przechowuje jednak kopię typu z następnego taktu — jest to jedyne powielenie).



Rysunek 18.16. Trajektorie, przejścia, takty i kroki czynności

Jeśli więc dysponujesz grupą trajektorii, w której każda trajektoria zawiera $t+1$ kroków (od taktu n do taktu $n+t$), to grupa ta przechowuje niemal wszystkie dane od taktu n do taktu $n+t$, oprócz nagrody i wartości dyskontowej z taktu n (zawiera natomiast nagrodę i wartość dyskontową z taktu $n+t+1$). W ten sposób reprezentowanych jest t przejść (od n do $n+1$, od $n+1$ do $n+2$, ..., od $n+t-1$ do $n+t$).

Funkcja `to_transition()` w module `tf_agents.trajectories.trajectory` przekształca trajektorię wsadową w listę zawierającą wsadowy obiekt `time_step`, wsadowy obiekt `action_step` i wsadowy obiekt `next_time_step`. Zwrót uwagę, że drugi wymiar to teraz 2, a nie 3, gdyż występuje t przejść pomiędzy $t+1$ taktów (nie przejmuj się, jeżeli czujesz się nieco zdezorientowana/zdezorientowany — zdążysz jeszcze to zrozumieć):

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # Trzy takty = Dwa przejścia
```



Wylosowana trajektoria może w rzeczywistości nachodzić na dwa (lub więcej) epizody! W takim przypadku będzie zawierała **przejścia graniczne** (ang. *boundary transitions*), czyli przejścia, w których `step_type` jest równy 2 (koniec), a `next_step_type` jest równy 0 (początek). Oczywiście biblioteka TF-Agents we właściwy sposób przetwarza te trajektorie (np. zeruje stan strategii w momencie natrafienia na granicę). Metoda `is_boundary()` trajektorii zwraca tensor wskazujący, czy któryś z kroków jest graniczny.

W naszej głównej pętli uczenia nie będziemy wywoływać metody `get_next()`, lecz użyjemy `tf.data.Dataset`. Dzięki temu będziemy mogli wykorzystać możliwości interfejsu danych (m.in. zrównoleglane i wstępne wczytywanie). W tym celu wywołamy metodę `as_dataset()` bufora odtwarzania:

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

Będziemy losować w każdym kroku uczącym grupy składające się z 64 trajektorii (tak samo jak w artykule z 2015 roku), gdzie każda trajektoria będzie zawierała dwa kroki (dwa kroki = jedno pełne przejście, włącznie z obserwacją następnego kroku). Nasz obiekt dataset będzie przetwarzał równolegle trzy elementy i wstępnie wczytywał trzy grupy.



W przypadku algorytmów ze strategią, takich jak algorytm gradientów strategii, każde doświadczenie powinno być losowane tylko raz, użyte w trakcie uczenia, a następnie odrzucone. W takim przypadku możesz nadal korzystać z bufora odtwarzania, ale zamiast używać obiektu Dataset należy wywoływać metodę `gather_all()` bufora odtwarzania w każdej iteracji uczenia po to, aby uzyskać tensor zawierający wszystkie zarejestrowane dotychczas trajektorie, których następnie można użyć podczas uczenia, a na koniec trzeba oczyścić bufor za pomocą metody `clear()`.

Mamy już wszystkie potrzebne elementy i możemy zająć się uczeniem modelu!

Tworzenie pętli uczenia

Aby przyspieszyć uczenie, przekształcimy główne funkcje w funkcje TensorFlow. Użyjemy w tym celu funkcji `tf_agents.utils.common.function()`, która otacza funkcję `tf.function()` i zapewnia pewne eksperymentalne możliwości:

```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

Utwórzmy małą funkcję, która będzie realizowała główną pętlę uczenia przez `n_iterations`:

```
def train_agent(n_iterations):
    time_step = None
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} F. straty:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

Funkcja sprawdza najpierw stan początkowy strategii zbierania (dla danego rozmiaru grupy środowisk, który w naszym przypadku wynosi 1). Strategia jest bezstanowa, zatem zwraca pustą krotkę (dlatego moglibyśmy napisać `policy_state = ()`). Następnie tworzymy iterator zestawu danych i uruchamiamy pętlę uczenia. W każdej iteracji wywołujemy metodę `run()` sterownika i przekazujemy jej bieżący takt (początkowo o wartości `None`) i bieżący stan strategii. W ten sposób zostanie użyta strategia zbierania i będą gromadzone doświadczenia z czterech kroków (zgodnie z przygotowaną wcześniej konfiguracją), a zgromadzone trajektorie zostaną przesłane do bufora odtwarzania

i wskaźników. Potem losujemy jedną grupę trajektorii z zestawu danych i przekazujemy ją do metody `train()` agenta. Zwraca ona obiekt `train_loss`, którego rodzaj jest uzależniony od typu używanego agenta. W dalszej kolejności wyświetlamy numer iteracji i wartość funkcji straty uczenia, a co tysiąc iteracji zostają zaprezentowane wszystkie wskaźniki. Teraz możesz wywoływać metodę `train_agent()` przez pewną liczbę iteracji i przekonasz się, że agent stopniowo uczy się grać w Breakout!

```
train_agent(10000000)
```

Proces ten jest bardzo kosztowny obliczeniowo i wymaga mnóstwa cierpliwości (w zależności od sprzętu może trwać godzinami, a nawet całymi dniami), a do tego być może trzeba będzie uruchamiać algorytm kilka razy z wyznaczonymi różnymi załączkami losowości, aby uzyskać dobre rezultaty, ale po zakończeniu treningu agent okaże się nadczłowiekiem (przynajmniej w grze Breakout). Możesz również spróbować nauczyć tego agenta DQN przechodzić inne gry na Atari: może uzyskiwać nadzwyczajne wyniki w większości gier akcji, ale nie radzi sobie zbyt dobrze w grach z bardziej rozbudowaną fabułą²³.

Przegląd popularnych algorytmów RN

Zanim przejdziemy do następnego rozdziału, przyjrzymy się kilku popularnym algorytmom uczenia przez wzmacnianie:

Algorytmy typu aktor – krytyk

Rodzina algorytmów RL łączących algorytm gradientów strategii z Q-sieciemi głębokimi. Agent typu aktor – krytyk zawiera dwie sieci neuronowe: sieć strategii i sieć DQN. Sieć DQN jest uczona tradycyjnie za pomocą doświadczeń agenta. Sieć strategii uczy się inaczej (i znacznie szybciej) w porównaniu do klasycznej sieci gradientów strategii: nie szacujemy tu wartości każdej czynności poprzez realizację wielu epizodów, następnie sumowanie przyszłych nagród dyskontowych dla każdej czynności i ich normalizację, lecz agent (aktor) jest zależny od wartości czynności szacowanych przez sieć DQN (krytyka). Rozwiązywanie to przypomina nieco sportowca (agenta) uczącego się z pomocą trenera (sieci DQN).

Algorytm A3C (ang. *Asynchronous Advantage Actor-Critic* — asynchroniczny algorytm typu aktor – krytyk oparty na korzyściach; <https://arxiv.org/abs/1602.01783>)²⁴

Istotny wariant modelu aktor – krytyk zaproponowany przez zespół DeepMind w 2016 roku. Wiele agentów uczy się równolegle i poznaje różne kopie środowiska. W regularnych odstępach (ale asynchronicznie, stąd nazwa) każdy agent przesyła aktualizacje niektórych wag do sieci nadzędnej i pobiera z niej najnowsze wagi. Zatem każdy agent wnosi wkład w usprawnianie sieci nadzędnej i korzysta z wiedzy zdobytej przez inne agenty. Co więcej, sieć DQN nie szacuje Q-wartości, lecz oszacowuje korzyść z każdej czynności (stąd druga litera A w angielskiej nazwie modelu), co dodatkowo stabilizuje proces uczenia.

²³ Porównanie wydajności tego algorytmu w różnych grach na Atari znajdziesz na rysunku 3. w artykule z 2015 roku (<https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>).

²⁴ Volodymyr Mnih i in., *Asynchronous Methods for Deep Reinforcement Learning*, „Proceedings of the 33rd International Conference on Machine Learning” (2016), s. 1928 – 1937.

Algorytm A2C (ang. *Advantage Actor-Critic* — algorytm typu aktor – krytyk oparty na korzyściach; <https://openai.com/blog/baselines-acktr-a2c/>)

Odmiana algorytmu A3C, w której usunięto asynchronicznosć. Wszystkie aktualizacje modelu są synchroniczne, zatem aktualizacje gradientów są przeprowadzane na większych grupach, przez co model może lepiej wykorzystywać moc procesorów graficznych.

Algorytm SAC (ang. *Soft Actor-Critic* — miękki algorytm typu aktor – krytyk; <https://arxiv.org/abs/1801.01290>)²⁵

Wariant modelu typu aktor – krytyk zaproponowany w 2018 roku przez Tuomasa Haarnoę i innych naukowców z Uniwersytetu Kalifornijskiego w Berkeley. Uczy się on nie tylko nagród, lecz także maksymalizowania entropii w realizowanych czynnościach. Innymi słowy, stara się być maksymalnie nieprzewidywalny, a jednocześnie uzyskiwać jak największe nagrody. Motywuje to agenta do eksplorowania środowiska, co przyspiesza proces uczenia i zmniejsza ryzyko utknięcia na tej samej czynności w przypadku, gdy sieć DQN wygeneruje niedoskonałe oszacowania. Algorytm ten wykazuje zdumiewającą skuteczność próbowania (w przeciwieństwie do wcześniej omawianych algorytmów, które uczą się bardzo powoli). Algorytm SAC jest dostępny w bibliotece TF-Agents.

Algorytm PPO (ang. *Proximal Policy Optimization* — proksymalna optymalizacja strategii; <https://arxiv.org/abs/1707.06347>)²⁶

Oparty na modelu A2C algorytm, w którym wartości funkcji straty są obcinane w celu zapobiegania nadmiernym aktualizacjom wag (które często prowadzą do niestabilnego procesu uczenia). Algorytm PPO stanowi uproszczoną wersję wcześniejszego algorytmu TRPO (ang. *Trust Region Policy Optimization* — optymalizacja strategii obszaru zaufania; <https://arxiv.org/abs/1502.05477>)²⁷, którego autorem jest także John Schulman wraz z innymi naukowcami z zespołu OpenAI. Pierwszy raz usłyszeliśmy o algorytmie PPO w kwietniu 2019 roku, gdy zespół OpenAI stworzył na jego podstawie sztuczną inteligencję nazwaną OpenAI Five, która pokonała mistrzów świata w grę sieciową Dota 2. Algorytm PPO również jest dostępny w bibliotece TF-Agents.

Poszukiwanie oparte na ciekawości (ang. *curiosity-based exploration*; <https://arxiv.org/abs/1705.05363>)²⁸

Powracającym problem w technikach uczenia przez wzmacnianie jest rzadkość występowania nagród, przez co proces uczenia jest bardzo żmudny i niewydajny. Deepak Pathak wraz z innymi uczonymi z Uniwersytetu Kalifornijskiego w Berkeley zaproponował bardzo interesujący sposób rozwiązania tego problemu: a może po prostu zignorować nagrody i sprawić, żeby agent chciał z olbrzymią ciekawością poznawać środowisko? W ten sposób nagrody stają się

²⁵ Tuomas Haarnoja i in., *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*, „Proceedings of the 35th International Conference on Machine Learning” (2018), s. 1856 – 1865.

²⁶ John Schulman i in., *Proximal Policy Optimization Algorithms*, arXiv preprint arXiv:1707.06347 (2017).

²⁷ John Schulman i in., *Trust Region Policy Optimization*, „Proceedings of the 32nd International Conference on Machine Learning” (2015), s. 1889 – 1897.

²⁸ Deepak Pathak i in., *Curiosity-Driven Exploration by Self-Supervised Prediction*, „Proceedings of the 34th International Conference on Machine Learning” (2017), s. 2778 – 2787.

wewnętrznym aspektem agenta, a nie pochodzą ze środowiska. Podobnie w przypadku dziecka: czasami wywołanie ciekawości przynosi lepsze efekty niż nagradzanie go za dobre oceny. Jak wygląda mechanizm działania tego algorytmu? Agent bez przerwy stara się przewidywać wyniki swoich czynności i szuka sytuacji, w których rezultat okazuje się niezgodny z prognozami. Innymi słowy, agent chce zostać zaskoczony. Jeżeli wynik będzie przewidywalny (nudny), to agent przejdzie do innego stanu. Jeśli jednak rezultat okaże się nieprzewidywalny, ale agent nie ma nad nim żadnej kontroli, to także znudzi się po pewnym czasie. Wyłącznie za pomocą mechanizmu ciekawości autorom udało się wytrenować agenta w przechodzeniu wielu gier wideo — nawet jeżeli agent nie otrzymuje kary za przegrana, rozpoczyna grę od nowa, co jest nudne, zatem stara się unikać porażek.

W niniejszym rozdziale omówię mnóstwo zagadnień: gradienty strategii, łańcuchy Markowa, procesy decyzyjne Markowa, Q-uczenie, przybliżone Q-uczenie oraz Q-uczenie głębokie wraz z jego odmianami (ustalone Q-wartości docelowe, podwójna sieć DQN, walcząca sieć DQN i odtwarzanie priorytetowych doświadczeń). Umiesz już wykorzystywać bibliotekę TF-Agents do uczenia agentów na dużą skalę. Przyjrzałeś się też kilku popularnym algorytmom. Dziedzina uczenia przez wzmacnianie jest bardzo rozbudowana i fascynująca, a do tego codziennie pojawiają się nowe koncepcje i algorytmy, mam więc nadzieję, że rozbudziłem Twoją ciekawość.

Ćwiczenia

1. Zdefiniuj pojęcie uczenia przez wzmacnianie. Czym się ono różni od klasycznych metod uczenia nadzorowanego i nienadzorowanego?
2. Wymień trzy zastosowania uczenia przez wzmacnianie niewymienione w tym rozdziale. Czym byłoby środowisko dla każdego z tych zastosowań? Czym jest agent? Jakie są możliwe czynności? Czym są nagrody?
3. Co to jest stopa dyskontowa? Czy po jej zmodyfikowaniu może ulec zmianie optymalna strategia?
4. W jaki sposób mierzmy wydajność agenta?
5. Na czym polega problem przypisania zasługi? Kiedy mamy z nim do czynienia? Jak możemy złagodzić jego skutki?
6. Jakie jest zadanie pamięci odtwarzania?
7. Czym jest algorytm bez strategii uczenia przez wzmacnianie?
8. Skorzystaj z algorytmu gradientów strategii, aby rozwiązać problem zaprezentowany w środowisku *LunarLander-v2* z narzędzia OpenAI Gym. Musisz zainstalować zależności Box2D (`python3 -m pip install -U gym[box2d]`).
9. Użyj biblioteki TF-Agents, aby za pomocą dowolnego algorytmu wytrenować agenta wykazującego ponadludzkie zdolności w grze *SpaceInvaders-v4*.
10. Jeśli jesteś w stanie wydać ok. 300 zł, warto zaopatrzyć się w platformę sprzętową Raspberry Pi 3 i tanie podzespoły, zainstalować na niej moduł TensorFlow i puścić wodze wyobraźni! Za

przykład niech posłuży świetny wpis (<https://www.oreilly.com/learning/how-to-build-a-robot-that-sees-with-100-and-tensorflow>) Lukasa Biewalda albo takie projekty jak GoPiGo lub BrickPi. Začnij od czegoś prostego, na przykład robota obracającego się w poszukiwania naijaśniejszego punktu (jeżeli ma czujnik światła) lub najbliższego obiektu (jeżeli zawiera sonar) i poruszającego się w tym kierunku. Następnie możesz przejść do uczenia głębokiego: przykładowo, jeżeli robot ma zamontowaną kamerę, możesz zaimplementować algorytm wykrywania obiektów, dzięki któremu będzie w stanie rozpoznawać ludzi i kierować się w ich stronę. Możesz też wykorzystać uczenie przez wzmacnianie tak, aby robot w celu wykonania tego zadania nauczył się samodzielnie sterować silniczkami. Miłej zabawy!

Rozwiązań tych zadań znajdziesz w dodatku A.

Wielkoskalowe uczenie i wdrażanie modeli TensorFlow

Co robimy, gdy uda nam się już utworzyć cudowny model uzyskujący zdumiewające prognozy? Wiadomo, umieszczamy go w środowisku produkcyjnym! Proces ten może ograniczać się do przetwarzania grupy danych przez model i być może napisania skryptu uruchamiającego ten model co noc. Często jednak mamy do czynienia ze znacznie bardziej skomplikowanymi sytuacjami. Wiele elementów infrastruktury może wymagać dostępu do bieżących danych, co najprawdopodobniej będzie oznaczało konieczność umieszczenia modelu w usłudze sieciowej — w ten sposób dowolna część infrastruktury może przepłytywać model w dowolnym momencie za pomocą prostego interfejsu API REST (lub innego protokołu), o czym wiesz już z rozdziału 2. Jednak w miarę upływu czasu musisz regularnie trenować model na świeżych danych i przesyłać zaktualizowaną wersję do środowiska produkcyjnego. Musisz zająć się wersjonowaniem modelu, płynnym przechodzeniem pomiędzy modelami, w razie problemów przywracaniem wcześniejszych wersji, a być może także równoległym przetwarzaniem wielu modeli w celu przeprowadzania eksperymentów A/B¹. Jeżeli Twój produkt odniesie sukces, usługa może zacząć otrzymywać mnóstwo kwerend na sekundę (ang. *queries per second* — QPS) i musi dostosowywać się do rosnącego obciążenia. Jak się wkrótce przekonasz, znakomitym rozwiązaniem jest system TF Serving, zainstalowany na własnym komputerze albo wykorzystujący usługę rozproszoną, taką jak Google Cloud AI Platform. Zajmie się on wydajnym eksplotowaniem modelu, płynnym przechodzeniem pomiędzy modelami itd. Jeśli korzystasz z usług rozproszonych, uzyskasz dostęp do wielu dodatkowych funkcji, m.in. potężnych narzędzi monitorowania.

Ponadto jeśli dysponujesz znaczną ilością danych uczących i kosztownymi obliczeniowo modelami, trenowanie modelu może trwać koszmarnie długo. Jeżeli Twój produkt musi szybko dostosować się do zmian, to długi czas uczenia jest niedopuszczalny (wyobraź sobie system rekomendacji wiadomości proponujący informacje sprzed tygodnia). Być może jeszcze ważniejszy jest fakt, że długi czas uczenia uniemożliwi Ci eksperymentowanie z nowymi pomysłami. W uczeniu maszynowym (podobnie jak w wielu innych dziedzinach) trudno określić z góry, które pomysły się sprawdzą, dlatego należy wypróbować ich jak najwięcej w jak najkrótszym czasie. Jednym ze sposobów

¹ Eksperyment A/B polega na testowaniu dwóch różnych wersji produktu na różnych grupach użytkowników po to, aby dowiedzieć się, która wersja działa najlepiej, oraz uzyskać inne opinie.

przyspieszenia procesu uczenia jest użycie akceleratorów sprzętowych, takich jak procesory graficzne lub tensorowe. To jednak nie koniec naszych możliwości, ponieważ możemy trenować model na wielu urządzeniach jednocześnie, z których każde jest wyposażone w wiele akceleratorów sprzętowych. Prosty, ale potężny interfejs strategii rozpraszania (ang. *Distribution Strategies API*) stanowiący część modułu TensorFlow znacznie ułatwia sprawę, o czym przekonasz się już niebawem.

W tym rozdziale nauczysz się wdrażać modele, najpierw do systemu TF Serving, a następnie do usługi Google Cloud AI Platform. Przyjrzyj się wdrażaniu modeli do aplikacji mobilnych, urządzeń wbudowanych i aplikacji sieciowych. Na koniec dowiesz się, w jaki sposób przyspieszyć obliczenia za pomocą jednostek GPU, a także jak używać interfejsu strategii rozpraszania do trenowania modeli na wielu urządzeniach i serwerach. Przed nami sporo tematów do omówienia, nie traćmy więc czasu!

Eksplatacja modelu TensorFlow

Po wytrenowaniu modelu TensorFlow możesz łatwo wykorzystywać go w dowolnym kodzie Pythona: jeżeli jest to model `tf.keras`, wystarczy wywołać jego metodę `predict()`! Jednak w miarę rozwoju infrastruktury zbliża się moment, kiedy bardziej opłaca się umieścić model w małej usłudze, której jedynym zadaniem jest uzyskiwanie prognoz dostępnych dla pozostałych elementów infrastruktury (np. poprzez interfejs REST lub gRPC)². Rozwiążanie to oddziela model od reszty infrastruktury, dzięki czemu możliwe staje się łatwe zmienianie wersji modelu lub skalowanie usługi w razie potrzeby (niezależnie od pozostały części infrastruktury) oraz przeprowadzanie eksperymentów A/B, a ponadto sprawia, że wszystkie elementy oprogramowania bazują na tej samej wersji modelu. Rozwiążanie to również upraszcza proces testowania, projektowania i nie tylko. Możesz utworzyć własną mikrosługę za pomocą dowolnej technologii (np. biblioteki Flask), ale po co odkrywać ponownie Amerykę, skoro można po prostu skorzystać z systemu TF Serving?

Korzystanie z systemu TensorFlow Serving

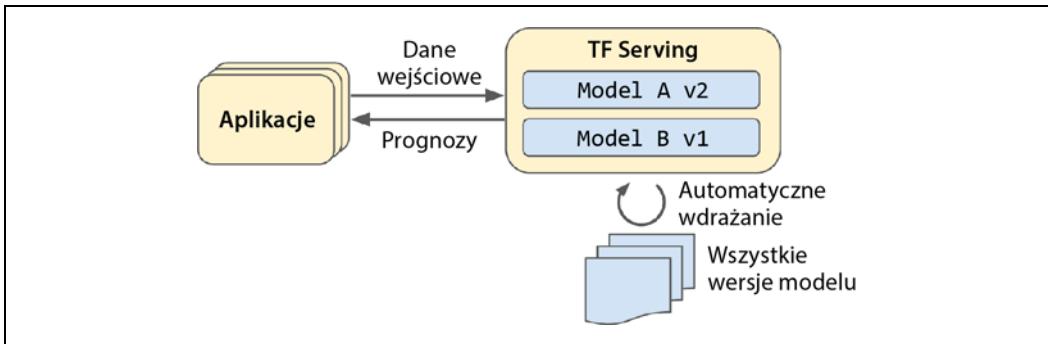
System TF Server to napisany w języku C++ bardzo wydajny i dobrze przetestowany serwer modeli. Radzi sobie z dużymi obciążeniami, eksplatuje wiele wersji modeli, pilnuje, aby repozytorium modeli automatycznie wdrażało najnowsze wersje itd. (rysunek 19.1).

Załóżmy więc, że wytrenowaliśmy model MNIST za pomocą interfejsu `tf.keras` i chcemy wdrożyć go do serwera TF Serving. Najpierw musimy eksportować ten model do **formatu SavedModel**.

Eksportowanie obiektów SavedModel

Moduł TensorFlow zawiera prostą funkcję `tf.saved_model.save()`, która eksportuje modele do formatu SavedModel. Wystarczy podać jej model (nazwę i numer wersji), a zapisze ona graf obliczeniowy i wagi modelu:

² Interfejs REST (lub RESTful) wykorzystuje standardowe czasowniki protokołu HTTP, takie jak GET, POST, PUT i DELETE, a także dane wejściowe i wyjściowe w formacie JSON. Protokół gRPC jest bardziej skomplikowany, ale wydajniejszy. Dane są wymieniane poprzez buforey protokołu (zob. rozdział 13.).



Rysunek 19.1. System TF Serving może eksploatować wiele modeli i automatycznie wdraża najnowszą wersję każdego modelu

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "moj_model_mnist"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Możesz ewentualnie wykorzystać metodę `save()` modelu (`model.save(model_path)`): dopóki rozszerzeniem pliku nie jest `.h5`, model zostanie zachowany w formacie `SavedModel`, a nie `HDF5`.

Zazwyczaj dobrym pomysłem jest umieszczenie wszystkich warstw wstępnego przetwarzania w eksportowanym modelu końcowym, dzięki czemu możesz dostarczać mu nieprzetworzone dane po wdrożeniu do środowiska produkcyjnego. Unikasz w ten sposób konieczności osobnego przetwarzania wstępnego w aplikacji wykorzystującej model. Połączenie fazy wstępnego przetwarzania z modelem ułatwia również jej późniejsze aktualizowanie i zmniejsza ryzyko pomiędzy niedopasowaniem modelu a wymaganymi czynnościami wstępnego przetwarzania.



W formacie `SavedModel` zostaje zapisany graf obliczeniowy, dlatego może być on używany tylko do modeli opartych wyłącznie na operacjach TensorFlow, nie licząc operacji `tf.py_function()` (w której umieszczamy dowolny kod Pythona). Zostają wykluczone również modele dynamiczne `tf.keras` (zob. dodatek G), ponieważ nie można przekształcić ich w grafy obliczeniowe. Modele dynamiczne muszą być eksploatowane za pomocą innych narzędzi (np. biblioteki `Flask`).

Format `SavedModel` reprezentuje wersję modelu. Jest on przechowywany w katalogu zawierającym plik `saved_model.pb`, w którym mieści się zdefiniowany graf obliczeniowy (w postaci serializowanego bufora protokołu), a także w podkatalogu `variables`, w którym znajdują się wartości zmiennych. W przypadku modeli zawierających dużą liczbę wag wartości zmiennych mogą być rozdzielone pomiędzy wiele plików. Format `SavedModel` zawiera też podkatalog `assets`, w którym mogą występować dodatkowe dane, takie jak pliki słownika, nazwy klas lub przykładowe próbki. Struktura katalogu tego formatu wygląda następująco (w omawianym przykładzie nie wykorzystuje dodatkowych elementów wstawianych do podkatalogu `assets`):

```

moj_model_mnist
0001
  assets
    saved_model.pb
  variables
    variables.data-00000-of-00001
    variables.index

```

Jak łatwo przewidzieć, można wczytać format SavedModel za pomocą funkcji `tf.saved_model.load()`. Zwrócony obiekt nie będzie jednak obiektem Keras: reprezentuje on format SavedModel, włącznie z grafem obliczeniowym i wartościami zmiennych. Możesz użyć go jako funkcji i będziesz uzyskiwać za jego pomocą prognozy (pamiętaj, żeby przekazywać dane wejściowe jako tenory odpowiedniego typu):

```

saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(tf.constant(X_new, dtype=tf.float32))

```

Ewentualnie możesz wczytać format SavedModel bezpośrednio do modelu Keras za pomocą funkcji `keras.models.load_model()`:

```

model = keras.models.load_model(model_path)
y_pred = model.predict(tf.constant(X_new, dtype=tf.float32))

```

Moduł TensorFlow zawiera także narzędzie wiersza polecenia `saved_model_cli` służące do badania obiektów SavedModel:

```

$ export ML_PATH="$HOME/um" # Ścieżka docelowa projektu
$ cd $ML_PATH
$ saved_model_cli show --dir moj_model_mnist/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['__saved_model_init_op']:
[...]
signature_def['serving_default']:
The given SavedModel SignatureDef contains the following input(s):
  inputs['flatten_input'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 28, 28)
    name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_1'] tensor_info:
    dtype: DT_FLOAT
    shape: (-1, 10)
    name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict

```

Format SavedModel zawiera przynajmniej jeden **metagraf**. Jest to połączenie grafu obliczeniowego z pewnymi definicjami sygnatur funkcji (w tym nazwami ich wejść i wyjść, typami i wymiarami). Każdy metagraf jest określany przez zestaw znaczników. Możesz na przykład potrzebować metagrafu zawierającego pełen graf obliczeniowy wraz z operacjami uczenia (może być on oznakowany jako np. "train"), a także drugiego metagrafu, który przechowuje przyjęty graf obliczeniowy definiujący jedynie operacje predykci oraz pewne operacje specyficzne dla procesorów graficznych (może być on oznakowany jako "serve", "gpu"). Jednak gdy przekazujesz model `tf.keras` do funkcji `tf.saved_model.save()`, domyślnie zapisuje ona znacznie prostszy obiekt SavedModel: pojedynczy metagraf o nazwie "serve" z dwiema definicjami sygnatur, funkcję inicjalizującą (`__saved_model_init_op`; nie musisz się

nią przejmować) i domyślną funkcję eksplorującą (`serving_default`). W czasie zapisywania modelu `tf.keras` domyślna funkcja eksplorująca jest równoważna funkcji `call()` modelu, która, co oczywiste, odpowiedzialna jest za uzyskiwanie prognoz.

Narzędzie `saved_model_cli` może również posłużyć do uzyskiwania prognoz (podczas testowania, nie podczas właściwych zadań produkcyjnych). Założymy, że dysponujemy tablicą NumPy (`X_new`) zawierającą trzy obrazy pisanych odręcznie cyfr, dla której chcesz uzyskać prognozy. Musisz najpierw eksportować je do formatu `npy`:

```
np.save("moje_testy_mnist.npy", X_new)
```

Teraz możesz zastosować polecenie `saved_model_cli` w następujący sposób:

```
$ saved_model_cli run --dir moj_model_mnist/0001 --tag_set serve \
                     --signature_def serving_default \
                     --inputs flatten_input=moje_testy_mnist.npy
[...] Result for output key dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

Uzyskujemy 10 prawdopodobieństw przynależności do klasy dla każdego z trzech przykładów. Świetnie! Mamy działający format `SavedModel`, możemy więc przejść do instalacji serwera TF Serving.

Instalowanie serwera TensorFlow Serving

Serwer TF Serving można zainstalować na wiele sposobów: za pomocą obrazu Dockera³, systemowego menedżera pakietów, wprost ze źródła itd. Skorzystajmy z Dockera, który jest zalecany przez sam zespół TensorFlow jako rozwiązań łatwe w instalacji, nieingerujące w ustawienia systemowe i wysoko-wydajne. Musimy najpierw zainstalować aplikację Docker (<https://www.docker.com/>), a potem pobrać oficjalny obraz TF Serving:

```
$ docker pull tensorflow/serving
```

Teraz możemy utworzyć kontener Dockera, aby uruchomić ten obraz:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
              -v "$ML_PATH/moj_model_mnist:/model/moj_model_mnist" \
              -e MODEL_NAME=moj_model_mnist \
              tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: moj_model_mnist version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

³ Jeżeli nie znasz z Dockera, umożliwia on łatwe pobieranie zestawów aplikacji zebranych w **obrazie Dockera** (ang. *Docker image*; włącznie z zależnościami i zazwyczaj dobrą konfiguracją domyślną), a następnie ich uruchamianie w systemie poprzez **silnik Dockera** (ang. *Docker engine*). W czasie przetwarzania obrazu silnik tworzy **kontener Dockera** (ang. *Docker container*), w którym przechowywane są aplikacje odizolowane od systemu (jeśli jednak chcesz, możesz im przyznać ograniczony dostęp). Rozwiązanie to przypomina nieco maszynę wirtualną, ale jest znacznie lepsze i pochłania mniej zasobów, gdyż kontener bazuje bezpośrednio na jądrze serwera głównego. Oznacza to, że obraz nie musi zawierać ani uruchamiać własnego jądra.

Zrobione! Uruchomiliśmy serwer TF Serving. Wczytaliśmy model MNIST (pierwszą wersję), który jest eksploatowany zarówno poprzez protokół gRPC (port 8500), jak i REST (port 8501). Oto znaczenie widocznych opcji wiersza polecenia:

-it

Włącza tryb interaktywny kontenera (dzięki niemu możesz zatrzymać serwer za pomocą kombinacji klawiszy *Ctrl+C*) i wyświetla rezultaty otrzymywane przez serwer.

--rm

Usuwa kontener po jego zatrzymaniu (nie ma potrzeby zaśmiecać komputera nieużywanymi kontenerami). Nie usuwa jednak obrazu.

-p 8500:8500

Silnik Dockera przekierowuje port 8500 TCP komputera do portu 8500 TCP kontenera. Domyślnie serwer TF Serving wykorzystuje ten port do eksploatowania interfejsu gRPC.

-p 8501:8501

Przekierowuje port 8501 TCP komputera do portu 8501 TCP kontenera. Domyślnie serwer TF Serving wykorzystuje ten port do eksploatowania interfejsu REST.

-v "\$ML_PATH/moj_model_mnist:/modele/moj_model_mnist"

Rzutuje katalog *\$ML_PATH/moj_model_mnist* komputera na ścieżkę */modele/mój_model_mnist* kontenera. W systemie Windows może być wymagana zamiana ukośników / na \ w ścieżce komputera (ale nie kontenera).

-e MODEL_NAME=moj_model_mnist

Wyznacza zmienną środowiskową MODEL_NAME kontenera, dzięki czemu serwer TF Serving wie, który model ma eksploatować. Domyślnie będzie szukać modeli w katalogu */modele* i automatycznie zacznie eksploatować najnowszą znalezioną wersję.

`tensorflow/serving`

Nazwa uruchamianego obrazu.

Wróćmy teraz do Pythona i wyślijmy zapytanie do tego serwera, najpierw poprzez interfejs REST, a następnie gRPC.

Wysyłanie zapytań do serwera TF Serving za pomocą interfejsu REST

Zacznijmy od utworzenia kwerendy. Musi ona zawierać nazwę sygnatury funkcji, którą chcesz wywołać, a także oczywiście dane wejściowe:

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Zwrót uwagę, że format JSON jest całkowicie tekstowy, zatem tablica *X_new* musi zostać przekształcona w listę Pythona, a potem sformatowana jako format JSON:

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

Prześlijmy teraz dane wejściowe do serwera TF Serving jako żądanie HTTP POST. Możemy to łatwo zrobić za pomocą biblioteki requests (nie jest ona częścią biblioteki standardowej Pythona, dlatego musisz ją najpierw zainstalować, np. za pomocą menedżera pip):

```
import requests

SERVER_URL = 'http://localhost:8501/v1/model/moje_mnist:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # W przypadku błędu wyświetla komunikat o błędzie
response = response.json()
```

W odpowiedzi otrzymujemy słownik zawierający klucz "predictions". Odpowiadającą mu wartością jest lista prognoz w postaci listy Pythona, dlatego przekształćmy ją w tablicę NumPy i zaokrąglamy zawarte w niej wartości zmiennoprzecinkowe do drugiej liczby po przecinku:

```
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Hurra, uzyskaliśmy prognozy! Model jest niemal na 100% pewny, że pierwsza cyfra to 7, na 99% pewny, że druga cyfra to 2, i ma niemal 96% pewności, że trzecią cyfrą jest 1.

Interfejs REST jest prosty i przejrzysty, sprawdza się świetnie, dane wejściowe i wyjściowe nie są zbyt duże. Ponadto w zasadzie każda aplikacja kliencka może tworzyć kwerendy REST bez konieczności instalowania dodatkowych zależności, podczas gdy inne protokoły nie zawsze są dostępne od ręki. Jednak rozwiązanie to bazuje na formacie JSON, który jest czysto tekstowy i dość rozwlekły. Musieliśmy na przykład przekształcać tablicę NumPy w listę Pythona, a każda wartość zmiennoprzecinkowa stała się łańcuchem znaków. Jest to bardzo niewydajne, zarówno pod względem czasu serializowania/deserializowania (przekształcanie wszystkich wartości zmiennoprzecinkowych w łańcuchy znaków i odwrotnie), jak i pojemności: okazuje się, że wiele wartości zmiennoprzecinkowych jest reprezentowanych jako łańcuchy 15 znaków, co oznacza ponad 120 bitów w przypadku 32-bitowych wartości zmiennoprzecinkowych! Skutkuje to dużym opóźnieniem i obciążeniem łącza podczas przesyłania dużych tablic NumPy⁴. Skorzystajmy zatem z interfejsu gRPC.



Podczas przesyłania dużych ilości danych o wiele lepiej jest skorzystać z interfejsu gRPC (jeżeli jest obsługiwany przez klienta), ponieważ bazuje na kompaktowym formacie binarnym i wydajnym protokole komunikacyjnym (zgodnym z formatem HTTP/2).

⁴ Trzeba jednak przyznać, że można uniknąć tego problemu, jeżeli najpierw przeprowadzimy serializację danych i zakodujemy je w formacie Base64, a dopiero później utworzymy żądanie REST. Ponadto zapytanie REST możemy kompresować za pomocą aplikacji gzip, co znacznie zmniejsza jego rozmiar.

Wysyłanie zapytań do serwera TF Serving za pomocą interfejsu gRPC

Interfejs gRPC oczekuje na wejściu serializowanego bufora protokołu PredictRequest, na wyjściu natomiast generuje serializowany bufor protokołu PredictResponse. Bufory te stanowią część biblioteki tensorflow-serving-api, którą trzeba osobno zainstalować (np. za pomocą menedżera pip). Utwórzmy najpierw żądanie:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

Listing ten tworzy bufor protokołu PredictRequest i wypełnia wymagane pola, m.in. podając nazwę modelu (zdefiniowaną wcześniej), nazwę sygnatury funkcji, którą chcemy wywołać, a także dane wejściowe w postaci bufora protokołu Tensor. Funkcja `tf.make_tensor_proto()` tworzy bufor protokołu Tensor na podstawie danego tensora lub tablicy NumPy, w tym przypadku `X_new`.

Wyślemy teraz zapytanie do serwera i otrzymamy odpowiedź (będzie nam do tego potrzebna biblioteka grpcio, którą możesz zainstalować za pomocą menedżera pip):

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

Listing ten nie jest skomplikowany: po zaimportowaniu klas tworzymy kanał komunikacyjny gRPC z komputerem lokalnym (`localhost`) poprzez port 8500 TCP, następnie tworzymy w tym kanale usługę gRPC i wykorzystujemy ją do przesłania żądania, wyznaczając 10-sekundową ramkę przekroczenia czasu (wywołanie jest synchroniczne: będzie zablokowane, dopóki nie otrzyma odpowiedzi lub dopóki nie zostanie przekroczyony wyznaczony czas oczekiwania). W tym przypadku kanał jest niezabezpieczony (brak szyfrowania i uwierzytelniania), ale zarówno interfejs gRPC, jak i serwer TF Serving obsługują bezpieczne kanały, wykorzystujące protokoły SSL/TLS.

Przekształćmy teraz bufor protokołu PredictResponse w tensor:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Jeżeli uruchomisz powyższy listing i wyświetlisz `y_proba.numpy().round(2)`, uzyskasz dokładnie takie same wyniki jak wcześniej. I to by było na tyle: wystarczy kilka wierszy kodu, aby uzyskać zdalny dostęp do modelu TensorFlow za pomocą interfejsu REST lub gRPC.

Wdrażanie nowej wersji modelu

Utwórzmy nową wersję modelu i eksportujmy jego format SavedModel do katalogu `moj_model_mnist/0002`:

```

model = keras.models.Sequential([...])
model.compile([...])
history = model.fit(...)

model_version = "0002"
model_name = "moj_model_mnist"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)

```

Serwer TensorFlow Serving sprawdza w regularnych odstępach czasu (można konfigurować opóźnienie) obecność nowych wersji modelu. Jeżeli znajdzie nową wersję, automatycznie zajmie się płynnym zastąpieniem starej wersji: domyślnie będzie odpowiadać na oczekujące zapytania (jeśli takowe będą występować) za pomocą dowolnej starszej wersji modelu, natomiast nowe kwerendy będą przetwarzane już za pomocą nowego modelu⁵. Gdy tylko oczekujące zapytania zostaną przetworzone, poprzednia wersja modelu zostanie usunięta. Widać to w dziennikach zdarzeń serwera TensorFlow Serving:

```

[...]
reserved resources to load servable {name: my_mnist_model version: 2}
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}

```

Mechanizm ten gwarantuje płynne przejście pomiędzy wersjami, ale może pochłonąć zbyt wiele pamięci operacyjnej (zwłaszcza w przypadku procesora graficznego, który dysponuje zazwyczaj najmniejszą ilością pamięci RAM). Wówczas możesz tak skonfigurować serwer TF Serving, aby obsłużył wszystkie oczekujące kwerendy za pomocą starej wersji modelu i wyładował go przed wczytaniem i użyciem nowej wersji. Dzięki takiej konfiguracji unikniemy sytuacji, w której dwie wersje modelu będą jednocześnie wczytane, ale usługa będzie niedostępna przez krótki czas.

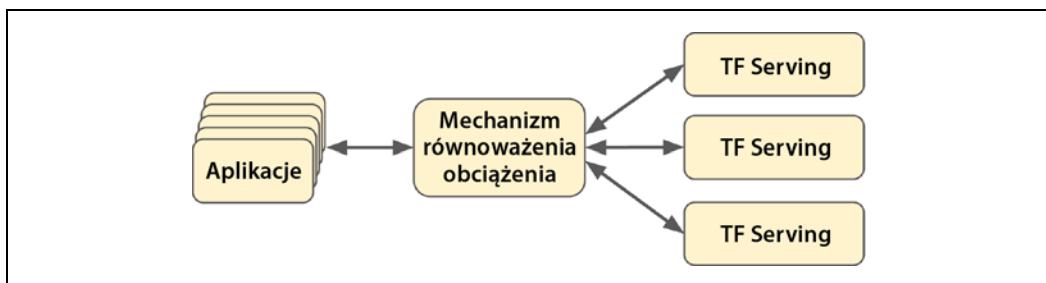


Inną znakomitą właściwością serwera TF Serving jest możliwość automatycznego tworzenia grup, którą uruchamiamy w momencie rozruchu serwera za pomocą opcji `--enable_batching`. Gdy serwer TF Serving otrzymuje wiele żądań w krótkim czasie (wartość opóźnienia jest modyfikowalna), to przed przetworzeniem ich przez model zostaną automatycznie zgrupowane. W ten sposób znacznie zwiększymy wydajność poprzez optymalizację użycia procesora graficznego. Po zwróceniu prognoz przez model serwer TF Serving przesyła wyniki do właściwych klientów. Możesz poprawić przepustowość kosztem opóźnienia poprzez zwiększenie opóźnienia grupowania kwerend (więcej informacji znajdziesz w dokumentacji opcji `--batching_parameters_file`).

⁵ Jeżeli obiekt SavedModel zawiera przykładowe próbki w katalogu `assets/extras`, możesz tak skonfigurować serwer TF Serving, aby model przetworzył te przykłady, zanim zacznie eksploatować nowe żądania. Jest to tzw. **rozgrzewanie modelu** (ang. *model warmup*): mamy w ten sposób pewność, że wszystkie elementy zostały właściwie wczytane, i uniemożliwiamy długiego czasu oczekiwania na pierwsze kwerendy.

Jak widać, serwer TF Serving skutecznie upraszcza wdrażanie nowych modeli. Ponadto jeżeli się okaże, że wersja druga modelu nie działa zgodnie z oczekiwaniami, to przywrócenie pierwszej wersji sprowadza się do usunięcia katalogu `moj_model_mnist/0002`.

Jeżeli przewidujesz, że model będzie otrzymywał wiele kwerend na sekundę, należy wdrożyć wiele serwerów i zrównoważyć obciążenie (rysunek 19.2). Proces ten wymaga wdrażania wielu kontenerów TF Serving i zarządzania nimi na tych serwerach. Jednym ze sposobów jest użycie narzędzie Kubernetes (<https://kubernetes.io/>), czyli systemu upraszczającego zarządzanie kontenerami na wielu serwerach. Jeżeli nie chcesz zakupić, utrzymywać i aktualizować infrastruktury sprzętowej, możesz korzystać z maszyn wirtualnych lub platformy usług rozproszonych, takiej jak Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud lub innej platformy w modelu PaaS (ang. *Platform as a Service* — platforma jako usługa). Zarządzanie wszystkimi maszynami wirtualnymi i kontenerami (nawet z pomocą systemu Kubernetes), konfigurowanie serwera TF Serving, poprawianie szczegółów i monitorowanie — wszystkie te czynności mogą zajmować cały etat pracy. Na szczęście niektórzy dostawcy usług całkowicie zdejmują to brzemień z Twoich barków. W tym rozdziale skorzystamy z platformy Google Cloud AI, ponieważ obecnie jako jedyna obsługuje procesory TPU i moduł TensorFlow 2 oraz oferuje zestaw interesujących usług sztucznej inteligencji (AutoML, Vision API czy Natural Language API), poza tym mam z nią największe doświadczenie. Niszę tę zajmuje jednak również kilka platform innych dostawców, w tym Amazon AWS SageMaker i Microsoft AI Platform, które umożliwiają eksploatację modeli TensorFlow.



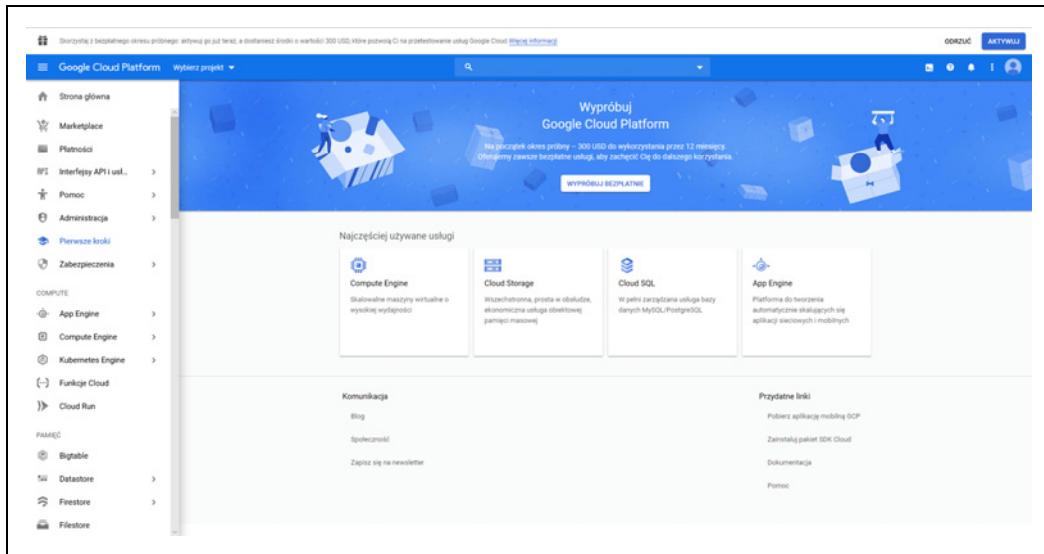
Rysunek 19.2. Skalowanie systemu TF Serving za pomocą równoważenia obciążenia

Zobaczmy teraz, jak możemy eksploatować nasz cudowny model MNIST w chmurze!

Tworzenie usługi predykcyjnej na platformie GCP AI

Zanim wdrożymy model, musimy się odpowiednio przygotować:

1. Zaloguj się na koncie Google i przejdź do konsoli Google Cloud Platform (GCP; <https://console.cloud.google.com/>) (rysunek 19.3). Jeżeli nie masz konta Google, musisz je założyć.
2. Jeżeli nie korzystałaś/korzystałś wcześniej z usługi GCP, musisz się zapoznać z warunkami korzystania z usług i je zaakceptować. Poświęć trochę czasu na zaznajomienie się z układem platformy. W czasie gdy piszę tę książkę, nowi użytkownicy mają do dyspozycji bezpłatny okres próbnego, w ramach którego otrzymują 300 dolarów do wykorzystania przez 12 miesięcy. Wykorzystamy ułamek tej kwoty podczas użytkowania usług omówionych w tym rozdziale.



Rysunek 19.3. Konsola platformy Google Cloud

Podczas rejestrowania się na okres próbnego musisz utworzyć konto rozliczeniowe i podać numer karty kredytowej, co ma na celu wyłącznie weryfikację użytkownika (prawdopodobnie po to, aby zapobiec wielokrotnemu wykorzystywaniu okresu próbnego przez tego samego użytkownika), ale nie zostaniesz obciążona/obciążony żadnymi kosztami. W razie potrzeby uaktywnij i zaktualizuj swoje konto do pełnej wersji.

3. Jeżeli korzystałaś/korzystałęś już wcześniej z platformy GCP i skończył Ci się okres próbnego, to za usługi używane w tym rozdziale musisz zapłacić. Nie powinno być to jednak zbyt kosztowne, zwłaszcza jeżeli nie zapomniałaś/zapomniałeś wyłączyć niepotrzebnych usług. Zanim uruchomisz jakąkolwiek usługę, zapoznaj się z warunkami naliczania za nią opłat. Niniejszym oświadczam, że nie ponoszę żadnej odpowiedzialności za sytuację, w której się okaże, że musisz zapłacić więcej, niż przewidywałaś/przewidywałeś! Sprawdź również, czy konto rozliczeniowe jest aktywne. Zrobisz to poprzez kliknięcie pozycji *Płatności* umieszczonej w menu nawigacyjnym po lewej stronie ekranu. Sprawdź tam, czy masz skonfigurowany sposób płatności oraz czy konto rozliczeniowe jest aktywne.
4. Każdy zasób platformy GCP przynależy do projektu. Dotyczy to wszystkich maszyn wirtualnych, z jakich możesz korzystać, przechowywanych plików oraz realizowanych zadań uczenia. W momencie utworzenia konta zostaje automatycznie utworzony pierwszy projekt, zatytułowany *My First Project*⁶. Jeżeli chcesz, możesz zmienić wyświetlającą nazwę w ustawieniach projektu: w menu nawigacyjnym (po lewej stronie ekranu) kliknij opcję *Administracja*, a następnie *Ustawienia*, zmień nazwę projektu i wciśnij przycisk *Zapisz*. Zwróć uwagę, że projekt zawiera także niepowtarzalny identyfikator i numer. Możesz wybrać identyfikator projektu w czasie jego tworzenia, ale nie da się go później zmienić. Numer projektu jest generowany automatycznie i jego również nie można zmienić. Jeżeli chcesz utworzyć nowy projekt, kliknij nazwę

⁶ Jeżeli zamierzasz zmienić nazwę projektu, pamiętaj, że platforma GCP nie rozpoznaje polskich znaków — *przyp. tłum.*

dotychczasowego projektu w górnej części ekranu, następnie kliknij opcję *Nowy projekt* i wprowadź nowy identyfikator. Sprawdź, czy dla nowego projektu są naliczane opłaty.



Zawsze wyznaczaj alarm przypominający o wyłączeniu usług, które są potrzebne jedynie przez kilka godzin, w przeciwnym wypadku możesz pozostawić je włączone przez całe dnie lub miesiące, co może spowodować naliczanie znaczących kosztów.

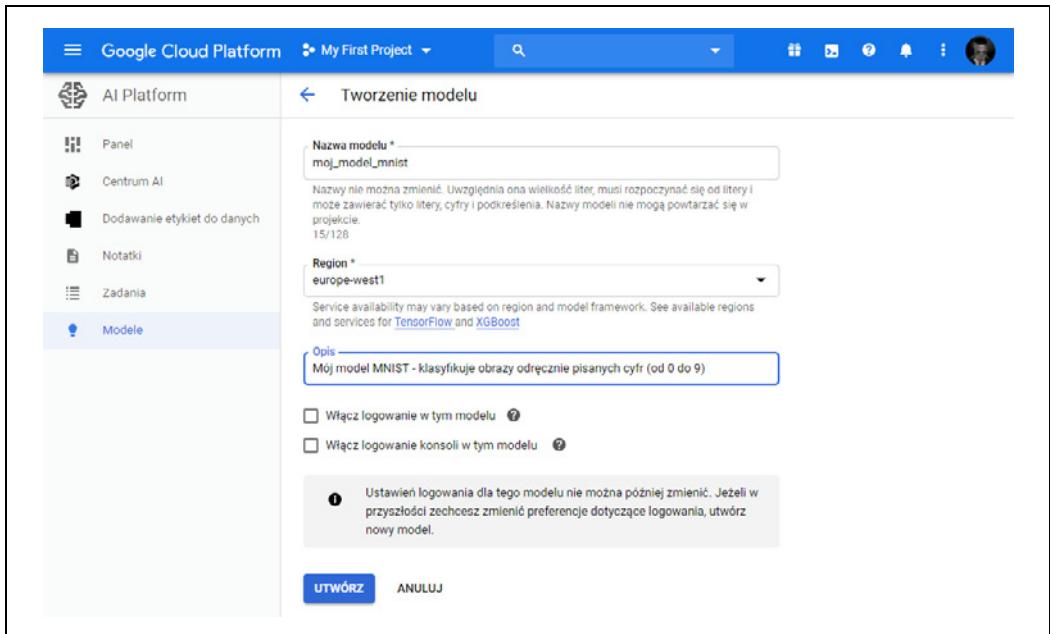
5. Po uaktywnieniu konta GCP wraz z kontem rozliczeniowym możemy zacząć korzystać z usług. Pierwszą z nich jest Przeglądarka Cloud Storage (ang. *Google Cloud Storage* — GCS): to właśnie w niej będziemy przechowywać obiekty SavedModel, dane uczące itd. W menu nawigacyjnym przejdź do sekcji *Pamięć*, stamtąd do *Przechowywanie danych*, a potem wybierz *Przeglądarka*. Wszystkie Twoje pliki trafiają do przynajmniej jednego **zasobnika** (ang. *bucket*). Kliknij przycisk *Utwórz zasobnik* i wybierz dla niego jakąś nazwę (być może trzeba będzie najpierw uaktywnić interfejs Storage API). Usługa GCS wykorzystuje globalną przestrzeń nazw, dlatego takie proste nazwy jak *machine-learning* najprawdopodobniej będą już zajęte. Upewnij się, że nazwa zasobnika jest zgodna z konwencją nazewnictwa DNS, gdyż może być on używany w rekordach DNS. Ponadto nazwy zasobników są publiczne, dlatego lepiej nie umieszczać w nich prywatnych informacji. W celu zapewnienia niepowtarzalności często jako prefiks jest umieszczana nazwa domeny lub firmy albo po prostu losowa liczba jako część nazwy. Wybierz lokalizację, w jakiej zasobnik ma być przechowywany, a pozostałe ustawienia domyślne prawdopodobnie nam wystarczą. Możesz teraz wcisnąć przycisk *Utwórz*.
6. Wczytaj do zasobnika utworzony wcześniej katalog *moj_model_mnist* (dowolną liczbę wersji). W tym celu wystarczy przejść do przeglądarki GCS, wybrać zasobnik i przeciągnąć do niego folder *moj_model_mnist* z komputera (rysunek 19.4). Ewentualnie możesz kliknąć opcję *Prześlij folder* i wybrać katalog *moj_model_mnist*. Domyślnie maksymalny dostępny rozmiar dla obiektu SavedModel wynosi 250 MB, ale można poprosić o większy przydział.

The screenshot shows the 'Szczegóły zasobnika' (Storage details) page. At the top, there are buttons for 'EDYTUJ ZASOBNIK', 'ODŚWIEŻ ZASOBNIK', and 'SAMOUCZEK'. Below that, the bucket name 'zasobnik_mojego_modelu_mnist' is displayed. There are tabs for 'Obiekty', 'Przegląd', 'Uprawnienia', and 'Blokada zasobnika'. Under the 'Obiekty' tab, there are buttons for 'Prześlij pliki', 'Prześlij folder', and 'Utwórz folder'. A search bar says 'Filtruj według prefiksów...'. Below that, a breadcrumb navigation shows 'Zasobnik / zasobnik_mojego_modelu_mnist'. A table lists three objects being uploaded:

Nazwa	Rozmiar	Typ
saved_model.pb		Ukończone
variables.data-00000-of-00001		Ukończone
variables.index		Ukończone

Rysunek 19.4. Wczytywanie obiektu SavedModel do Przeglądarki Cloud Storage

7. Musimy teraz skonfigurować usługę Platform AI (znaną wcześniej pod nazwą ML Engine), dzięki czemu będą używane właściwe modele i ich wersje. W menu nawigacyjnym przejdź do sekcji *Sztuczna inteligencja*, wybierz opcję *AI Platform*, a potem *Modele*. Kliknij przycisk *Włącz API* (potrwa to kilka minut), a następnie *Utwórz model*. Podaj informacje o modelu (rysunek 19.5) i kliknij przycisk *Utwórz*.



Rysunek 19.5. Tworzenie nowego modelu w usłudze Google Cloud AI Platform

8. Utworzony model w usłudze AI Platform, musimy określić jego wersję. Na liście dostępnych modeli wybierz utworzony model, następnie wybierz opcję *Utwórz wersję* (dostępna po rozwinięciu menu z prawej strony modelu) i uzupełnij informacje o wersji (rysunek 19.6): nazwę, opis, wersję Pythona (3.5 lub nowszą), platformę (TensorFlow), wersję platformy (2.0, jeśli jest dostępna, a jeśli nie jest, to 1.13.1)⁷, wersję środowiska wykonawczego ML (masz do wyboru tylko 1.13), rodzaj urządzenia (ang. *machine type*; na razie pozostaw *Procesor jednorzędniowy*), ścieżkę do modelu w usłudze GCS (np. `gs://zasobnik_mojego_modelu_mnist/moj_model_mnist/0002/`), skalowanie (pozostaw *Autoskalowanie*) oraz minimalną liczbę kontenerów (tu: węzłów) TF Serving działających przez cały czas (pozostaw to pole puste). Na koniec kliknij przycisk *Zapisz*.

⁷ W czasie pisania książki nie była dostępna wersja TensorFlow 2 w usłudze Platform AI, ale nic nie szkodzi — możesz wybrać wersję 1.13 i obiekty SavedModel utworzone w module TensorFlow 2 będą działały jak należy.

[←](#) Utwórz wersję

Aby utworzyć nową wersję modelu, wprowadź niezbędne poprawki do pliku z zapisanym modelem, wyeksportuj go i zapisz wyeksportowany model w Cloud Storage. [Więcej informacji](#)

Nazwa *
v0001

Nazwy nie można zmienić, rozpoznawana jest wielkość użytych w niej liter, musi zaczynać się od litery i może zawierać wyłącznie litery, cyfry i znaki podkreślenia. 5/128

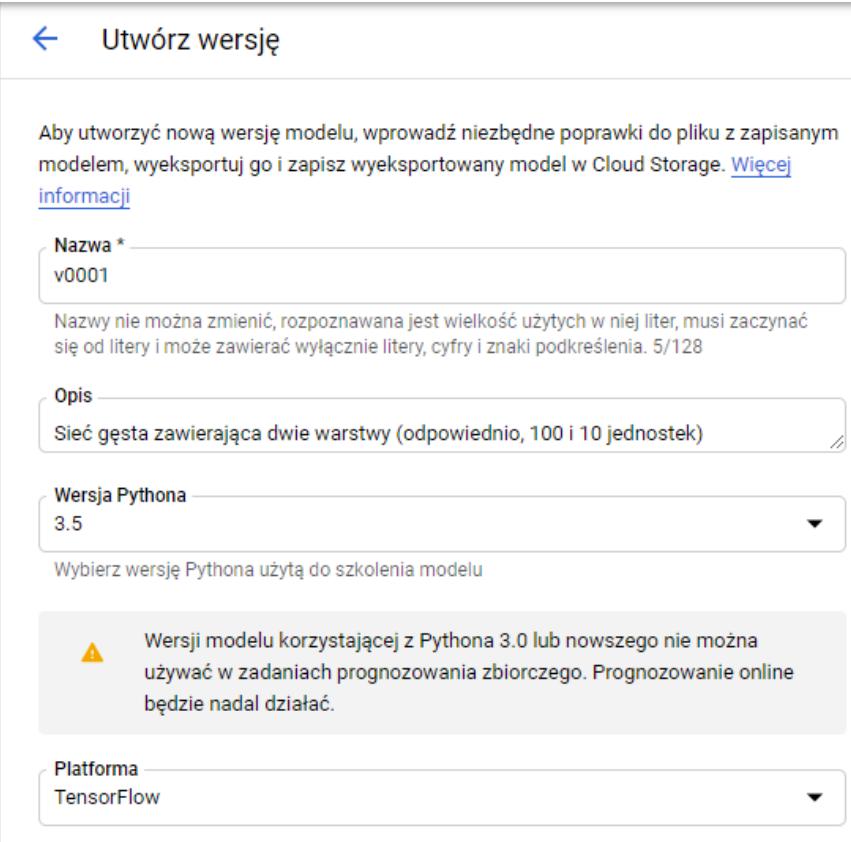
Opis
Sieć gęsta zawierająca dwie warstwy (odpowiednio, 100 i 10 jednostek)

Wersja Pythona
3.5

Wybierz wersję Pythona użytą do szkolenia modelu

⚠ Wersji modelu korzystającej z Pythona 3.0 lub nowszego nie można używać w zadaniach prognozowania zbiorczego. Prognozowanie online będzie nadal działać.

Platforma
TensorFlow



Rysunek 19.6. Tworzenie nowej wersji modelu w usłudze Google Cloud AI Platform

Gratulacje! Właśnie wdrożyłaś/wdrożyłeś swój pierwszy model do środowiska rozproszonego. Wybrałaś/wybrałeś autoskalowanie, dlatego usługa AI Platform będzie uruchamiać kolejne kontenery TF Serving w miarę zwiększania się liczby kwerend na sekundę i będzie równoważyć rozkład tych zapytań pomiędzy poszczególnymi kontenerami. W przypadku zmniejszania się wartości QPS kontenery będą automatycznie wyłączane. Koszty są więc bezpośrednio powiązane z liczbą kwerend na sekundę (a także z wyborem urządzenia i ilością danych przechowywanych w usłudze GCS). Taki model rozliczeniowy jest szczególnie korzystny dla użytkowników okazjonalnych oraz dla usług wykazujących znaczące skoki użytkowania, jak również dla młodych firm (start-upów): cena pozostaje znikoma aż do zwiększenia zakresu działalności.

Wyślijmy teraz zapytanie do tej usługi prognozowania!



Jeżeli nie korzystasz z usługi prognozowania, AI Platform wyłączy wszystkie kontenery. Oznacza to, że będziesz płacić tylko za użytkowaną pojemność magazynu (kilka centów za każdy gigabajt miesiąc). Zwróć uwagę, że gdy wysłesz zapytanie do usługi, konieczne będzie uruchomienie kontenera TF Serving, co zajmuje kilka sekund. Jeżeli takie opóźnienie jest niedopuszczalne, to podczas tworzenia wersji modelu będziesz musiał wyznaczyć wartość 1 w polu minimalnej liczby węzłów. Oznacza to oczywiście, że przynajmniej jedno urządzenie będzie bez przerwy uruchomione, co niesie ze sobą wzrost miesięcznych kosztów.

Korzystanie z usługi prognozowania

Usługa AI Platform zasadniczo uruchamia jedynie serwery TF Serving, więc jeżeli znasz adres URL, na który należy przesyłać kwerendy, to powinien działać ten sam kod, którego użyliśmy wcześniej. Pojawia się jednak pewien problem. Otóż platforma GCP zajmuje się także szyfrowaniem i uwierzytelnianiem. Szyfrowanie bazuje na protokołach SSL/TLS, a uwierzytelnianie na tokenach: wraz z każdym żądaniem wysyłanym do serwera musi być dołączony do niego tajny token uwierzytelniający. Zatem zanim Twój kod będzie mógł skorzystać z usługi prognozowania (lub jakiekolwiek innej usługi platformy GCP), musi najpierw otrzymać token. Nauczysz się go dostarczać już wkrótce, jednak musimy skonfigurować uwierzytelnianie i nadać aplikacji odpowiednie uprawnienia dostępu do platformy GCP. Mamy do wyboru dwa rodzaje uwierzytelniania:

- Twoja aplikacja (tzn. kod kliencki, który będzie wysyłał kwerendy do usługi prognozowania) mogłaby do uwierzytelniania wykorzystywać poświadczenie użytkownika w postaci nazwy użytkownika i hasła używanych w serwisie Google. W ten sposób aplikacja uzyskiwałaby takie same uprawnienia jak Twój konto GCP, co jest zdecydowanie nadmiarowe. Do tego musiałabyś/musiałbyś wdrożyć poświadczenie do aplikacji, przez co naraziłabyś/naraziłbyś je na kradzież i złodziej uzyskałby pełen dostęp do Twojego konta GCP. Krótko mówiąc, nie wybieraj tej opcji; jest ona wymagana w bardzo rzadkich sytuacjach (np. wtedy, gdy aplikacja potrzebuje dostępu do konta GCP użytkownika).
- Kod kliencki może być uwierzytelniany za pomocą **konta usługi** (ang. *service account*). Jest to konto reprezentujące aplikację, nie użytkownika. Zazwyczaj wyznacza ono bardzo ograniczone uprawnienia dostępu: jedynie te, które są potrzebne. Jest to zalecane rozwiązanie.

Utwórzmy więc konto usługi dla Twojej aplikacji. W menu nawigacyjnym kliknij *Administracja*, potem *Konta usługi*, następnie przycisk *Utwórz konto usługi*, po czym wypełnij formularz (nazwa, identyfikator i opis konta usługi) i wciśnij przycisk *Utwórz* (rysunek 19.7). Teraz trzeba nadać temu kontu uprawnienia dostępu. Wybierz rolę *Programista ML Engine* — w ten sposób konto usługi będzie mogło uzyskiwać prognozy i raczej nic poza tym. Możesz ewentualnie dać dostęp wybranym użytkownikom do konta usługi (przydaje się to wtedy, gdy Twoje konto użytkownika GCP stanowi część organizacji i chcesz, aby pozostali jej członkowie wdrażali aplikacje oparte na tym koncie usługi lub sami zarządzali kontem usługi). Kliknij teraz przycisk *Utwórz klucz*, dzięki czemu eksportujesz klucz prywatny konta usługi, wybierz format JSON i kliknij kolejny przycisk *Utwórz*. Nie udostępniaj tego klucza osobom nieupoważnionym!

Utworzenie konta usługi

1 Szczegóły konta usługi — 2 Przynaj temu kontu usługi dostęp do projektu (opcjonalnie) —
3 Przynaj użytkownikom dostęp do tego konta usługi (opcjonalnie)

Szczegóły konta usługi

Nazwa konta usługi
moja_aplikacja

Wyświetlana nazwa tego konta usługi

Identyfikator konta usługi
moja-aplikacja @dark-sensor-265913.iam.gserviceaccount.com X C

Opis konta usługi
Oto moja aplikacja, która wykorzystuje prognozy uzyskiwane przez model

Opisz, do czego będzie służyć to konto usługi.

UTWÓRZ ANULUJ

Rysunek 19.7. Tworzenie nowego konta usługi w panelu administracyjnym

Doskonale! Utwórzmy teraz niewielki skrypt, który będzie wysyłał zapytania do usługi prognozowania. Serwis Google udostępnia kilka bibliotek ułatwiających dostęp do jego usług:

Google API Client Library

Jest to dość prosta warstwa oparta na standardach OAuth 2.0 (<https://oauth.net/>; używany do uwierzytelniania) i REST. Możesz wykorzystywać ją ze wszystkimi usługami GCP, w tym AI Platform. Najłatwiej zainstalować ją za pomocą menedżera pip: biblioteka ta nosi nazwę `google-api-python-client`.

Google Cloud Client Libraries

Biblioteki te są nieco bardziej rozbudowane: każda z nich jest stworzona z myślą o określonej usłudze, np. GCS, Google BigQuery, Google Cloud Natural Language czy Google Cloud Vision. Wszystkie te biblioteki można zainstalować za pomocą menedżera pip (np. biblioteka GCS Client ma nazwę `google-cloud-storage`). Gdy biblioteka kliencka jest dostępna dla danej usługi, lepiej korzystać z niej niż z Google API Client Library, ponieważ ma zaimplementowane najlepsze dostępne rozwiązania i często wykorzystuje protokół gRPC zamiast REST, co skutkuje lepszą wydajnością.

W czasie gdy pisałem tę książkę, nie istniała biblioteka kliencka dla usługi AI Platform, dlatego użyjemy Google API Client Library. Będzie ona korzystała z klucza prywatnego konta usługi; możemy

określić jego położenie za pomocą zmiennej środowiskowej GOOGLE_APPLICATION_CREDENTIALS albo przed uruchomieniem skryptu, albo w samym skrypcie, na przykład:

```
import os  
  
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "moj_klucz_konta_uslugi.json"
```



Jeżeli wdrażasz aplikację w maszynie wirtualnej w usłudze Google Compute Engine (GCE), w kontenerze przy użyciu usługi Google Cloud Kubernetes Engine, jako aplikację sieciową w ramach usługi Google Cloud App Engine lub jako mikrouslugę w ramach usługi Funkcje Cloud i nie zdefiniowałeś zmiennej środowiskowej GOOGLE_APPLICATION_CREDENTIALS, to biblioteka będzie wykorzystywać domyślne konto usługi (czyli np. domyślne konto usługi GCE, jeżeli aplikacja korzysta z usługi GCE).

Następnie należy utworzyć obiekt zasobów, określający dostęp do usługi prognozowania⁸:

```
import googleapiclient.discovery  
  
project_id = "dark-sensor-265913" # Wstaw tu identyfikator swojego projektu  
model_id = "moj_model_mnist"  
model_path = "projects/{}/models/{}".format(project_id, model_id)  
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Zwróć uwagę, że możesz dodać /versions/0001 (lub dowolną inną wersję) do ścieżki model_path, jeżeli zależy Ci na określeniu wersji, do której mają być wysyłane kwerendy — rozwiązań to jest przydatne w testach A/B lub podczas testowania nowej wersji na małej grupie użytkowników przed opublikowaniem (tzw. **testy kanarka**). Utwórzmy teraz funkcję wykorzystującą obiekt zasobów do wywołania usługi prognozowania i otrzymywania predykcji:

```
def predict(X):  
    input_data_json = {"signature_name": "serving_default",  
                      "instances": X.tolist()}  
    request = ml_resource.predict(name=model_path, body=input_data_json)  
    response = request.execute()  
    if "error" in response:  
        raise RuntimeError(response["error"])  
    return np.array([pred[output_name] for pred in response["predictions"]])
```

Funkcja przyjmuje tablicę NumPy zawierającą obrazy wejściowe i przygotowuje słownik, który biblioteka kliencka przekształci w format JSON (tak jak wcześniej). Następnie przygotowuje żądanie predykcji i realizuje je: wyświetla komunikat o wyjątku, jeżeli odpowiedź będzie zawierać błąd, w przeciwnym razie wydobywa prognozy dla każdego przykładu i wstawia je do tablicy NumPy. Sprawdźmy ją w działaniu:

```
>>> Y_probas = predict(X_new)  
>>> np.round(Y_probas, 2)  
array([[0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],  
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. ],  
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

⁸ Jeżeli będzie pojawiła się komunikat o błędzie informujący, że nie znaleziono modułu google.appengine, wyznacz parametr cache_discovery=False w wywołaniu metody build(). Więcej informacji znajdziesz pod adresem <https://stackoverflow.com/q/55561354>.

Tak! Dysponujesz teraz działającą w chmurze usługą prognozowania, która będzie automatycznie skalowana do dowolnej wartości QPS, a także możesz bezpiecznie wysyłać do niej kwerendy z dowolnego miejsca. Ponadto koszt jej przechowywania, gdy nie jest używana, jest równy niemal zero: płacisz jedynie co miesiąc kilka centów za każdy gigabajt zajęty w usłudze GCS. Jakby tego było mało, uzyskujesz też dostęp do szczegółowych rejestrów zdarzeń i wskaźników dzięki usłudze Google Stackdriver (<https://cloud.google.com/stackdriver/>).

A może chcesz wdrożyć model do aplikacji mobilnej albo do urządzenia wbudowanego?

Wdrażanie modelu na urządzeniu mobilnym lub wbudowanym

Jeżeli musisz wdrożyć model do urządzenia mobilnego lub wbudowanego, decydującą rolę może odgrywać rozmiar tego modelu, ponieważ duży może wymagać zbyt wiele czasu na pobranie oraz zabierać zbyt wiele pamięci operacyjnej i mocy obliczeniowej, co prowadzi do zmniejszenia responsywności aplikacji, przegrzewania się urządzenia i szybkiego wyczerpywania baterii. Aby tego uniknąć, musisz stworzyć model dostosowany do urządzeń mobilnych, lekki i wydajny, bez nadmiernego poświęcania jego dokładności. Biblioteka TFLite (<https://www.tensorflow.org/lite>) zawiera narzędzia serwerowe⁹ ułatwiające wdrażanie modeli do urządzeń mobilnych i wbudowanych. Przyświecają jej trzy główne cele:

- zmniejszenie rozmiaru modelu w celu skrócenia czasu pobierania i zaoszczędzenia pamięci operacyjnej;
- zmniejszenie liczby obliczeń wymaganych do uzyskania każdej prognozy w celu zredukowania opóźnienia, zużycia baterii i nagrzewania urządzenia;
- dostosowanie modelu do ograniczeń sprzętowych.

Aby zmniejszyć rozmiar modelu, konwerter modeli TFLite przyjmuje obiekt SavedModel i kompresuje go do znacznie lżejszego formatu opartego na technologii FlatBuffer (<https://google.github.io/flatbuffers/>). Jest to wydajna międzyplatformowa biblioteka serializacji (przypominająca nieco buforey protokołów), stworzona przez firmę Google z myślą o grach. Została tak zaprojektowana, że można wczytywać obiekty FlatBuffer wprost do pamięci operacyjnej bez konieczności wstępnego przetwarzania — rozwiązanie to zmniejsza czas wczytywania i wykorzystanie pamięci. Po wczytaniu modelu do urządzenia mobilnego lub wbudowanego interpreter TFLite zacznie uzyskiwać za jego pomocą prognozy. Oto sposób przekształcenia formatu SavedModel w obiekt FlatBuffer, który następnie zapisujemy w pliku *.tflite*:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("przekształcony_model.tflite", "wb") as f:
    f.write(tflite_model)
```

⁹ Sprawdź również platformę Graph Transform Tools (<https://homl.info/tfgtt>) zespołu TensorFlow, która służy do modyfikowania i optymalizowania grafów obliczeniowych.

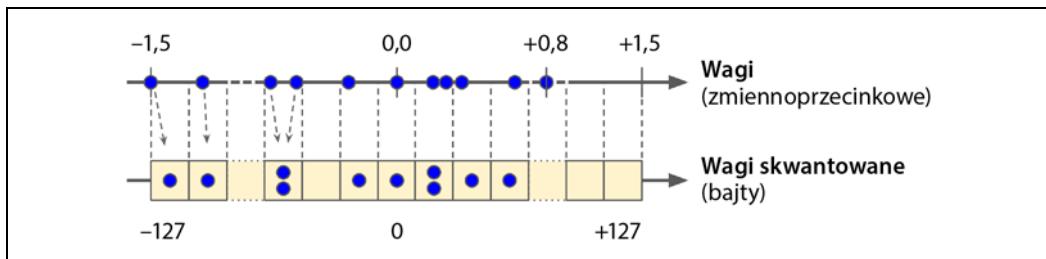


Możesz również zapisać model tf.keras bezpośrednio w formacie FlatBuffer za pomocą funkcji `from_keras_model()`.

Konwerter optymalizuje także model: zarówno go zmniejsza, jak i redukuje związane z nim opóźnienie. Usuwa wszystkie operacje, które nie są niezbędne do uzyskiwania prognoz (np. operacje uczenia), i optymalizuje obliczenia tam, gdzie to możliwe — na przykład operacja $3 \times a + 4 \times a + 5 \times a$ zostanie przekształcona w postać $(3+4+5) \times a$. Próbuje on także łączyć operacje — przykładowo warstwy normalizacji wsadowej zostają dołączone do operacji dodawania i mnożenia w warstwie poprzedzającej, jeśli to możliwe. Jeżeli chcesz się przekonać, w jakim stopniu biblioteka TFLite optymalizuje model, pobierz jeden z gotowych modeli TFLite (https://www.tensorflow.org/lite/guide/hosted_models), rozpakuj archiwum, otwórz znakomite narzędzie wizualizacji grafów Netron (<https://lutzroeder.github.io/netron/>) i wczytaj plik `.pb`, aby ujrzeć pierwotny model. Jest duży i skomplikowany, prawda? Teraz otwórz zoptymalizowany model `.tflite` i podziwiaj jego piękno!

Innym sposobem zredukowania rozmiaru modelu (nie licząc oczywiście wprowadzania mniejszych struktur sieci neuronowej) jest stosowanie mniejszej liczby bitów — na przykład jeżeli wykorzystujesz połówkowe wartości zmiennoprzecinkowe (16 bitów), a nie pełne (32 bity), to rozmiar modelu zmniejszy się dwukrotnie kosztem spadku dokładności (zazwyczaj niewielkiego). Ponadto proces uczenia będzie przebiegał szybciej i będzie wykorzystywana mniej więcej o połowę mniejsza ilość pamięci operacyjnej.

To jednak nie koniec możliwości konwertera TFLite, ponieważ jest on w stanie kwantyzować wagi do postaci 8-bitowych, stałoprzecinkowych wartości całkowitych! Oznacza to czterokrotne zmniejszenie rozmiaru w stosunku do 32-bitowych wartości zmiennoprzecinkowych. Najprostszym rozwiązaniem jest tak zwana **kwantyzacja potreningu** (ang. *post-training quantization*): jak sama nazwa wskazuje, wagi zostają skwantowane po zakończeniu procesu uczenia za pomocą całkiem prostej, ale skutecznej techniki kwantyzacji symetrycznej. Najpierw zostaje znaleziona maksymalna wartość bezwzględna wagi m , która następnie jest przekształcana z zakresu wartości zmiennoprzecinkowych od $-m$ do m na zakres wartości stałoprzecinkowych od -127 do 127. Na zaprezentowanym przykładzie (rysunek 19.8), jeżeli wagi mieszczą się w zakresie od -1,5 do 0,8, to bajty -127, 0 i 127 będą odpowiadać wartościami zmiennoprzecinkowymi, odpowiednio: -1,5, 0,0 i 1,5. Zwróć uwagę, że w technice kwantyzacji symetrycznej wartość 0,0 odpowiada zawsze bajtowi 0 (zauważ także, że wartości bajtowe od 68 do 127 nie będą używane, ponieważ odwzorowują one wartości zmiennoprzecinkowe większe od 0,8).



Rysunek 19.8. Przejście z 32-bitowych wartości zmiennoprzecinkowych do 8-bitowych wartości stałoprzecinkowych za pomocą kwantyzacji symetrycznej

Aby przeprowadzić kwantyzację potreningu, wystarczy dodać parametr `OPTIMIZE_FOR_SIZE` do listy optymalizacji konwertera przed wywołaniem metody `convert()`:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

Technika ta drastycznie zmniejsza rozmiar modelu, dlatego można go znacznie szybciej pobrać i zajmuje mniej miejsca. Jednak w momencie jego uruchomienia skwantyzowane wagi zostają przywrócone do wartości zmiennoprzecinkowych (tak odzyskane wartości nie są identyczne z pierwotnymi wartościami zmiennoprzecinkowymi, ale zbliżone w wystarczającym stopniu, zatem wartość funkcji straty dokładności jest zazwyczaj akceptowalna). Aby uniknąć ich ciągłego przeliczania, odzyskane wartości zmiennoprzecinkowe zostają umieszczone w pamięci podręcznej, więc nie zajmują miejsc w pamięci operacyjnej. Nie odczuwamy również spadku szybkości obliczeń.

Najskuteczniejszym sposobem ograniczenia opóźnienia i zużycia mocy obliczeniowej jest skwantyzowanie także funkcji aktywacji, dzięki czemu obliczenia będą przeprowadzane wyłącznie na wartościach stałoprzecinkowych, bez konieczności stosowania operacji zmiennoprzecinkowych. Nawet jeżeli wyznaczyś tę samą precyzyję (np. zastąpisz 32-bitowe wartości zmiennoprzecinkowe 32-bitowymi wartościami stałoprzecinkowymi), to obliczenia stałoprzecinkowe wymagają mniejszej liczby cykli procesora, zużywają mniej energii i generują mniej ciepła. A jeżeli do tego zmniejszysz precyzyję (np. do 8-bitowych wartości stałoprzecinkowych), to jeszcze znacznie przyspieszysz działanie modelu. Ponadto niektóre akceleratory sieci neuronowych (np. Edge TPU) mogą przetwarzarć jedynie wartości stałoprzecinkowe, dlatego niezbędna jest pełna kwantyzacja, zarówno wag, jak i funkcji aktywacji. Proces ten może być zrealizowany po zakończeniu uczenia: pierwszym etapem jest kalibracja, polegająca na znalezieniu maksymalnej wartości bezwzględnej pobudzeń, dlatego należy dostarczyć bibliotece TFLite reprezentatywną próbę danych uczących (nie musi być duża), które zostaną przetworzone przez model, co pozwoli określić statystyki funkcji aktywacji wymagane do kwantyzacji (faza ta najczęściej nie trwa zbyt długo).

Głównym problemem kwantyzacji jest utrata pewnego stopnia dokładności: jest ona równoważna dodaniu szumu do wag i pobudzeń. Jeżeli utrata dokładności okaże się zbyt znacząca, możesz wykorzystać technikę **uczenia w kontekście kwantyzacji** (ang. *quantization-aware training*). Oznacza to dodanie udawanych operacji kwantyzacji do modelu, dzięki czemu model uczy się ignorować szum kwantyzacji w trakcie treningu, wagi końcowe będą więc bardziej odporne na skutki kwantyzacji. Poza tym etap kalibracji może być przeprowadzony automatycznie podczas uczenia, co upraszcza cały proces.

Wyjaśniłem podstawowe pojęcia związane z biblioteką TFLite, ale na opis procesu tworzenia aplikacji mobilnej lub przeznaczonej na urządzenie wbudowane należałoby poświęcić osobną książkę. Na szczęście taka już istnieje: jeżeli chcesz nauczyć się pisać aplikacje TensorFlow na urządzenia mobilne/wbudowane, polecam lekturę książki *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers* (<https://homl.info/tinyml>), której autorami są Pete Warden (kierownik zespołu TFLite) i Daniel Situnayake.

Teraz nauczysz się przyspieszać obliczenia za pomocą procesorów graficznych!

Moduł TensorFlow w przeglądarce

Co w przypadku, gdybyśmy chcieli zastosować model w serwisie internetowym, bezpośrednio na poziomie przeglądarki użytkownika? Rozwiążanie to może być przydatne w wielu sytuacjach, na przykład:

- Gdy aplikacja jest używana w okolicznościach przerywanego lub powolnego łącza (np. serwis turystyczny), dlatego jedynie działanie modelu bezpośrednio po stronie klienta może wpływać pozytywnie na odbiór witryny.
- Gdy wymagane są błyskawiczne reakcje modelu (np. w grach sieciowych). Zrezygnowanie z konieczności wysyłania zapytań do serwera w celu uzyskania prognoz zdecydowanie redukuje opóźnienie i zwiększa responsywność serwisu.
- Gdy usługa sieciowa uzyskuje prognozy na podstawie prywatnych danych użytkownika i chcesz chronić jego prywatność poprzez przeprowadzanie obliczeń po stronie klienta, dzięki czemu informacje nie są przesyłane na zewnątrz¹⁰.

W tych wszystkich przypadkach możesz eksportować model do specjalnego formatu, wczytywanego przez bibliotekę TensorFlow.js (<https://www.tensorflow.org/js>). Biblioteka ta może następnie wykorzystać model do uzyskiwania prognoz bezpośrednio w przeglądarce użytkownika. Projekt TensorFlow.js zawiera narzędzie `tensorflowjs_converter`, które przekształca obiekt `SavedObject` lub plik modelu Keras do formatu **TensorFlow.js Layers** — jest to katalog zawierający zestaw podzielonych plików wag w formacie binarnym, a także plik `model.json`, opisujący architekturę modelu i łączący ją z plikami wag. Format ten został zoptymalizowany pod względem szybkości pobierania. Użytkownicy mogą następnie pobrać model i uzyskiwać prognozy z poziomu przeglądarki za pomocą biblioteki TensorFlow.js. Oto fragment pokazujący wygląd interfejsu JavaScriptu:

```
import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel('https://przyklad.com/tfjs/model.json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

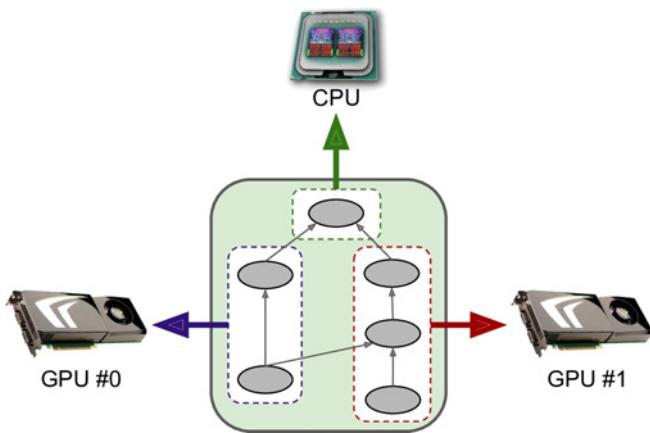
Również w tym przypadku jest to temat na osobną książkę. Jeżeli chcesz uzyskać więcej informacji o bibliotece TensorFlow.js, to Anirudh Koul, Siddha Ganju i Meher Kasam napisali książkę *Practical Deep Learning for Cloud, Mobile, and Edge* (<https://homl.info/tfjsbook>; O'Reilly).

Przyspieszanie obliczeń za pomocą procesorów graficznych

W rozdziale 11. omówiłem kilka technik, które znaczco przyśpieszają proces uczenia: lepszą inicjalizację wag, normalizację wsadową, zaawansowane optymalizatory i tak dalej. Jednak nawet przy zastosowaniu tych metod trenowanie dużej sieci neuronowej na komputerze wyposażonym w jeden procesor może trwać dni, a nawet tygodnie.

Ten podrozdział poświęciłem zagadniению przyspieszania modeli za pomocą procesorów graficznych. Nauczysz się także rozdzielać obliczenia pomiędzy wiele urządzeń, w tym jednostki CPU i wiele jednostek GPU (rysunek 19.9). Najpierw będziemy korzystać z jednego komputera, ale w dalszej części rozdziału rozdysponujemy obliczenia pomiędzy wiele serwerów.

¹⁰ Osoby zainteresowane tematem powinny zapoznać się z koncepcją **uczenia federacyjnego** (ang. *federated learning*; <https://www.tensorflow.org/federated>).



Rysunek 19.9. Równoczesne przetwarzanie grafu TensorFlow na wielu urządzeniach

Dzięki procesorom graficznym nie musimy teraz czekać całymi dniami czy tygodniami na zakończenie procesu uczenia — wystarczy na to kilka minut lub godzin. W ten sposób nie tylko możesz zaoszczędzić mnóstwo czasu, ale oznacza to także, że możesz swobodnie eksperymentować z różnymi modelami i często trenować model na nowych danych.



Często do uzyskania znacznego skoku wydajności wystarczy dokupienie kart graficznych do jednego komputera. Rozwiążanie to faktycznie w zupełności wystarczy w wielu przypadkach — nie musisz wcale korzystać z wielu urządzeń. Na przykład możesz zazwyczaj trenować sieć neuronową za pomocą komputera wyposażonego w cztery procesory graficzne równie szybko jak za pomocą ośmiu kart graficznych umieszczonych w różnych urządzeniach. Wynika to z opóźnienia komunikacyjnego, które cechuje konfiguracje rozproszone. Podobnie lepiej wybrać jedną mocniejszą kartę graficzną niż kilka słabszych jednostek GPU.

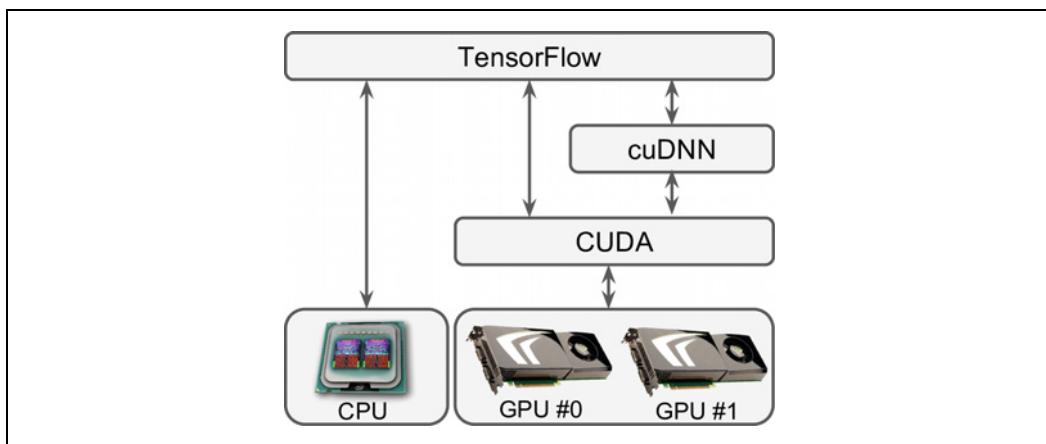
Pierwszym etapem jest uzyskanie dostępu do procesora graficznego. Masz tutaj dwie możliwości: albo kupić własne karty graficzne, albo skorzystać z maszyn wirtualnych w chmurze wyposażonych w jednostki GPU. Zastanówmy się nad pierwszym rozwiązaniem.

Zakup własnej karty graficznej

Jeżeli chcesz się zaopatryć we własną kartę graficzną, poświęć trochę czasu na dokonanie dobrego wyboru. Tim Dettmers zamieścił znakomity wpis (<https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/>), mający na celu pomóc w wyborze, a do tego jest regularnie aktualizowany, dlatego polecam przeczytanie go z uwagą. W czasie gdy piszę tę książkę, moduł TensorFlow obsługuje jedynie karty firmy Nvidia wyposażone w technologię CUDA Compute Capability 3.5+ (<https://developer.nvidia.com/cuda-gpus>), a także oczywiście procesory TPU firmy Google, ale być może rozszerzy obsługę również na produkty innych firm. Ponadto na razie jednostki TPU są dostępne wyłącznie na platformie GCP, ale jest bardzo prawdopodobne, że w niedalekiej przyszłości trafią do sprzedaży i moduł TensorFlow będzie je obsługiwał. Mówiąc krótko, zajrzyj do dokumentacji

modułu TensorFlow (<https://www.tensorflow.org/install>), aby się dowiedzieć, jakie urządzenia są obsługiwane na daną chwilę.

Jeżeli chcesz kupić kartę graficzną firmy Nvidia, musisz zainstalować odpowiednie sterowniki i biblioteki¹¹. Wśród nich znajdziesz bibliotekę **CUDA** (ang. *Compute Unified Device Architecture*), która pozwala programistom wykorzystywać karty graficzne zaopatrzone w tę funkcję do przeprowadzania różnorodnych obliczeń (nie tylko akceleracji graficznej). Z kolei dzięki bibliotece **cuDNN** (ang. *CUDA Deep Neural Network* — głęboka sieć neuronowa CUDA) zostaje wprowadzona obsługa podstawowych elementów sieci neuronowych na kartach graficznych. Dostępne są w niej zoptymalizowane implementacje najczęściej stosowanych składników sieci neuronowych, takie jak warstwy aktywacji, normalizacja, przednie i wsteczne sploty czy warstwy łączące (zob. rozdział 14.). Stanowi ona część zbioru Deep Learning SDK (w celu jego pobrania należy utworzyć konto Nvidia Developer). Moduł TensorFlow wykorzystuje biblioteki CUDA i cuDNN do sterowania kartami graficznymi i przyspieszania obliczeń (rysunek 19.10).



Rysunek 19.10. Moduł TensorFlow wykorzystuje biblioteki CUDA i cuDNN do sterowania kartami graficznymi i przyspieszania głębokich sieci neuronowych

Po zainstalowaniu kart(y) GPU wraz ze wszystkimi wymaganymi sterownikami i bibliotekami możesz użyć polecenia `nvidia-smi` do sprawdzenia, czy biblioteka CUDA została prawidłowo zainstalowana. Zostanie wyświetlona lista wszystkich kart graficznych, a także procesów uruchomionych na każdej z nich:

```
$ nvidia-smi
Sun Jun 2 10:05:22 2019
+-----+
| NVIDIA-SMI 418.67      Driver Version: 410.79 CUDA Version: 10.0 |
+-----+
| GPU Name      Persistence-M | Bus-Id      Disp.A  | Volatile Uncorr. ECC |
| Fan Temp     Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====|
| 0 Tesla T4        Off | 00000000:00:04.0 Off |          0 |
```

¹¹ Zajrzyj do dokumentacji zawierającej szczegółowe i aktualne instrukcje instalacji, gdyż proces ten jest dość często modyfikowany.

<i>N/A</i>	<i>61C</i>	<i>P8</i>	<i>17W / 70W</i>	<i>0MiB / 15079MiB</i>	<i>0%</i>	<i>Default</i>
<hr/>						
<i>Processes:</i>				<i>GPU Memory</i>		
<i>GPU</i>	<i>PID</i>	<i>Type</i>	<i>Process name</i>		<i>Usage</i>	
<hr/>						
<i>No running processes found</i>						
<hr/>						

W czasie gdy pisałem tę książkę, wymagane także było zainstalowanie modułu TensorFlow w wersji dla procesorów graficznych (tzn. biblioteki tensorflow-gpu); twórcy jednak pracują nad ujednoliceniem procesu instalacji dla komputerów wyposażonych wyłącznie w jednostki CPU oraz w jednostki CPU i GPU, dlatego sprawdź w dokumentacji, którą bibliotekę należy zainstalować. Właściwa instalacja każdej wymaganej biblioteki jest długim i nieco skomplikowanym zadaniem (i niebo wali się na głowę, jeżeli nie zainstalujesz odpowiednich wersji bibliotek), dlatego mamy do dyspozycji obraz Dockera zawierający wszystkie niezbędne składniki. Aby jednak kontener Dockera mógł uzyskać dostęp do procesora GPU, musisz mimo wszystko zainstalować sterowniki karty graficznej na danym komputerze.

Aby sprawdzić, czy moduł TensorFlow rzeczywiście widzi karty graficzne, przeprowadzimy następujący test:

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Funkcja `is_gpu_available()` sprawdza, czy jest dostępny przynajmniej jeden procesor graficzny. Funkcja `gpu_device_name()` podaje nazwę pierwszej jednostki GPU — domyślnie właśnie w tej jednostce będą przetwarzane operacje. Z kolei funkcja `list_physical_devices()` zwraca listę wszystkich dostępnych urządzeń graficznych (w naszym przykładzie widnieje tylko jedno takie urządzenie)¹².

Być może nie chcesz poświęcić czasu i pieniędzy na zakup własnej karty graficznej. Zawsze możesz skorzystać z wirtualnego procesora graficznego w chmurze!

Korzystanie z maszyny wirtualnej wyposażonej w procesor graficzny

Obecnie wszystkie główne platformy usług rozproszonych mają w ofercie maszyny wirtualne wyposażone w procesory graficzne, niektóre wstępnie skonfigurowane za pomocą wszystkich niezbędnych sterowników i bibliotek (w tym również modułu TensorFlow). Platforma GCP wyznacza różne przydziały jednostek GPU, zarówno globalnie, jak i regionalnie: nie możesz utworzyć tysięcy takich

¹² W wielu listingach w tym rozdziale wykorzystujemy eksperymentalne interfejsy API. Prawdopodobnie w kolejnych wersjach zostaną przeniesione do głównego API. Jeżeli więc jakaś eksperymentalna funkcja nie będzie działać, spróbuj usunąć wyraz `experimental`, a być może rozwiąże to problem. Jeśli nie, to interfejs mógł zostać nieco zmodyfikowany; zatrzymy do notatnika Jupyter, ponieważ będę go regularnie aktualizował.

maszyn wirtualnych bez zgody firmy Google¹³. Domyślnie przydział globalny wynosi 0, zatem nie możesz korzystać z żadnej maszyny wirtualnej wyposażonej w jednostkę GPU. Z tego powodu pierwszą czynnością, jaką należy wykonać, jest wysłanie prośby o zwiększenie przydziału globalnego. W konsoli GCP otwórz menu nawigacyjne i przejdź do opcji *Administracja*, a następnie *Limity*. Rozwiń zakładkę *Wskaźnik* i kliknij *Nic*, aby odznaczyć wszystkie elementy. Poszukaj teraz pozycji GPU i wybierz *GPUs (wszystkie regiony)* przy odpowiednim limicie. Jeżeli wartość przydziału jest równa 0 (lub niewystarczająca do Twoich potrzeb), to zaznacz pole przy tym limicie (tylko ono powinno być zaznaczone) i kliknij przycisk *Edytuj limity*. Podaj wymagane informacje i wciśnij przycisk *Wyślij zgłoszenie*. Zgłoszenie jest przetwarzane zazwyczaj kilka godzin (do kilku dni) i przeważnie akceptowane. Domyślnie istnieje również przydział na jeden procesor GPU na region oraz na typ karty graficznej. Te przydziały też możesz zwiększyć: tak jak wcześniej, rozwiń listę *Wskaźnik*, kliknij *Nic*, poszukaj GPU i wybierz rodzaj karty graficznej (np. *NVIDIA P4 GPU*), następnie rozwiń menu *Lokalizacja*, wybierz opcję *Nic*, wyszukaj interesującą Cię lokalizację, zaznacz limit, który chcesz zmienić, i wciśnij przycisk *Edytuj limity*.

Gdy prośba o powiększenie limitu zostanie spełniona, możesz błyskawicznie utworzyć maszynę wirtualną zaopatrzoną w co najmniej jeden procesor graficzny za pomocą narzędzia **Deep Learning VM Images**, dostępnego w ramach usługi Google Cloud AI Platform, którą znajdziesz na stronie <https://homl.info/dlvm>. Kliknij przycisk *View Console*, a następnie *Uruchom w Compute Engine* i wypełnij formularz. Zwróć uwagę, że nie wszystkie lokalizacje mają wszystkie rodzaje kart graficznych, a niektóre nie mają ich wcale (aby zobaczyć, czy i jakie typy jednostek są dostępne, zmień lokalizację). Upewnij się, że wybraną platformą (*Framework*) jest TensorFlow 2.1, a także zaznacz opcję *Install NVIDIA GPU driver automatically on first startup*. Warto również zaznaczyć opcję *Enable access to JupyterLab via URL instead of SSH (beta)*, gdyż znacznie ułatwi to uruchamianie notatników Jupyter na tej maszynie wirtualnej poprzez silnik JupyterLab (jest to alternatywny interfejs sieciowy, który służy do otwierania notatników Jupyter). Po utworzeniu maszyny wirtualnej przejdź do sekcji *Sztuczna inteligencja* w menu nawigacyjnym, a następnie wybierz *AI Platform* i *Notatki*. Po pojawienniu się instancji notatnika (może to potrwać kilka minut, więc co jakiś czas klikaj przycisk *Odśwież*) kliknij odnośnik *Otwórz JupyterLab*. Zostanie uruchomiony interfejs JupyterLab na maszynie wirtualnej i połączony z Twoją przeglądarką. Możesz tworzyć notatniki Jupyter i realizować dowolny kod w tej maszynie wirtualnej, a także korzystać z dobrodziejstw procesorów graficznych!

Jeżeli jednak zależy Ci jedynie na przeprowadzaniu szybkich testów lub łatwym udostępnianiu notatników współpracownikom, wypróbuj narzędzie Colaboratory.

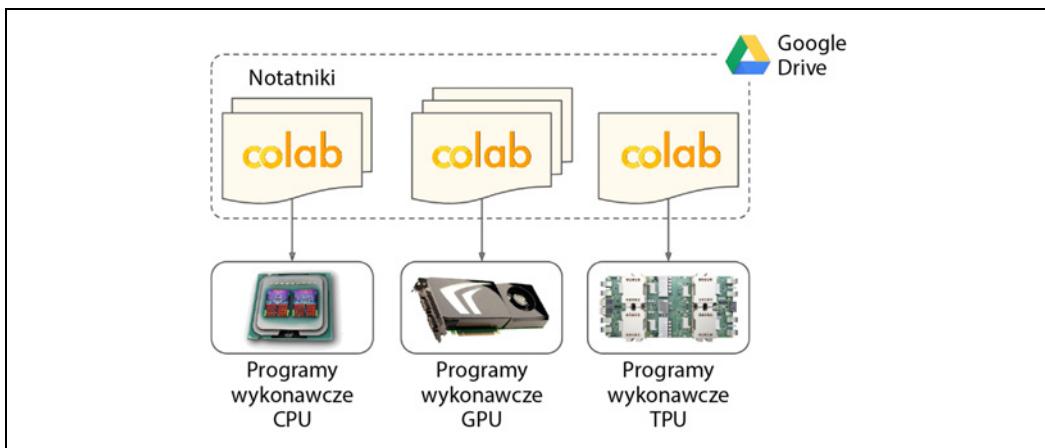
Colaboratory

Najprostszym i najtańszym sposobem uzyskania dostępu do maszyny wirtualnej wyposażonej w procesor graficzny jest narzędzie **Colaboratory** (w skrócie **Colab**). Jest ono darmowe! Wystarczy odwiedzić stronę <https://colab.research.google.com/> i utworzyć nowy notatnik Pythona 3 — będzie on przechowywany w Google Drive (ewentualnie możesz otworzyć dowolny notatnik Jupyter z repozytorium GitHub, Google Drive, a nawet wczytywać własne notatniki). Interfejs narzędzia Colab

¹³ Limity mają prawdopodobnie za zadanie uniemożliwienie złoczyńcom wykorzystywania platformy GCP za pomocą skradzionych kart kredytowych do generowania kryptowalut.

przypomina interfejs Jupytera, ale dodatkowo możesz udostępniać i wykorzystywać notatniki jak standardowe dokumenty Google, a poza tym istnieje kilka drobnych różnic (np. możesz tworzyć przydatne widżety za pomocą specjalnych komentarzy umieszczanych w kodzie).

Po otwarciu notatnika Colab będzie on działał na googlowskiej darmowej maszynie wirtualnej, zwanej Collab Runtime (rysunek 19.11). Domyślnie bazuje ona wyłącznie na procesorze CPU, ale możesz to zmienić: otwórz menu *Runtime* i wybierz opcję *Change Runtime Type*, po czym w menu rozwijalnym *Hardware Type* wybierz *GPU* i wcisnij przycisk *Save*. W rzeczywistości możesz wybrać nawet procesor TPU (tak, naprawdę możesz używać jednostki tensorowej za darmo; jej omówieniem zajmiemy się jednak w dalszej części rozdziału, dlatego na razie pozostanmy przy procesorach graficznych)!



Rysunek 19.11. Programy wykonawcze Colab i notatniki

Colab ma pewne ograniczenia: przede wszystkim istnieje limit liczby jednocześnie uruchomionych notatników (obecnie 5 na każdy typ programu wykonawczego). Ponadto w dziale FAQ znajdziemy zapis, zgodnie z którym narzędzie Colaboratory jest przeznaczone do użytku interaktywnego. Wszelkie długotrwałe obliczenia wykonywane w tle (zwłaszcza za pomocą procesorów graficznych) mogą zostać zatrzymane. Nie należy też używać narzędzia do generowania kryptowalut. Poza tym nastąpi automatyczne rozłączenie interfejsu sieciowego z programem wykonawczym, jeśli nic nie będziesz przy nim robić przez jakiś czas (ok. 30 minut). Podczas przywracania połączenia program wykonawczy może zostać zresetowany, dlatego zawsze eksportuj istotne dane (np. pobierz je lub zapisz w serwisie Google Drive). Nawet jeżeli nie nastąpi rozłączenie, program wykonawczy zostaje automatycznie wyłączony po 12 godzinach, gdyż nie jest przeznaczony do realizowania długotrwałych obliczeń. Pomimo tych ograniczeń jest to wspariałe narzędzie do szybkiego przeprowadzania testów, błyskawicznego uzyskiwania rezultatów i udostępniania środowiska pracy współpracownikom.

Zarządzanie pamięcią operacyjną karty graficznej

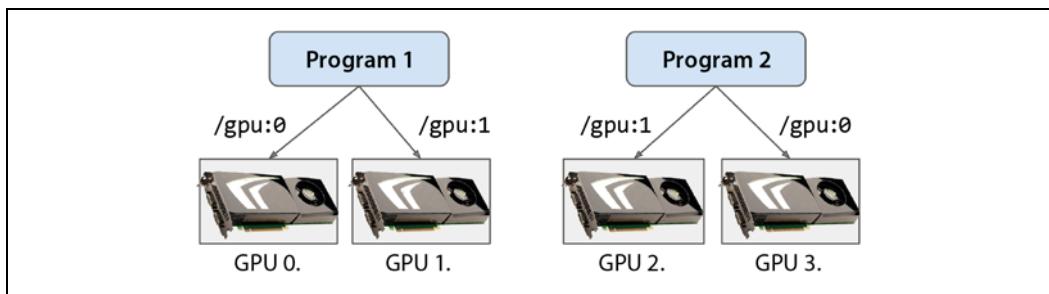
Domyślnie moduł TensorFlow automatycznie przechwytuje całą pamięć operacyjną dostępną na kartach graficznych w momencie uruchomienia pierwszych obliczeń. Ma to na celu zapobieżenie fragmentacji pamięci operacyjnej procesorów GPU. Z tego powodu jeśli włączysz drugi program

TensorFlow (lub jakąkolwiek inną aplikację próbującą uzyskać dostęp do karty graficznej), to szybko zapełnisz całą pamięć RAM. Nie przytrafi się to tak często, jak mogłoby się wydawać, gdyż najczęściej korzystamy tylko z jednego programu TensorFlow na komputerze: zazwyczaj skryptu uczącego, węzła TF Serving albo notatnika Jupyter. Jeżeli z jakiegoś powodu musisz mieć uruchomione jednocześnie dwa programy (np. po to, aby uczyć równolegle dwa różne modele na jednym komputerze), to musisz w miarę równo rozdzielić pamięć operacyjną karty graficznej pomiędzy te procesy.

Jeżeli masz w komputerze zainstalowanych wiele kart graficznych, prostym rozwiązaniem jest uruchomienie osobnych procesów na poszczególnych kartach graficznych. Aby to zrobić, najłatwiej jest wyznaczyć zmienną środowiskową CUDA_VISIBLE_DEVICES, dzięki czemu każdy proces będzie „widział” wyłącznie wyznaczoną kartę graficzną. Wyznacz także w zmiennej środowiskowej CUDA_DEVICE_ORDER wartość PCI_BUS_ID, co sprawi, że dany identyfikator będzie zawsze dotyczył tej samej karty graficznej. Na przykład jeżeli masz cztery karty graficzne, możesz uruchomić dwa programy i do każdego z nich przypdzielić po dwie jednostki GPU — w tym celu wprowadź następujące polecenia, każde z nich w osobnym oknie terminalu:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py  
# A w drugim terminalu:  
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Pierwszy program będzie widział wyłącznie karty 0. i 1. (/gpu:0 i /gpu:1), natomiast drugi program weźmie pod uwagę wyłącznie karty 2. i 3. (/gpu:1 i /gpu:0; zwróć uwagę na kolejność). Wszystko będzie działało jak należy (rysunek 19.12). Oczywiście możesz także zdefiniować te zmienne środowiskowe w Pythonie za pomocą konstrukcji os.environ["CUDA_DEVICE_ORDER"] i os.environ["CUDA_VISIBLE_DEVICES"], pod warunkiem że zrobisz to przed uruchomieniem programu TensorFlow.

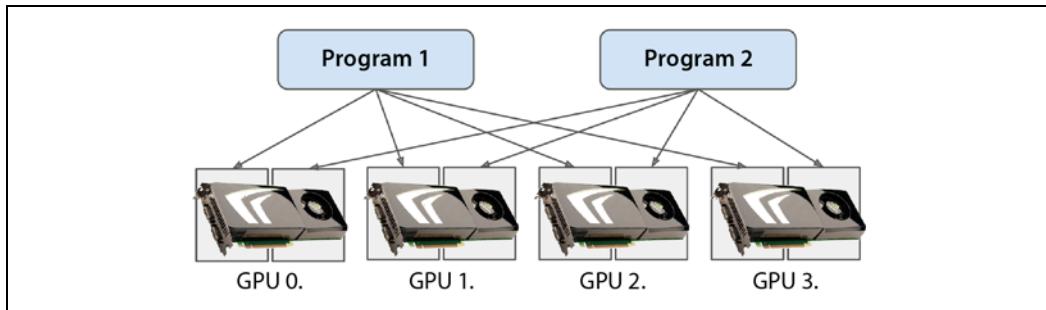


Rysunek 19.12. Każdy program będzie miał po dwie karty graficzne dla siebie

Innym rozwiązaniem jest umożliwienie modułowi TensorFlow użycia tylko określonej ilości pamięci operacyjnej. Należy to zrobić tuż po zimportowaniu modułu TensorFlow. Na przykład gdybyśmy chcieli umożliwić korzystanie tylko z 2 GiB pamięci RAM każdej karty graficznej, musielibyśmy utworzyć **wirtualne urządzenie GPU** (ang. *virtual GPU device*), zwane też **logicznym urządzeniem GPU** (ang. *logical GPU device*), dla każdej fizycznej karty graficznej i wyznaczyć jego limit pamięci operacyjnej do 2 GiB (czyli 2048 MiB):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):  
    tf.config.experimental.set_virtual_device_configuration(  
        gpu,  
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Jeśli założymy, że masz cztery graficzne, z których każda zawiera 4 GiB pamięci RAM, to teraz możemy uruchomić równolegle dwa programy i każdy z nich będzie wykorzystywał wszystkie cztery procesory graficzne (rysunek 19.13).



Rysunek 19.13. Każdy program wykorzystuje wszystkie cztery karty graficzne, ale uzyskuje na każdej z nich dostęp do jedynie 2 GiB pamięci

Jeśli wpiszesz polecenie nvidia-smi podczas działania obydwu programów, zauważysz, że każdy z procesów wykorzystuje 2 GiB pamięci operacyjnej każdej z kart:

```
$ nvidia-smi
[...]
+-----+
| Processes:
| GPU  PID  Type  Process name          GPU Memory Usage |
| 0    2373   C    /usr/bin/python3        2241MiB |
| 0    2533   C    /usr/bin/python3        2241MiB |
| 1    2373   C    /usr/bin/python3        2241MiB |
| 1    2533   C    /usr/bin/python3        2241MiB |
[...]
```

Jeszcze jednym rozwiązaniem jest funkcja rezerwowania pamięci operacyjnej jedynie wtedy, gdy moduł TensorFlow będzie jej potrzebował (również w tym przypadku musimy to zrobić tuż po zimportowaniu modułu TensorFlow):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

Kolejnym sposobem jest wyznaczenie wartości true w zmiennej środowiskowej TF_FORCE_GPU_ALLOW_GROWTH. Opcja ta spowoduje, że moduł TensorFlow nie zwolni raz zarezerwowanej pamięci (aby uniknąć fragmentacji pamięci), nie licząc oczywiście sytuacji, w której działanie programu zostaje zakończone. Opcja ta nie gwarantuje deterministycznego zachowania (tzn. jeden program może przestać działać, ponieważ zużycie pamięci w drugiej aplikacji znacznie wzrosło), dlatego zazwyczaj nie jest zalecana w środowisku produkcyjnym. Jednak zdarzają się sytuacje, w których okazuje się bardzo przydatna, na przykład podczas uruchamiania na jednym komputerze wielu notatników Jupyter, z których część bazuje na module TensorFlow. Z tego właśnie powodu w programach wykonawczych Colab zmienna TF_FORCE_GPU_ALLOW_GROWTH ma wyznaczoną wartość true.

Na koniec w pewnych sytuacjach możesz chcieć rozdzielać kartę graficzną na co najmniej dwa **wirtualne procesory GPU** (ang. *virtual GPU*), na przykład do testowania algorytmu rozproszonego

(jest to dobre rozwiązanie do wypróbowania listingów zawartych w dalszej części rozdziału, nawet jeśli dysponujesz tylko jedną kartą graficzną, choćby w postaci programu wykonawczego Colab). Poniższy listing rozdziela pierwszy procesor GPU na dwie wirtualne jednostki graficzne zawierające pod 2 GiB pamięci operacyjnej (jak zwykle należy to zrobić tuż po zainstalowaniu modułu TensorFlow):

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Obydwa urządzenia wirtualne będą nazwane `/gpu:0` i `/gpu:1`. Ty zaś możesz umieszczać w każdym z nich operacje i zmienne tak, jakby były dwoma niezależnymi procesorami graficznymi. Sprawdźmy teraz, w jaki sposób moduł TensorFlow decyduje o rozmieszczaniu zmiennych i realizowaniu operacji na poszczególnych urządzeniach.

Umieszczanie operacji i zmiennych na urządzeniach

Dokumentacja (<http://download.tensorflow.org/paper/whitepaper2015.pdf>)¹⁴ modułu TensorFlow zawiera opis przyjaznego algorytmu **dynamicznego rozmieszczania** (ang. *dynamic placer*), który automagicznie rozdziela operacje pomiędzy wszystkie dostępne urządzenia i bierze pod uwagę takie rzeczy, jak czas trwania obliczeń we wcześniejszych przebiegach grafu, oszacowania rozmiarów tensorów wejść i wyjść dla każdej operacji, ilość pamięci operacyjnej dostępnej na każdym urządzeniu, opóźnienie podczas przesyłania danych pomiędzy urządzeniami, wymagania i ograniczenia narzucone przez użytkownika itd. W praktyce algorytm ten okazał się mniej skuteczny od małego zestawu reguł rozmieszczania definiowanych przez użytkownika, dlatego został ostatecznie porzucony przez zespół TensorFlow.

Z kolei interfejsy `tf.keras` i `tf.data` zazwyczaj spisują się bardzo dobrze w kwestii rozmieszczania operacji i zmiennych w odpowiednich miejscach (np. skomplikowane obliczenia są umieszczane w procesorze graficznym, a wstępny przetwarzaniem danych zajmuje się jednostka CPU). Jeżeli jednak wymagasz większej kontroli, możesz ręcznie rozmieszczać operacje i zmienne:

- Jak już wspomniałem, przeważnie chcemy umieszczać operacje wstępnego przetwarzania danych w jednostce CPU, a operacje sieci neuronowych w jednostce GPU.
- Karty graficzne mają zazwyczaj dość ograniczone pasmo komunikacyjne, dlatego ważne jest, aby unikać niepotrzebnego przesyłania danych do procesora GPU i z niego.
- Dodawanie pamięci operacyjnej procesora głównego jest zadaniem prostym i względnie tanim, dlatego zwykle nie cierpimy na jego brak, czego nie można powiedzieć o pamięci RAM ściśle zintegrowanej z kartą graficzną: jest to kosztowny, a więc ograniczony zasób, dlatego jeśli jakaś zmienna nie jest potrzebna w kilku następnych krokach uczenia, powinna prawdopodobnie zostać umieszczona w jednostce CPU (np. zestawy danych zazwyczaj są do niej przydzielane).

¹⁴ Martin Abadi i in., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, dokumentacja Google Research (2015).

Domyślnie wszystkie zmienne i operacje zostaną umieszczone w pierwszej jednostce GPU (`/gpu:0`) oprócz zmiennych i operacji niezawierających jądra GPU¹⁵ — one trafiają do procesora CPU (`/cpu:0`). Atrybut `device` tensora lub zmiennej informuje nas, do którego urządzenia dany element trafił¹⁶:

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

Na razie możesz spokojnie zignorować przedrostek `/job:localhost/replica:0/task:0` (pozwala umieszczać operacje w innych komputerach podczas korzystania z klastra TensorFlow; w dalszej części rozdziału przyjrzymy się grupom zadań, replikom i zadaniom). Jak widać, pierwsza zmienna została umieszczona w procesorze GPU 0., czyli w domyślnym urządzeniu. Jednak druga zmienna trafiła do jednostki CPU — wynika to z faktu, że nie istnieją jądra GPU dla zmiennych stałoprzecinkowych (lub dla operacji zawierających tenzory stałoprzecinkowe), dlatego moduł TensorFlow skorzystał z procesora głównego.

Jeżeli chcesz umieścić operację na innym urządzeniu niż domyślne, skorzystaj z kontekstu `tf.device()`:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```



Jednostka CPU jest zawsze traktowana jako jedno urządzenie (`/cpu:0`), nawet jeśli procesor składa się z wielu rdzeni. Każda operacja mająca jądro wielowątkowe po umieszczeniu w procesorze głównym może być przetwarzana równolegle przez wiele rdzeni.

Jeżeli spróbujesz otwarcie umieszczenia operacji lub zmienną w urządzeniu, które nie istnieje lub dla którego nie istnieje odpowiednie jądro, zostanie wyświetlony komunikat o wyjątku. W pewnych sytuacjach lepiej jednak pozostać przy procesorze głównym, na przykład jeżeli program może działać zarówno na komputerze zawierającym tylko procesor główny, jak i na komputerze zaopatrzonym w kartę graficzną, możesz chcieć, aby zapis `tf.device("/gpu:0")` był ignorowany na komputerach pozbawionych procesorów GPU. W tym celu możesz wywołać `tf.config.set_soft_device_placement(True)` tuż po zimportowaniu modułu TensorFlow: gdy żądanie rozmieszczenia zakończy się niepowodzeniem, zastosowane zostaną domyślnie reguły rozmieszczenia (tzn. domyślnie GPU 0., jeśli jest dostępne urządzenie i jądro, a w przeciwnym razie CPU 0.).

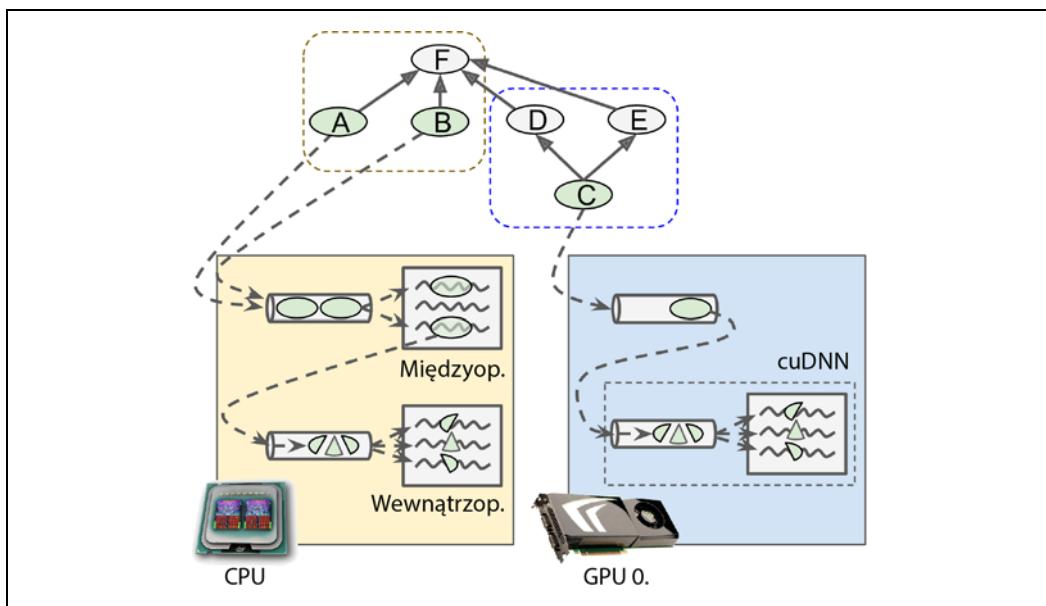
W jaki sposób moduł TensorFlow realizuje te operacje na różnych urządzeniach?

¹⁵ Jak wiesz z rozdziału 12., jądro jest implementacją zmiennej lub operacji dostosowaną do określonego typu danych i urządzenia. Na przykład istnieje jądro GPU funkcji `float32 tf.matmul()`, ale nie ma jądra GPU dla funkcji `int32 tf.matmul()` (dostępne jest wyłącznie jądro CPU).

¹⁶ Listę rozmieszczeń we wszystkich urządzeniach uzyskasz za pomocą funkcji `tf.debugging.set_log_device_placement(True)`.

Przetwarzanie równolegle na wielu urządzeniach

Jak widzieliśmy w rozdziale 12., jedną z zalet funkcji TF jest możliwość zrównoleglania. Przyjrzyjmy się tej koncepcji nieco uważniej. Gdy moduł TensorFlow zaczyna przetwarzać funkcje TF, najpierw analizuje graf i wyszukuje listę operacji, które mają zostać obliczone, a następnie zlicza zależności występujące w poszczególnych operacjach. W dalszej kolejności zostają dodane operacje niezawierające żadnych zależności (tzn. każda operacja źródłowa) do kolejki ewaluacji w danym urządzeniu (rysunek 19.14). Po zrealizowaniu operacji licznik zależności każdej zależnej od niej operacji zostaje zmniejszony. Gdy licznik ten osiągnie wartość 0, zostaje przesłany do kolejki ewaluacji. Kiedy wszystkie węzły wymagane przez moduł TensorFlow będą obliczone, zostaną zwrócone ich wyniki.



Rysunek 19.14. Przetwarzanie równolegle grafu TensorFlow

Operacje w kolejce CPU zostają przesłane do puli wątków zwanej **międzyoperacyjną pulą wątków** (ang. *inter-op thread pool*). Jeżeli procesor zawiera wiele rdzeni, to operacje te będą skutecznie realizowane równolegle. Niektóre operacje zawierają jądra wielordzeniowych jednostek CPU: zadania są tu rozdzielane na wiele operacji podrzędnych, które zostają umieszczone w następnej kolejce ewaluacji i przekazane do drugiej puli wątków, zwanej **wewnętrzoperacyjną pulą wątków** (ang. *intra-op thread pool*; jest ona współdzielona pomiędzy wszystkie jądra wielowątkowych jednostek CPU). Mówiąc krótko, wiele operacji i operacji podrzędnych może być obliczanych równolegle przez różne rdzenie procesora.

W przypadku jednostek GPU jest nieco prościej. Operacje w kolejce GPU są realizowane sekwencyjnie, ale większość z nich zawiera jądra wielowątkowych procesorów GPU, zazwyczaj implementowane przez biblioteki wymagane przez moduł TensorFlow, np. CUDA czy cuDNN. Implementacje te zawierają własne pule wątków i przeważnie wykorzystują maksymalną dostępną liczbę wątków jednostki GPU (z tego właśnie powodu nie jest tu potrzebna międzyoperacyjna pula wątków: każda operacja z góry angażuje maksymalną liczbę wątków).

Na przykład na rysunku 19.14 węzły A, B i C są operacjami źródłowymi, więc mogą zostać od razu wyliczone. Operacje A i B są umieszczone w procesorze głównym, zatem zostają przesłane do kolejki ewaluacji procesora CPU, a stamtąd do międzyoperacyjnej puli wątków i od razu przetworzone równolegle. Okazuje się, że operacja A ma jądro wielowątkowe — jego obliczenia zostają rozdzielone na trzy części i przetworzone równolegle przez wewnętrznzoperacyjną pulę wątków. Operacja C trafia do kolejki ewaluacji pierwszej karty graficznej; w tym przykładzie jądro GPU wykorzystuje bibliotekę cuDNN, która zarządza własną wewnętrznzoperacyjną pulą wątków i równolegle oblicza tę operację w wielu wątkach karty graficznej. Założymy, że operacja C zostanie ukończona jako pierwsza. Liczniki zależności dla operacji D i E będą maleć i zostaną zredukowane do wartości 0, zatem obydwie operacje trafią do kolejki ewaluacji pierwszej karty graficznej i zostaną obliczone sekwencyjnie. Operacja C zostanie obliczona tylko raz, nawet jeśli zależą od niej operacje D i E. Założymy teraz, że w następnej kolejności zostanie obliczona operacja B. Licznik zależności operacji F zostanie zmniejszony z 4 do 3, a skoro nie osiągnieśmy wartości 0, nie może ona jeszcze zostać uruchomiona. Po obliczeniu operacji A, D i E licznik ten osiąga w końcu wartość 0 i operacja F zostaje przesłana do kolejki ewaluacji procesora głównego, a następnie obliczona. Na koniec moduł TensorFlow zwraca wymagane rezultaty.

Mamy do czynienia z odrobiną magii, gdy funkcja TF modyfikuje zasób stanowy, np. zmienną: sprawia, że kolejność wykonywanych obliczeń jest zgodna z kolejnością zdefiniowaną w kodzie, nawet jeśli nie występuje jawną zależność pomiędzy instrukcjami. Na przykład jeżeli funkcja TF zawiera kolejno instrukcje `v.assign_add(1)` i `v.assign(v * 2)`, moduł TensorFlow sprawi, że zostaną one zrealizowane dokładnie w takiej kolejności.



Możesz kontrolować liczbę wątków w puli międzyoperacyjnej, jeśli wywołasz metodę `tf.config.threading.set_inter_op_parallelism_threads()`. Natomiast do wyznaczania liczby wątków puli wewnętrzoperacyjnej służy metoda `tf.config.threading.set_intra_op_parallelism_threads()`. Rozwiążanie to przydaje się wtedy, gdy nie chcesz, aby moduł TensorFlow wykorzystywał wszystkie rdzenie procesora, lub jeżeli zależy Ci na jednowątkowości¹⁷.

Masz już wszystkie elementy wymagane do zrealizowania każdej operacji na dowolnym urządzeniu i wykorzystania potęgi kart graficznych! Oto kilka spośród możliwości, jakie się przed Tobą otworzyły:

- Możesz wytrenować kilka modeli równolegle, każdy w osobnym procesorze GPU — wystarczy napisać skrypt uczący dla każdego modelu i uruchomić je równolegle za pomocą zmiennych środowiskowych `CUDA_DEVICE_ORDER` i `CUDA_VISIBLE_DEVICES`, dzięki czemu każdy skrypt widziałby tylko jedną kartę graficzną. Jest to wspaniałe rozwiązanie w przypadku strojenia hiperparametrów, ponieważ możesz w ten sposób trenować wiele modeli o różnych wartościach hiperparametrów. Jeżeli masz dwie karty graficzne w jednym komputerze i trenowanie jednego modelu na jednej karcie trwa godzinę, to uczenie równolegle dwóch modeli na dwóch osobnych jednostkach GPU również zajmie tylko godzinę. Proste!

¹⁷ Przydaje się to wtedy, gdy chcesz zapewnić idealną odtwarzalność, co wyjaśniam (na przykładzie modułu TF 1) w następującej prezentacji: <https://www.youtube.com/watch?v=Ys8ofBeR2kA>.

- Możesz wytrenować model na jednej karcie graficznej i równolegle zająć się wstępny przetwarzaniem danych za pomocą procesora głównego; metoda `prefetch()`¹⁸ obiektu `Dataset` może przygotowywać po kilka grup przykładów, dzięki czemu w razie potrzeby będą dostępne dla jednostki GPU (zob. rozdział 13.).
- Jeżeli Twój model przyjmuje na wejściu dwa obrazy i przetwarza je za pomocą dwóch sieci splotowych przed połączeniem ich wyników, to proces ten prawdopodobnie zostanie znacznie szybciej zrealizowany, jeśli umieścis każdą sieć splotową w osobnej karcie graficznej.
- Możesz utworzyć wydajny zespół — wystarczy umieścić modele wytrenowane na różne sposoby w osobnych procesorach GPU, dzięki czemu znacznie szybciej otrzymasz prognozy poszczególnych składników zespołu, a zatem również prognozę końcową.

No dobrze, ale gdybyśmy chcieli **wytrenować** pojedynczy model za pomocą wielu procesorów graficznych?

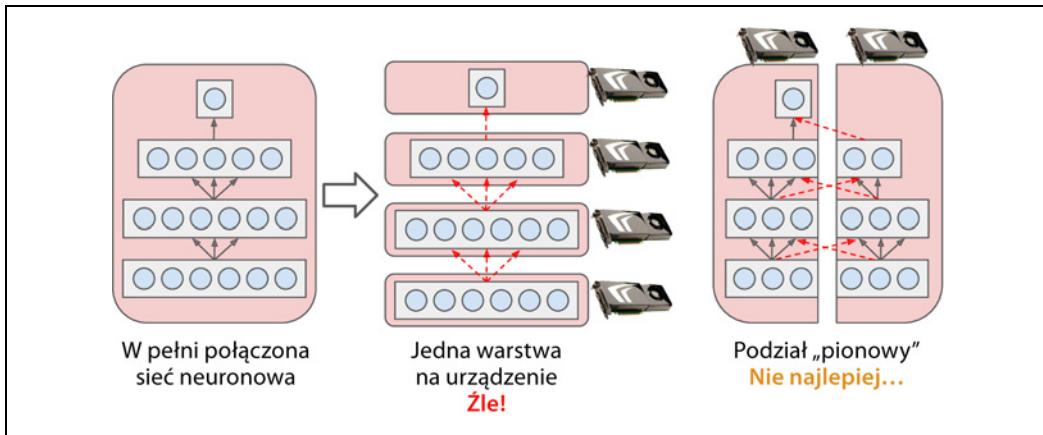
Uczenie modeli za pomocą wielu urządzeń

Istnieją dwie podstawowe strategie uczenia pojedynczego modelu za pomocą wielu urządzeń: **zrównoleglanie modelu** (ang. *model parallelism*), zgodnie z którym model zostaje rozdzielony pomiędzy urządzenia, i **zrównoleglanie danych** (ang. *data parallelism*), gdzie model zostaje powielony na poszczególne urządzenia i każda taka replika jest uczona za pomocą podzbioru danych. Przyjrzyjmy się uważnie obydwu opcjom, zanim przystąpimy do uczenia modelu za pomocą wielu procesorów graficznych.

Zrównoleglanie modelu

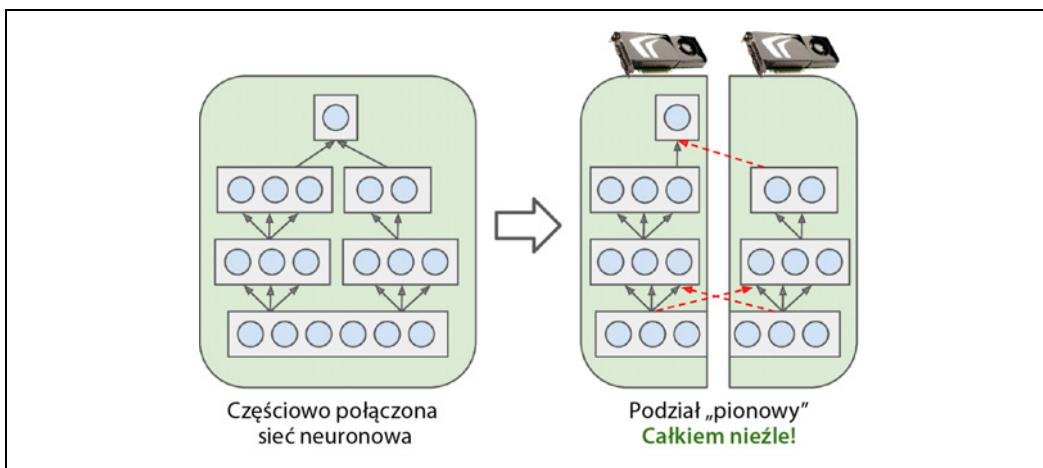
Na razie uruchamialiśmy każdą sieć neuronową na jednym urządzeniu. A gdybyśmy chcieli, żeby sieć była przetwarzana przez wiele urządzeń? Proces ten wymaga rozdzielenia modelu na mniejsze składowe, z których każda będzie używana przez inne urządzenie. Niestety stosowanie takiego zrównoleglania modelu wiąże się z pewnymi trudnościami i w dużej mierze zależy od architektury sieci neuronowej. W przypadku w pełni połączonych sieci rozwiązanie to nie przynosi wielu korzyści (rysunek 19.15). Intuicja podpowiada, że najłatwiej jest podzielić model, umieszczając każdą warstwę sieci na innym urządzeniu, nie ma to jednak sensu, ponieważ każda wyższa warstwa musi czekać na wynik z niższej. Może więc warto podzielić ją „pionowo” — przykładowo lewą połowę sieci umieścić na jednym urządzeniu, a prawą na innym? To już jest nieco lepsze rozwiązanie, ponieważ rzeczywiście obydwie połowy mogą pracować równolegle, pamiętaj jednak, że w wyższej warstwie każda połowa wymaga wyników pochodzących z obydwu części sieci, więc bylibyśmy świadkami wzmożonej komunikacji pomiędzy urządzeniami (przerywane strzałki). Zjawisko to prawie na pewno całkowicie zniweluje korzyści płynące z przetwarzania równoległego, ponieważ komunikacja pomiędzy urządzeniami jest dość wolna (zwłaszcza jeżeli znajdują się one na osobnych komputerach).

¹⁸ W czasie gdy pisałem tę książkę, metoda ta wstępnie wczytywała dane jedynie do pamięci operacyjnej procesora głównego, ale za pomocą funkcji `tf.data.experimental.prefetch_to_device()` możesz wstępnie wczytywać dane i przesyłać je do wybranego urządzenia, dzięki czemu jednostka GPU nie będzie tracić czasu w oczekiwaniu na transfer danych.



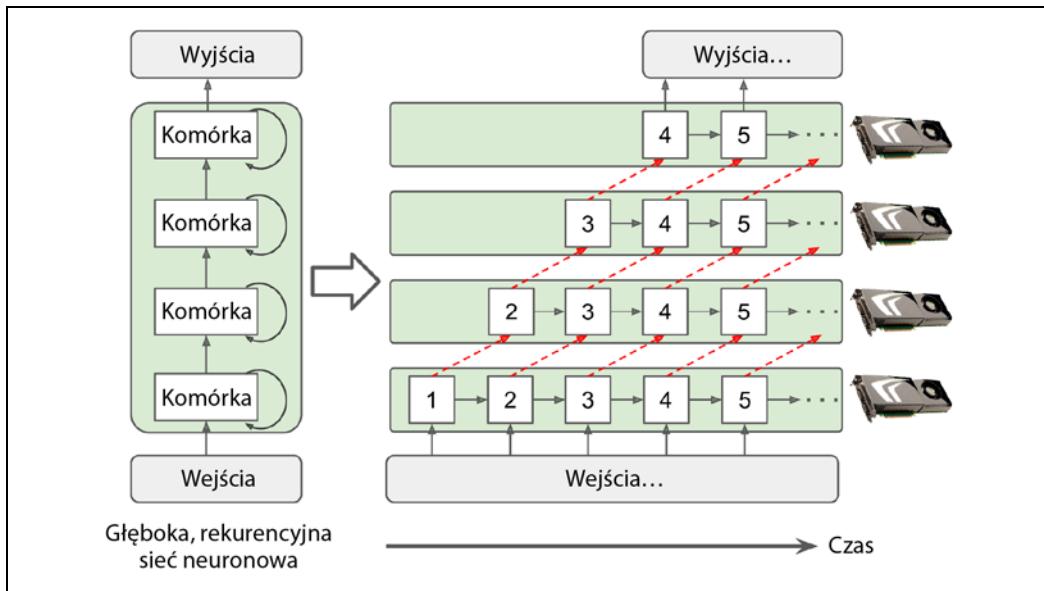
Rysunek 19.15. Rozdzielanie w pełni połączonej sieci neuronowej

Niektóre architektury sieci, na przykład splotowe (zob. rozdział 14.), zawierają warstwy tylko częściowo połączone z niższymi warstwami, zatem znacznie łatwiej i skuteczniej można je podzielić i rozmieścić na różnych urządzeniach (rysunek 19.16).



Rysunek 19.16. Rozdzielanie częściowo połączonej sieci neuronowej

Głębokie rekurencyjne sieci neuronowe (zob. rozdział 15.) można rozdzielać nieco skuteczniej pomiędzy wiele procesorów graficznych. Jeśli rozdzielimy taką sieć „poziomo” poprzez umieszczenie każdej warstwy w osobnym urządzeniu, to w pierwszym takcie będzie aktywne tylko jedno urządzenie (będzie pracować nad pierwszą wartością sekwencji), w drugim takcie będą to już dwa urządzenia (druga warstwa będzie przetwarzarwać wyniki warstwy pierwszej, uzyskane dla pierwszej wartości, natomiast pierwsza warstwa będzie przetwarzarła drugą wartość), a do chwili dotarcia sygnału do warstwy wyjściowej zostaną uaktywnione już wszystkie urządzenia i będą pracować jednocześnie (rysunek 19.17). Również tutaj mamy do czynienia ze znaczną komunikacją pomiędzy urządzeniami, ale każda komórka może być bardzo rozbudowana, dlatego zysk z równoczesnego przetwarzania wielu komórek często (teoretycznie) przeważa nad opóźnieniami wynikającymi z komunikacji. W praktyce jednak o wiele szybciej działa standardowy stos warstw LSTM przetwarzanych przez jeden procesor graficzny.



Rysunek 19.17. Dzielenie głębokiej rekurencyjnej sieci neuronowej

Mówiąc krótko, zrównoleglanie modelu pozwala przyspieszyć uczenie i działanie niektórych rodzajów sieci neuronowych, ale nie wszystkich, a do tego wymaga specyficznego podejścia i strojenia, na przykład sprawienia, żeby urządzenia najczęściej komunikujące się ze sobą były umieszczone w jednym komputerze¹⁹. Przyjrzyjmy się o wiele prostszemu i zazwyczaj skuteczniejszemu rozwiążaniu: zrównoleglaniu danych.

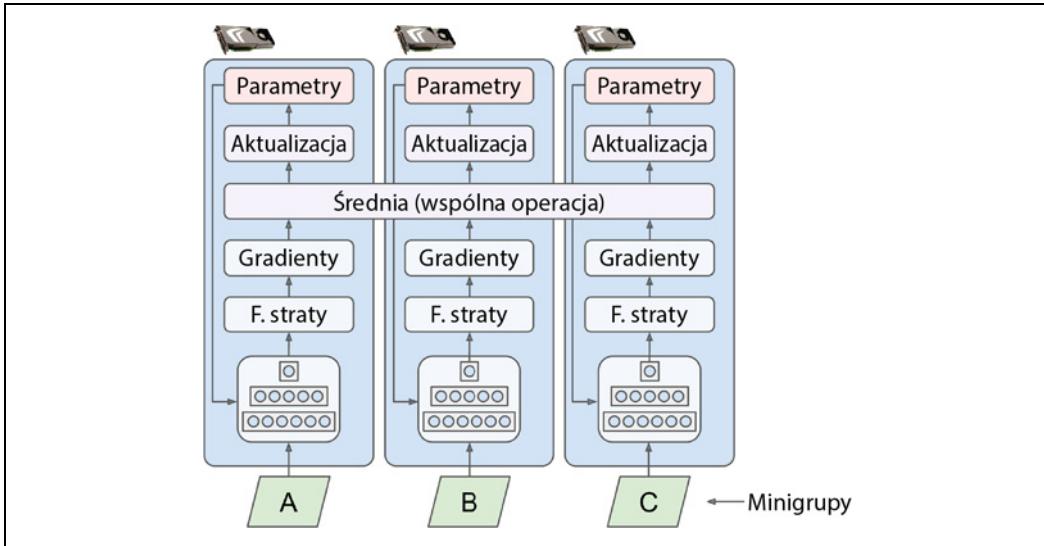
Zrównoleglanie danych

Innym sposobem równoległego przetwarzania procesu uczenia sieci uczącej jest replikowanie jej na każde urządzenie i równoczesne uruchomienie algorytmu uczącego we wszystkich replikach za pomocą różnych minigrup. Gradienty obliczone przez każdą replikę zostają następnie uśrednione, a uzyskany wynik służy do aktualizacji parametrów modelu.. Mamy tu do czynienia ze **zrównoleganiem danych** (ang. *data parallelism*). Istnieje wiele odmian tej koncepcji, dlatego skoncentrujmy się na najważniejszych.

Zrównoleglanie danych za pomocą strategii duplikowania

Bodaj najprostszym rozwiązaniem jest całkowite zduplikowanie wszystkich parametrów modelu na wszystkie karty graficzne i przeprowadzanie takich samych aktualizacji parametrów w każdej jednostce GPU. W ten sposób wszystkie repliki są zawsze identyczne. Jest to tak zwana **strategia duplikowania** (ang. *mirrored strategy*) i okazuje się całkiem skuteczna, zwłaszcza gdy korzystasz tylko z jednego komputera (rysunek 19.18).

¹⁹ Jeżeli chcesz uzyskać więcej informacji na temat zrównoleglania modelu, sprawdź bibliotekę Mesh TensorFlow (<https://github.com/tensorflow/mesh>).



Rysunek 19.18. Zrównoleglanie danych za pomocą strategii duplikowania

Największe wyzwanie w tej strategii stanowi skuteczne obliczenie średniej ze wszystkich gradientów i procesorów graficznych, a następnie rozesłanie rezultatu do wszystkich układów GPU. Możemy to zrobić za pomocą algorytmu **AllReduce**, należącego do klasy algorytmów, dzięki którym wiele węzłów współpracuje ze sobą w celu wydajnego przeprowadzenia operacji redukowania (np. obliczenia średniej, sumy czy wartości maksymalnej), przy jednoczesnym zagwarantowaniu dostarczenia wszystkim węzłom tych samych rezultatów. Na szczęście istnieją gotowe implementacje tego algorytmu, o czym przekonamy się już wkrótce.

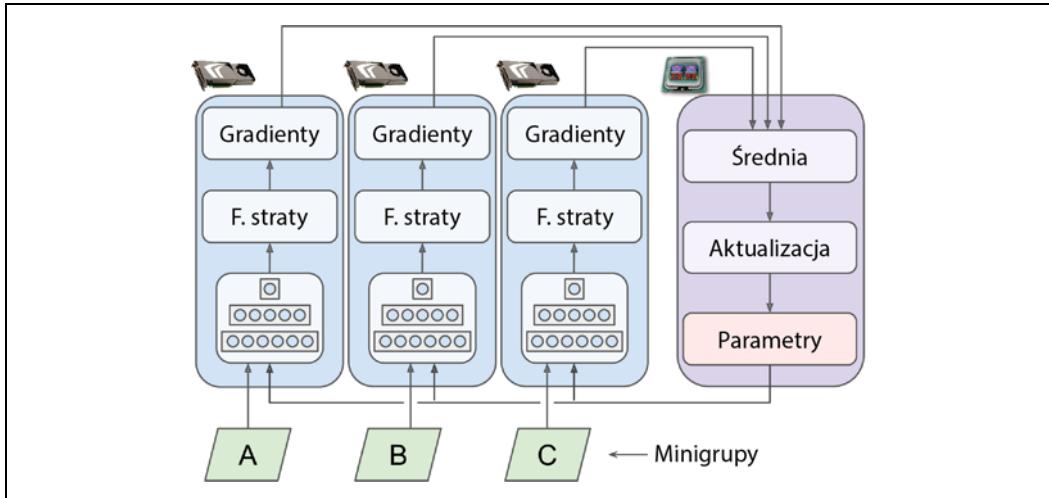
Zrównoleglanie danych za pomocą scentralizowanych parametrów

Innym sposobem jest przechowywanie parametrów modelu poza kartami graficznymi wykonującymi obliczenia (tzw. **roboczymi grupami zadań** — ang. *workers*), na przykład w procesorze głównym (rysunek 19.19). W konfiguracji rozproszonej możesz umieścić wszystkie parametry na co najmniej jednym serwerze wyposażonym wyłącznie w jednostki CPU, zwany **serwerem parametrów** (ang. *parameter server*), którego jedynym zadaniem jest przechowywanie i aktualizowanie parametrów.

Strategia duplikowania narzuca synchroniczne aktualizowanie wag we wszystkich jednostkach GPU, natomiast strategia scentralizowana umożliwia zarówno synchroniczne, jak i asynchroniczne aktualizowanie. Sprawdźmy wady i zalety każdego z tych rozwiązań.

Aktualizacje synchroniczne

W przypadku **aktualizacji synchronicznych** (ang. *synchronous updates*) aggregator czeka, aż wszystkie gradienty będą dostępne przed wyliczeniem z nich średniej, po czym przekazuje ją do optymalizatora, który zaktualizuje parametry modelu. Po obliczeniu gradientu replika musi czekać z koleją na zaktualizowane parametry, zanim będzie mogła skorzystać z kolejnej minigrupy. Wada tego rozwiązania jest



Rysunek 19.19. Zrównoleglanie danych za pomocą scentralizowanych parametrów

taka, że niektóre urządzenia mogą być wolniejsze od innych, dlatego pozostałe muszą na nie czekać. Poza tym wartości parametrów zostaną rozesłane niemal równocześnie do wszystkich urządzeń (tuż po użyciu gradientów), co może zapełnić przepustowość łączna serwera parametrów.



Aby skrócić czas oczekiwania w każdym przebiegu, możemy zignorować gradienty pochodzące z kilku najwolniejszych replik (zazwyczaj około 10% replik). Możemy na przykład korzystać z 20 replik, ale w każdym przebiegu gromadzić gradienty wyłącznie z 18 najszybszych, a wyniki pozostałych dwóch zwyczajnie zignorować. Po zaktualizowaniu parametrów 18 replik może natychmiast zacząć dalszą pracę bez konieczności czekania na dwa najbardziej powolne urządzenia. O takiej konfiguracji mówimy, że składa się z 18 replik i **dwoch replik zapasowych**²⁰.

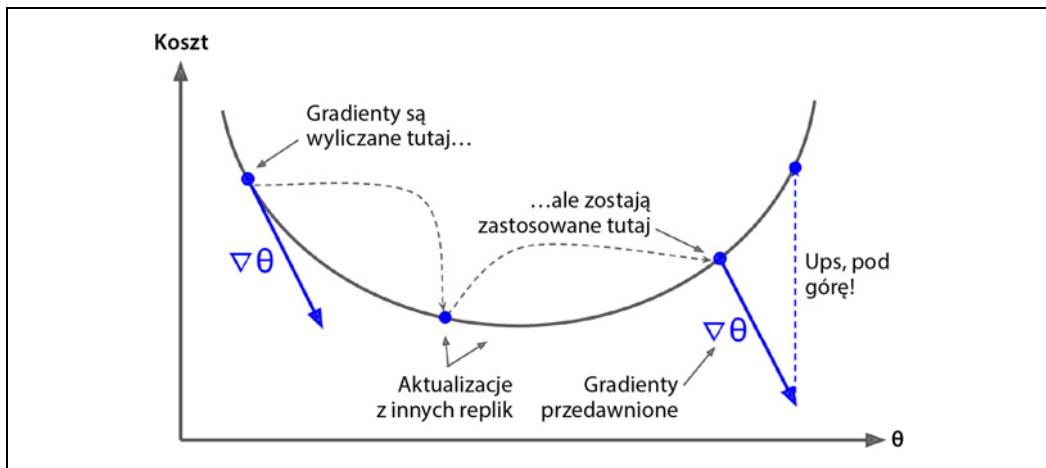
Aktualizacje asynchronousne

Jeśli chodzi o **aktualizacje asynchronousne** (ang. *asynchronous updates*), replika po wyliczeniu gradientów natychmiast wykorzystuje je do zaktualizowania parametrów modelu. Nie ma tu etapu gromadzenia gradientów (znika etap *średnia* widoczny na rysunku 19.19), a także synchronizacji. Każda replika działa niezależnie od pozostałych. Skoro nie ma oczekiwania na pozostałe repliki, mechanizm ten wykonuje więcej przebiegów uczących w ciągu minuty. Ponadto parametry muszą wprawdzie w każdym przebiegu być kopiowane na wszystkie urządzenia, ale nie jest to wykonywane równocześnie, dlatego istnieje mniejsza szansa na nasycenie przepustowości łączna.

Zrównoleglanie danych przy użyciu aktualizacji asynchronousnych stanowi atrakcyjny wybór z powodu prostoty, braku opóźnień synchronizacji i lepszego wykorzystania przepustowości. Jest jednak

²⁰ Określenie to jest nieco mylące, bo może sugerować, że te dwie repliki są wyjątkowe i nic nie robią. W rzeczywistości wszystkie repliki są sobie równoważne: wszystkie „starają się” być jak najszybsze i w każdym przebiegu para „przegranych” jest zazwyczaj inną (chyba że najwolniejsze urządzenia znacznie odstają mocą obliczeniową od pozostałych). Oznacza to jednak, że w przypadku awarii serwera proces uczenia będzie bezproblemowo kontynuowany.

niemal zdumiewające, że ta metoda w ogóle działa! W istocie do czasu zakończenia obliczania gradientów na podstawie wartości parametrów przez daną replikę parametry te zdążą być kilkakrotnie zaktualizowane przez inne repliki (średnio $N-1$ razy, jeśli mamy N replik) i wcale nie ma pewności, że wyliczone gradienty będą ciągle skierowane w prawidłową stronę (rysunek 19.20). Zdezaktualizowane gradienty, nazywane **gradientami przedawnionymi** (ang. *stale gradients*), mogą znacznie spowolnić proces uzyskiwania zbieżności, wprowadzać zaszumienie i wahania (krzywa uczenia może zawierać chwilowe oscylacje), a nawet prowadzić do rozbieżności algorytmu.



Rysunek 19.20. Gradienty przedawnione podczas stosowania aktualizacji asynchronicznych

Istnieje kilka sposobów na ograniczenie zjawiska gradientów przedawnionych:

- zmniejszenie współczynnika uczenia;
- ignorowanie lub zmniejszanie wartości gradientów przedawnionych;
- zmodyfikowanie rozmiaru minigrupy;
- korzystanie w kilku pierwszych epokach tylko z jednej repliki (tzw. **faza rozgrzewki** — ang. *warmup phase*) — gradienty przedawnione zazwyczaj są najgroźniejsze na początku procesu uczenia, gdy gradienty mają jeszcze duże wartości, a parametry nie zostały upłasowane w pobliżu minimum funkcji kosztu, dlatego różne repliki mogą modyfikować parametry w zgoła odmiennych kierunkach.

W publikacji przygotowanej w 2016 roku przez zespół Google Brain (<https://arxiv.org/abs/1604.00981>)²¹ zmierzono wydajność różnych metod i stwierdzono, że najlepsze wyniki uzyskuje zrównoleglanie danych za pomocą aktualizacji synchronicznych i kilku zapasowych replik — model nie tylko szybciej osiąga zbieżność, lecz również jest skuteczniejszy. Pamiętaj jednak, że ta dziedzina jest bardzo aktywnie analizowana, dlatego nie należy jeszcze wykluczać możliwości aktualizacji asynchronicznych.

²¹ Jianmin Chen i in., *Revisiting Distributed Synchronous SGD*, arXiv preprint arXiv:1604.00981 (2016).

Nasycenie przepustowości

Bez względu na to, czy korzystasz z aktualizacji synchronicznych czy asynchronicznych, zrównoleglane danych wymaga przekazywania parametrów modelu z serwera parametrów do wszystkich replik na początku każdego przebiegu uczenia, a także, w drugą stronę, gradientów na końcu każdej epoki. Podobnie w przypadku strategii duplikowania: gradienty uzyskane przez każdą jednostkę GPU muszą zostać przesłane do wszystkich pozostałych procesorów graficznych. Oznacza to niestety, że istnieje moment, w którym dodawanie kolejnych kart graficznych przestanie poprawiać wydajność, ponieważ czas poświęcony na przesyłanie danych z pamięci operacyjnej karty graficznej (i ewentualnie przez sieć w konfiguracji rozproszonej) oraz do niej zniweluje szybkość uzyskaną poprzez rozdzielenie obciążenia obliczeniowego. W takiej sytuacji każda kolejna karta graficzna będzie tylko dodatkowo zapełniała przepustowość łącza i dodatkowo spowalniała proces uczenia.



Zazwyczaj o wiele lepiej jest trenować względnie małe modele korzystające z dużych zbiorów danych za pomocą jednego komputera wyposażonego w wydajną kartę graficzną o dużej przepustowości pamięci operacyjnej.

Problem nasycenia jest poważniejszy w przypadku dużych, gęstych modeli, gdyż przesyłanych jest w nich mnóstwo wartości parametrów i gradientów. Nie jest on tak odczuwalny w małych modelach (ale również zysk z przetwarzania równoległego okazuje się niewielki), a także w modelach rzadkich, gdyż gradienty najczęściej mają w nich wartość 0, co znacznie ułatwia komunikację pomiędzy urządzeniami. Jeff Dean, inicjator i kierownik projektu Google Brain, stwierdził, że szybkość obliczeń rozproszonych (modele gęste) pomiędzy 50 kart graficznych jest wyższa 25 – 40 razy, a 300 razy dla rzadszych modeli wytrenowanych za pomocą 500 kart graficznych (<https://www.youtube.com/watch?v=sUzQpd-Ku4o>). Jak widać, modele rzadkie rzeczywiście radzą sobie znacznie lepiej. Oto kilka konkretnych przykładów:

- neuronowe tłumaczenie maszynowe: 6-krotny wzrost szybkości na 8 kartach graficznych;
- algorytm Inception/ImageNet: 32-krotny wzrost szybkości na 50 kartach graficznych;
- system RankBrain: 300-krotny wzrost szybkości na 500 kartach graficznych.

Liczba kart graficznych przekraczająca kilkadziesiąt dla modeli gęstych lub kilkaset dla modeli rzadkich wprowadza nasycenie przepustowości łącza będące przyczyną spadku wydajności. Wielu badaczy koncentruje się na rozwiązaniu tego problemu (badają architektury równorzędne zamiast scentralizowanych serwerów parametrów, stosują kompresję straną modeli, optymalizują harmonogram komunikacji urządzeń itd.), dlatego w ciągu kilku lat powinniśmy być świadkami postępu w równoległym przetwarzaniu sieci neuronowych.

Tymczasem zaś możesz spróbować zmniejszyć problem nasycenia poprzez zastąpienie wielu słabysznych kart graficznych kilkoma wydajniejszymi i umieszczenie jednostek GPU na kilku optymalnie połączonych ze sobą serwerach. Możesz także zmienić precyzję wartości parametrów z 32 bitów (`tf.float32`) na 16 bitów (`tf.bfloat16`). Zmniejszysz w ten sposób o połowę ilość przesyłanych danych bez znacznego wpływu na szybkość uzyskiwania zbieżności lub wydajność modelu. Jeżeli zaś korzystasz ze scentralizowanych parametrów, możesz rozdzielić je pomiędzy wiele serwerów

parametrów: każdy dodatkowy serwer parametrów zmniejsza obciążenie sieciowe na każdym z pozostałych i ogranicza ryzyko nasycenia przepustowości.

No dobrze, wytrenujmy teraz model za pomocą wielu kart graficznych!

Uczenie wielkoskalowe za pomocą interfejsu strategii rozpraszania

Wiele modeli można całkiem skutecznie trenować za pomocą pojedynczej karty graficznej, a nawet jednostki CPU. Jeżeli jednak proces uczenia przebiega zbyt powoli, możesz spróbować rozdysponować go pomiędzy wiele procesorów graficznych podłączonych do jednego komputera. Jeżeli to ciągle za mało, wypróbuj wydajniejsze karty graficzne albo podłącz kilka dodatkowych jednostek GPU. W przypadku skomplikowanych obliczeń (np. mnożenia dużych macierzy) znacznie lepiej będą się sprawdzać wydajne karty graficzne; możesz również wypróbować możliwości jednostek TPU dostępnych w usłudze Google Cloud AI Platform, które spisują się w takich sytuacjach jeszcze lepiej. Jeżeli jednak już nie masz fizycznej możliwości dołączania kolejnych kart graficznych do komputera, a procesory tensorowe nie odpowiadają Ci jakiegoś powodu (np. struktura modelu nie pozwala wykorzystywać pełni możliwości procesorów TPU albo wolisz korzystać z własnej infrastruktury sprzętowej), to możesz spróbować wyuczyć go na kilku serwerach, z których każdy wyposażony jest w kilka kart graficznych (jeśli to ciągle nie wystarczy, możesz spróbować zaimplementować zrównoleglenie modelu, ale wymaga to o wiele większego wysiłku). W dalszej części rozdziału dowiesz się, jak można trenować modele na dużą skalę: najpierw zacznijmy od wykorzystania wielu kart graficznych (lub tensorowych) umieszczonej w jednym komputerze, a potem przejdziemy do obsługi jednostek GPU na wielu urządzeniach.

Na szczęście moduł TensorFlow zawiera bardzo prosty API, który zajmuje się wszystkimi zawiłościami: **interfejs strategii rozpraszania** (ang. *Distribution Strategies API*). Aby wytrenować model Keras na wszystkich dostępnych kartach graficznych (na razie ograniczymy się do jednego komputera) za pomocą strategii duplikowania, utworzymy najpierw obiekt `MirroredStrategy`, wywołamy metodę `scope()`, co pozwoli określić kontekst rozpraszania, po czym umieścimy w tym kontekście fazy tworzenia i kompilowania modelu. Pozostanie nam już wtedy tylko wywołanie metody `fit()` modelu w tradycyjny sposób:

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile(...)

batch_size = 100 # Wartość ta musi być podzielna przez liczbę replik
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Interfejs tf-keras jest dostosowany do operacji rozproszonych, dlatego dzięki kontekstowi `MirroredStrategy` wie, że ma powieścić wszystkie operacje i zmienne na wszystkie dostępne karty graficzne. Zwróć uwagę, że metoda `fit()` automatycznie rozdzieli każdą grupę danych uczących pomiędzy wszystkie repliki, dlatego jest tak ważne, aby rozmiar grupy stanowił wielokrotność liczby replik. I to wszystko! Proces uczenia będzie trwał zazwyczaj znacznie krócej w porównaniu do trenowania za pomocą jednej karty graficznej, a my prawie nie musielibyśmy modyfikować kodu.

Po zakończeniu uczenia modelu możesz skutecznie używać go do uzyskiwania prognoz: wywołaj metodę `predict()`, a grupa zostanie automatycznie rozdzielona pomiędzy repliki, co spowoduje równoległe obliczanie prognoz (również w tym przypadku rozmiar grupy musi być podzielny przez liczbę replik). Jeżeli wywołasz metodę `save()` modelu, zostanie zapisany jako tradycyjny model, a **nie** jako model rozproszony pomiędzy wiele replik. Oznacza to, że po jego wczytaniu możesz go używać w standardowy sposób na pojedynczym urządzeniu (domyślnie GPU 0 lub CPU w przypadku braku kart graficznych). Jeżeli chcesz wczytać model tak, aby był przetwarzany przez wszystkie dostępne urządzenia, musisz wywołać metodę `keras.models.load_model()` wewnątrz kontekstu rozpraszania:

```
with distribution.scope():
    mirrored_model = keras.models.load_model("moj_model_mnist.h5")
```

Jeżeli chcesz używać tylko podzbioru dostępnych kart graficznych, wystarczy przekazać ich listę konstruktorowi klasy `MirroredStrategy`:

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

Domyślnie klasa `MirroredStrategy` wykorzystuje bibliotekę **NCCL** (NVIDIA Collective Communications Library) do przeprowadzania operacji uśredniania za pomocą algorytmu AllReduce, ale możesz to zmienić poprzez wyznaczenie w argumencie `cross_device_ops` instancji klasy `tf.distribute.HierarchicalCopyAllReduce` lub klasy `tf.distribute.ReductionToOneDevice`. Domyślny wybór biblioteki NCCL bazuje na klasie `tf.distribute.NcclAllReduce`, która jest zazwyczaj szybsza, zależy jednak od liczby i rodzajów kart graficznych, dlatego warto też wypróbować rozwiązania alternatywne²².

Jeżeli chcesz wypróbować zrównoleglanie danych ze scentralizowanymi parametrami, zastąp kontekst `MirroredStrategy` kontekstem `CentralStorageStrategy`:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

Możesz ewentualnie wstawić argument `compute_devices` do wyznaczenia listy urządzeń pełniących funkcję roboczych grup zadań (domyślnie są to wszystkie dostępne karty graficzne); z kolei za pomocą argumentu `parameter_device` możesz wyznaczyć urządzenie, w którym będą przechowywane parametry (domyślnie procesor główny albo karta graficzna, jeżeli dostępna jest tylko jedna).

Wy trenujmy teraz model za pomocą klastra TensorFlow!

Uczenie modelu za pomocą klastra TensorFlow

Klaster TensorFlow (ang. *TensorFlow cluster*) to grupa równolegle przetwarzanych (zwykle na osobnych komputerach) procesów TensorFlow komunikujących się ze sobą w celu ukończenia jakiegoś zadania, na przykład wytrenowania lub uruchomienia sieci neuronowej. Każdy proces w klastrze TF nosi nazwę **zadania** (ang. *task*) lub **serwera TF** (ang. *TF server*). Każde zadanie zawiera własny adres IP i port i przynależy do określonego typu, zwanego **rolą** (ang. *role*) albo **grupą zadań** (ang. *job*).

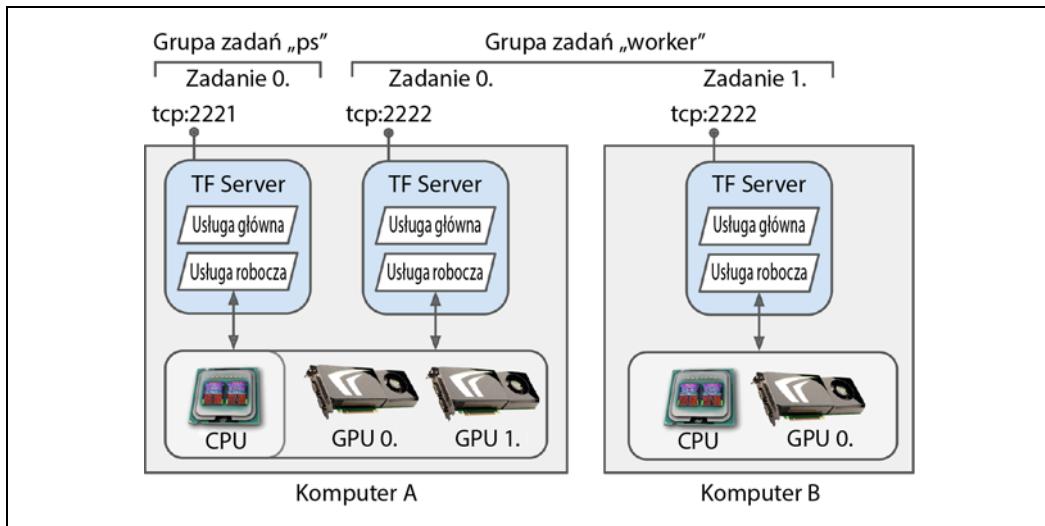
²² Szczegółowy opis algorytmu AllReduce znajdziesz w znakomitym wpisie Yuichiro Ueno (<https://homl.info/uenopost>), a także na stronie poświęconej skalowaniu za pomocą biblioteki NCCL (<https://homl.info/ncclalgo>).

Wyróżniamy cztery typy zadań: robocza grupa zadań (worker), serwer główny (chief), serwer parametrów (ps; ang. *parameter server*) i ewaluator (evaluator):

- Każda robocza **grupa zadań** przeprowadza obliczenia, zazwyczaj na komputerze wyposażonym w co najmniej jedną kartę graficzną.
- **Serwer główny** również przeprowadza obliczenia (jak robocza grupa zadań), ale zajmuje się też dodatkowymi czynnościami, takimi jak generowanie komunikatów zdarzeń TensorFlow czy zapisywanie punktów kontrolnych. W każdym klastrze znajduje się jeden serwer główny. Jeżeli nie został wyznaczony żaden serwer główny, rola ta przypada pierwszej roboczej grupie zadań.
- **Serwer parametrów** zajmuje się jedynie wartościami zmiennych i mieści się zwykle na komputerze wyposażonym wyłącznie w jednostkę CPU. Ten typ zadania jest wykorzystywany tylko w klasie ParameterServerStrategy.
- Ewaluator zajmuje się oceną modelu.

Przed uruchomieniem klastra TensorFlow należy go zdefiniować. Oznacza to określenie adresu IP, portu i typu każdego zadania. Na przykład poniższa **specyfikacja klastra** wyznacza klasterek trzyzadaniowy (dwie robocze grupy zadań i jeden serwer parametrów) (rysunek 19.21). Specyfikacja klastra to słownik zawierający po jednym kluczu na każdą grupę zadań, natomiast jego wartości tworzą listy adresów zadań (*IP:port*):

```
cluster_spec = {
    "worker": [
        "komputer-a.przyklad.com:2222", # /job:worker/task:0
        "komputer-b.przyklad.com:2222" # /job:worker/task:1
    ],
    "ps": ["komputer-a.przyklad.com:2221"] # /job:ps/task:0
}
```



Rysunek 19.21. Klasterek TensorFlow

Zasadniczo na każdy komputer przypada jedno zadanie, ale jak widać na powyższym przykładzie, możesz skonfigurować wiele zadań na jednym komputerze (jeżeli korzystają z tej samej karty graficznej, zapewnij im równomierny przydział jej pamięci operacyjnej).



Domyślnie zadania w klastrze mogą się ze sobą komunikować, dlatego skonfiguruj odpowiednio zaporę sieciową, tak aby umożliwiała komunikację pomiędzy tymi komputerami poprzez wyznaczone porty (zazwyczaj ułatwiasz sobie życie, jeśli na wszystkich komputerach korzystasz z tego samego portu).

W momencie uruchamiania zadania musisz podać mu specyfikację klastra, a także wyznaczyć typ i indeks (np. *worker 0*). Najprostszym sposobem zdefiniowania wszystkich elementów jednocześnie (specyfikacji klastra oraz typu i indeksu bieżącego zadania) jest wyznaczenie zmiennej środowiskowej `TF_CONFIG` przed uruchomieniem klastra TensorFlow. Musi mieć ona postać słownika kodowanego w formacie JSON przechowującego specyfikację klastra (klucz "cluster") oraz typ i indeks bieżącego zadania (klucz "task"). Na przykład ta zmienna środowiskowa `TF_CONFIG` wykorzystuje zdefiniowany przez nas wcześniej klaster i za jej pomocą wyznaczamy pierwszą roboczą grupę zadań jako zadanie do uruchomienia:

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```



Najczęściej chcemy definiować zmienną środowiskową `TF_CONFIG` poza Pythonem, dlatego kod nie musi zawierać typu i indeksu bieżącego zadania (dzięki temu możemy wykorzystywać ten sam kod w przypadku wszystkich roboczych grup zadań).

Wytrenujmy teraz model w klastrze! Zaczniemy od strategii duplikowania, gdyż jest zaskakująco prosta! Najpierw musimy odpowiednio zdefiniować zmienną środowiskową `TF_CONFIG` dla każdego zadania. Nie korzystamy z serwera parametrów, dlatego należy usunąć klucz "ps" w specyfikacji klastra. Zasadniczo chcemy, aby na każdym komputerze mieściła się jedna robocza grupa zadań. Upewnij się, że dla każdego zadania wyznaczyłaś/wyznaczyłeś unikatowy indeks. Teraz możesz uruchomić ten oto kod w każdej roboczej grupie zadań:

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile(...)

batch_size = 100 # Wartość ta musi być podzielna przez liczbę replik
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Zgadza się, dokładnie z tego samego kodu korzystaliśmy wcześniej, tym razem jednak korzystamy z klasy `MultiWorkerMirroredStrategy` (prawdopodobnie w przyszłych wersjach klasa `MirroredStrategy` będzie obsługiwać zarówno przypadki wykorzystywania pojedynczych komputerów, jak i całych klastrów). Gdy uruchomisz ten skrypt w pierwszych roboczych grupach zadań, pozostaną zabloko-

wane w fazie algorytmu AllReduce, ale tuż po uruchomieniu ostatniej roboczej grupy zadań rozpoczęcie się proces uczenia i zauważysz, że wszystkie te serwery robią postępy z taką samą szybkością (ponieważ są synchronizowane w każdym kroku).

W tej strategii rozpraszania możesz wybrać jedną z dwóch implementacji algorytmu AllReduce: wariant pierścieniowy, wykorzystujący protokół gRPC do komunikacji sieciowej, a także implementację NCCL. Dobór algorytmu zależy od liczby roboczych grup zadań, liczby i rodzajów procesorów GPU, a także sieci. Moduł TensorFlow domyślnie wykorzysta algorytmy heurystyczne do wybrania odpowiedniego rozwiązania, ale możesz wymusić stosowanie wybranego przez siebie poprzez przekazanie opcji `CollectiveCommunication.RING` lub `CollectiveCommunication.NCCL` (z interfejsu `tf.distribute.experimental`) do konstruktora strategii.

Jeżeli wolisz zaimplementować asynchroniczne zrównoleglanie danych wykorzystujące serwer parametrów, zmień strategię na `ParameterServerStrategy`, dodaj przynajmniej jeden serwer parametrów i skonfiguruj odpowiednio zmienną środowiskową `TF_CONFIG`. Zwróć uwagę, że robocze grupy zadań będą działać asynchronicznie, ale repliki na każdym z tych serwerów będą przetwarzane synchronicznie.

Na koniec, jeżeli masz dostęp do procesorów tensorowych na platformie Google Cloud (<https://cloud.google.com/tpu/>), możesz utworzyć `TPUStrategy` w następujący sposób (i korzystać z niej jak z innych strategii):

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```



Pracownicy naukowi mogą bezpłatnie korzystać z mocy jednostek TPU (szczegóły znajdziesz pod adresem <https://www.tensorflow.org/tfrc>).

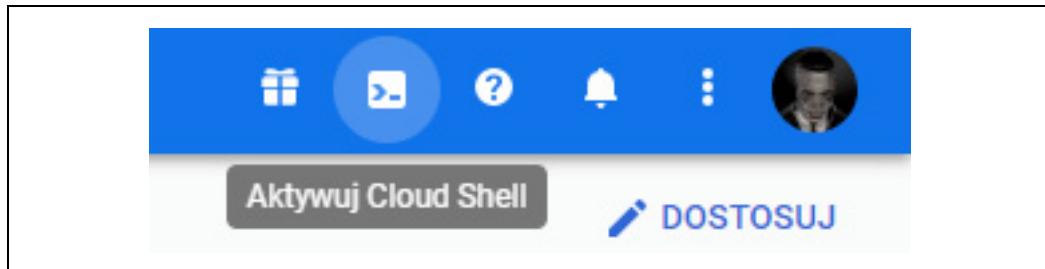
Możesz teraz trenować modele za pomocą wielu kart graficznych i serwerów. Jeżeli chcesz trenować duży model, będziesz potrzebować wielu procesorów GPU rozmieszczonego na wielu serwerach, co oznacza albo zakup znacznej ilości sprzętu, albo zarządzanie dużą liczbą maszyn wirtualnych. W wielu przypadkach mniej pracochłonne i kosztowne jest wykorzystanie usługi rozproszonej, która w razie potrzeby zajmie się za Ciebie zapewnieniem całej infrastruktury i zarządzaniem nią. Zobaczmy, jak może nam w tym pomóc platforma GCP.

Realizowanie dużych grup zadań uczenia za pomocą usługi Google Cloud AI Platform

Jeżeli postanowisz korzystać z usługi Google AI Platform, możesz wdrożyć grupę zadań uczenia zdefiniowaną za pomocą tego samego kodu, jakiego użyłabyś/użyłbyś we własnym klastrze TF, a platforma zajmie się dostarczaniem i konfigurowaniem wymaganej przez Ciebie liczby maszyn wirtualnych wyposażonych w procesory GPU (w zakresie limitu oczywiście).

Aby uruchomić grupę zadań, musisz zaopatrzyć się w narzędzie wiersza poleceń `gcloud` z zestawu Google Cloud SDK (<https://cloud.google.com/sdk/>). Możesz zainstalować ten zestaw na własnym

komputerze albo skorzystać z usługi Google Cloud Shell, dostępnej na platformie GCP. Możesz korzystać z tego terminalu bezpośrednio w przeglądarce — działa on na darmowej maszynie wirtualnej Linux VM (Debian), na której pakiet SDK jest już zainstalowany i skonfigurowany. Dostęp do terminalu Cloud Shell możesz uzyskać z dowolnego miejsca platformy GCP — wystarczy kliknąć ikonę *Aktywuj Cloud Shell* w prawym górnym rogu ekranu (rysunek 19.22).



Rysunek 19.22. Aktywowanie terminalu Google Cloud Shell

Jeżeli wolisz korzystać z zestawu SDK na własnym komputerze, to po jego zainstalowaniu musisz go zainicjalizować za pomocą polecenia `gcloud init`: musisz zalogować się w platformie GCP i przydzielić dostęp do Twoich zasobów GCP, a następnie wybrać projekt GCP, którego chcesz używać (jeżeli masz ich większą liczbę), oraz region, w którym ma być uruchomiona grupa zadań. Polecenie `gcloud` daje Ci dostęp do wszystkich funkcji GCP, w tym do opisanych na początku rozdziału.

Nie musisz za każdym razem korzystać z interfejsu sieciowego — możesz pisać skrypty, które uruchamiają i zatrzymują maszyny wirtualne za Ciebie, wdrażają modele lub wykonują dowolną inną czynność dostępną z poziomu platformy GCP.

Zanim uruchomisz grupę zadań uczenia, musisz najpierw napisać kod uczenia w taki sam sposób, jak wcześniej zostało omówione przy okazji strategii rozpraszania (np. przy użyciu strategii Parameter →ServerStrategy). Usługa Platform AI automatycznie skonfiguruje zmienną środowiskową `TF_CONFIG` w każdej maszynie wirtualnej. Po zakończeniu tego etapu możesz wdrożyć swój model i uruchomić go w klastrze TF w sposób podobny do następującego²³:

```
$ gcloud ai-platform jobs submit training moja_grupa_zadan_20190531_164700 \
  --region asia-southeast1 \
  --scale-tier PREMIUM 1 \
  --runtime-version 2.0 \
  --python-version 3.5 \
  --package-path /moj_projekt/src/trainer \
  --module-name trainer.task \
  --staging-bucket gs://moj-zasobnik-publikujacy \
  --job-dir gs://moj-zasobnik-modelu-mnist/trained_model \
  --
  --moj-dodatkowy-argument1 foo --moj-dodatkowy-argument2 bar
```

Przeanalizujmy te opcje. Polecenie rozpoczynamy od grupy zadań, nazwanej tu `moja_grupa_zadan_20190531_164700`, którą umieszczamy w regionie `asia-southeast1` i wykorzystujemy poziom

²³ Szczegółowe instrukcje znajdziesz pod adresem <https://cloud.google.com/ai-platform/training/docs/packaging-trainer> — przyp. red.

skali (ang. *scale tier*) PREMIUM_1, uzyskując dzięki niemu dostęp do 20 roboczych grup zadań (w tym serwera głównego) i 11 serwerów parametrów (pozostałe poziomy skali zostały opisane na stronie <https://cloud.google.com/ml-engine/docs/machine-types>). Wszystkie te maszyny wirtualne będą bazować na środowisku AI Platform 2.0 (czyli konfiguracji maszyny wirtualnej obejmującej moduł TensorFlow 2.0 i wiele innych pakietów)²⁴ i Python 3.5. Kod uczący mieści się w katalogu /moj_projekt/src/trainer, a polecenie gcloud automatycznie przetworzy go w pakiet pip i umieści w usłudze GCS pod adresem gs://moj-zasobnik-publikujacy. Następnie usługa AI Platform uruchomi kilka maszyn wirtualnych, wdroży do nich pakiet i uruchomi moduł trainer.task. Na koniec argument --job-dir i argumenty dodatkowe (tzn. wszystkie argumenty umieszczone po separatorze --) zostaną przekazane do programu uczącego: zadanie serwera głównego będzie zazwyczaj wykorzystywało argument --job-dir do określenia miejsca, w którym ma być zapisywany model końcowy w usłudze GCS (w tym przypadku jest to gs://moj-zasobnik-modelu-mnist/trained_model). I to wszystko! Z poziomu konsoli GCP możesz otworzyć menu nawigacyjne, przejść do sekcji *Sztuczna inteligencja*, a następnie kliknąć opcję *AI Platform i Zadania*. Powinna wyświetlić się *Twoja grupa zadań*, a po jej kliknięciu ujrzesz wykresy zużycia procesora, kart graficznych i pamięci operacyjnej dla każdego zadania. Możesz kliknąć opcję *Show Logs*, aby uzyskać dostęp do szczegółowych informacji o zdarzeniach generowanych przez usługę Stackdriver.



Jeżeli chcesz umieścić dane uczące w usłudze GCS, możesz utworzyć obiekt `tf.data.TextLineDataset` lub `tf.data.TFRecordDataset`, aby uzyskiwać do nich dostęp. Jako nazwy plików wystarczy użyć ścieżek usługi GCS (np. `gs://moj-zasobnik-danych/moje_dane_001.csv`). Dostęp do plików za pomocą tych obiektów jest zależny od pa-kietu `tf.io.gfile`: obsługuje on zarówno pliki lokalne, jak i pliki GCS (upewnij się jednak, że wykorzystywane konto usługi ma dostęp do usługi GCS).

Jeśli chcesz przetestować kilka różnych wartości hiperparametrów, wystarczy uruchomić kilka grup zadań i wyznaczyć te wartości za pomocą dodatkowych argumentów. Jeśli jednak chcesz wydajnie sprawdzać wiele hiperparametrów, to warto skorzystać z usługi strojenia hiperparametrów dostępnych w ramach Google AI Platform.

Penetracyjne strojenie hiperparametrów w usłudze AI Platform

W ramach AI Platform znajdziemy potężną bayesowską usługę optymalizacji strojenia hiperparametrów o nazwie Google Vizier (<https://homl.info/vizier>)²⁵. Aby z niej skorzystać, musisz przesyłać plik konfiguracyjny YAML podczas tworzenia grupy zadań (--config tuning.yaml). Może on wyglądać na przykład tak:

```
trainingInput:  
  hyperparameters:  
    goal: MAXIMIZE  
    hyperparameterMetricTag: accuracy
```

²⁴ W czasie gdy pisałem tę książkę, środowisko uruchomieniowe w wersji 2.0 było jeszcze niedostępne, ale powinno zostać opublikowane, gdy będziesz ją czytać. Listę dostępnych środowisk znajdziesz pod adresem <https://cloud.google.com/ml-engine/docs/runtime-version-list>.

²⁵ Daniel Golovin i in., *Google Vizier: A Service for Black-Box Optimization*, „Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining” (2017), s. 1487 – 1495.

```
maxTrials: 10
maxParallelTrials: 2
params:
  - parameterName: n_layers
    type: INTEGER
    minValue: 10
    maxValue: 100
    scaleType: UNIT_LINEAR_SCALE
  - parameterName: momentum
    type: DOUBLE
    minValue: 0.1
    maxValue: 1.0
    scaleType: UNIT_LOG_SCALE
```

Informujemy w ten sposób usługi AI Platform, że chcemy maksymalizować wskaźnik o nazwie "accuracy", grupa zadań będzie działać maksymalnie przez 10 prób (w każdej próbie model jest trenowany od podstaw za pomocą danych uczących) i maksymalnie będą przetwarzane dwie próby jednocześnie. Chcemy stroić dwa hiperparametry: n_layers (liczba całkowita w zakresie od 10 do 100) i momentum (wartość zmiennoprzecinkowa w zakresie od 0,1 do 1,0). Argument scaleType wyznacza rozkład wartości hiperparametru: UNIT_LINEAR_SCALE oznacza rozkład płaski (czyli jednostajny a priori), z kolei zgodnie z UNIT_LOG_SCALE jesteśmy przekonani, że wartość optymalna hiperparametru znajduje się bliżej wartości maksymalnej (odwrotność tego rozkładu stanowi skala UNIT_REVERSE_LOG_SCALE, zgodnie z którą uważaemy, że wartość optymalna mieści się w pobliżu wartości minimalnej).

Argumenty n_layers i momentum zostaną przekazane w postaci argumentów wiersza polecenia do kodu uczącego, gdzie oczywiście zostaną wykorzystane. Pojawia się pytanie, w jaki sposób kod ten przekaże wskaźniki do usługi AI Platform, aby mogła ona zadecydować, jakie wartości hiperparametrów mają zostać użyte w następnej próbie. Usługa ta po prostu monitoruje katalog wyjściowy (wyznaczony za pomocą argumentu --job-dir) i wychwytuje każdy plik zdarzeń (zob. rozdział 10.) zawierający wskaźnik o nazwie "accuracy" (ogółem każdą nazwę wskaźnika zdefiniowaną w hyperparameter MetricTag), a następnie odczytuje te wartości. Zatem wystarczy, że kod uczący wykorzysta wywołanie zwrotne TensorBoard() (którego chcemy tak czy siak używać do monitorowania) i to wszystko!

Po zrealizowaniu grupy zadań wszystkie wartości hiperparametrów użyte w każdej próbie oraz wyliczona dokładność będą dostępne w wynikach grupy zadań (dostępne po wybraniu *AI Platform/Zadania*).



Grupy zadań w usłudze AI Platform mogą być używane do skutecznego przetwarzania dużych ilości danych przez model: każda robocza grupa zadań może odczytywać część danych z usługi GCS, uzyskiwać prognozy i zapisywać je w usłudze GCS.

Dysponujesz już wszystkimi narzędziami i wiedzą wymaganymi do tworzenia nowoczesnych struktur sieci neuronowych i ich wielkoskalowego uczenia za pomocą różnych strategii rozpraszania przy użyciu własnej infrastruktury lub usług w chmurze, a do tego jesteś nawet w stanie przeprowadzać zaawansowaną optymalizację bayesowską w celu strojenia hiperparametrów!

Ćwiczenia

1. Jakie elementy zawiera obiekt SavedModel? Jak można sprawdzić jego zawartość?
2. Kiedy należy korzystać z serwera TF Serving? Jakie są jego główne własności? Za pomocą jakich narzędzi możesz go wdrożyć?
3. W jaki sposób wdrażamy model do wielu instancji TF Serving?
4. Kiedy lepiej wykorzystywać interfejs gRPC zamiast interfejsu REST do przepisywania modelu eksplotowanego za pomocą serwera TF Serving?
5. Na jakie sposoby biblioteka TFLite zmniejsza rozmiar modelu po to, aby można było go uruchomić na urządzeniu mobilnym/wbudowanym?
6. Czym jest uczenie w kontekście kwantyzacji i dlaczego jest potrzebne?
7. Czym są zrównoleglanie modelu i zrównoleglanie danych? Dlaczego bardziej zalecane jest stosowanie drugiego rozwiązania?
8. Jakie strategie rozpraszania możesz zastosować podczas uczenia modelu na wielu serwerach? W jaki sposób wybieramy odpowiednią strategię?
9. Wyucz jakiś model (dowolny) i wdróż go do serwera TF Server lub usługi Google Cloud AI Platform. Napisz kod kliencki wysyłający zapytania do tego modelu poprzez protokół REST albo gRPC. Zaktualizuj ten model i wdróż jego nową wersję. Twój kod kliencki będzie teraz wysyłał kwerendy do nowej wersji modelu. Przywróć poprzednią wersję modelu.
10. Wytrenuj dowolny model za pomocą wielu jednostek GPU na tym samym komputerze przy użyciu klasy MirroredStrategy (jeżeli nie masz dostępu do kart graficznych, skorzystaj z serwisu Colaboratory: wybierz program uruchomieniowy GPU i utwórz dwa wirtualne procesory GPU). Wyucz ponownie model za pomocą strategii CentralStorageStrategy i porównaj czas przeznaczony na trening.
11. Wytrenuj mały model w usłudze Google Cloud AI Platform i skorzystaj z penetracyjnego strojenia hiperparametrów.

Dziękuję!

Przed zakończeniem ostatniego rozdziału chciałbym podziękować Ci za przeczytanie książki aż do ostatniego akapitu. Mam szczerą nadzieję, że jej lektura sprawiła Tobie tyle samo przyjemności, ile mnie jej pisanie, oraz że okaże się przydatna w Twoich przyszłych projektach, bez względu na ich rozmiar.

Jeżeli znajdziesz jakiekolwiek błędy, będę wdzięczny, gdy mnie o nich poinformujesz. Jestem bardzo ciekaw, co myślisz o tej pozycji, dlatego śmiało odezwij się do mnie poprzez stronę projektu *ageron/handson-ml2* w serwisie GitHub albo na Twitterze (@aureliengeron).

Cóż Ci mogę teraz doradzić? Trening czyni mistrza, spróbuj więc rozwiązać wszystkie dostępne tu ćwiczenia, jeśli jeszcze tego nie zrobiłaś/zrobiliś, pobaw się notatnikami Jupyter, dołącz do społeczności Kaggle.com lub jakiejś innej społeczności uczenia maszynowego, oglądaj powiązane z tematem prezen-

tacje, czytaj pojawiające się codziennie nowe publikacje, uczęszczaj na specjalistyczne konferencje i kontaktuj się z ekspertami. Bardzo pomaga również wyznaczenie konkretnego projektu, nieważne, czy do pracy czy dla zabawy (najlepiej do obydwu jednocześnie), dlatego jeżeli marzyłaś/marzyłeś o stworzeniu jakiegoś modelu, teraz jest na to idealny czas! Pracuj stopniowo, nie wybieraj się od razu na Księzyc, lecz skoncentruj się na projekcie i buduj go cegiełka po cegielce. Będziesz potrzebować mnóstwa cierpliwości i samozaparcia, ale stworzenie poruszającego się robota, działającego czatbota czy czegokolwiek innego okaże się bardzo satysfakcjonujące.

Mam nadzieję, że niniejsza książka zainspiruje Cię do przygotowania cudownych zastosowań uczenia maszynowego, które przysłużą się nam wszystkim. Masz już jakiś pomysł?

— Aurélien Géron, 17 czerwca 2019 roku

Rozwiązania ćwiczeń



Rozwiązania ćwiczeń wymagających napisania kodu znajdziesz w notatnikach Jupyter dostępnych pod adresem <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 1. Krajobraz uczenia maszynowego

1. Uczenie maszynowe polega na tworzeniu systemów zdolnych do uczenia się z danych. Przez „uczenie się” rozumiemy uzyskiwanie coraz lepszych wyników w jakimś zadaniu przy uwzględnieniu określonej miary wydajności.
2. Uczenie maszynowe nadaje się doskonale do złożonych problemów, dla których nie jesteśmy w stanie stworzyć algorytmicznego rozwiązania, do zastępowania długich list ręcznie dostrajanych reguł, do tworzenia systemów dostosowujących się do zmiennych środowisk, a także do uczenia ludzi (np. wydobywania danych).
3. Oznakowanym zestawem danych uczących nazywamy zbiór danych zawierający oczekiwane rozwiązanie (etykietę) dla każdej próbki.
4. Najpopularniejsze rodzaje zadań nadzorowanych to regresja i klasyfikacja.
5. Do popularnych zadań nienadzorowanych należą analiza skupień, wizualizacja, redukcja wymiarowości oraz uczenie przy użyciu reguł asocjacyjnych.
6. Uczenie przez wzmacnianie jest bodaj najlepszym wyborem, jeśli chcemy nauczyć robota poruszania się po różnych nieznanych terenach, ponieważ strategia ta jest przystosowana do rozwiązywania tego typu problemów. Możliwe jest wyrażenie takiego problemu w postaci zadania uczenia nienadzorowanego lub nadzorowanego, lecz byłoby ono mniej naturalne.
7. Jeśli nie wiesz, w jaki sposób definiować grupy, możesz skorzystać z algorytmu analizy skupień (uczenie nienadzorowane) do rozdzielenia klientów na grupy według podobnych zainteresowań. Jeśli jednak wiesz, jakie grupy chcesz mieć, to możesz dostarczyć wiele przykładów z każdej takiej grupy do algorytmu klasyfikującego (uczenie nadzorowane), który rozmieści klientów w odpowiednich skupieniach.
8. Wykrywanie spamu stanowi klasyczny problem uczenia nadzorowanego: dostarczamy algorytmowi wiele wiadomości e-mail wraz z etykietami (spam lub zwykła wiadomość).

9. System uczenia przyrostowego, jak sama nazwa wskazuje, może uczyć się przyrostowo, w przeciwieństwie do systemu uczenia wsadowego. Dzięki temu jest on w stanie szybko dostosowywać się zarówno do zmieniających się danych, jak i do systemów autonomicznych, a także potrafi uczyć się na bardzo dużych ilościach danych.
10. Algorytmy pozakorowe mogą obsługiwać olbrzymie ilości danych, które nie mieszczą się w głównej pamięci komputera. Algorytm uczenia pozakorowego rozdziela dane na minigrupy i do nauki z tych minigrup wykorzystuje techniki uczenia przyrostowego.
11. System uczenia z przykładów uczy się dostarczanych przykładów „na pamięć”, następnie po dostarczeniu nowej próbki wykorzystuje miarę podobieństwa do wyszukania przypominających ją próbek uczących i za ich pomocą oblicza prognozę.
12. Model zawiera przynajmniej jeden parametr określający prognozowany czynnik przy uwzględnieniu nowej próbki (np. nachylenie modelu liniowego). Algorytm uczący stara się znaleźć optymalne wartości tych parametrów, dzięki którym model będzie dobrze uogólniał wyniki dla nowych próbek. Hiperparametrem nazywamy parametr samego algorytmu uczącego, a nie modelu (np. stopień stosowanej regularizacji).
13. Algorytmy uczenia z modelu wyszukują optymalne wartości parametrów modelu, za pomocą których model ten będzie dobrze uogólniał wyniki do nowych przykładów. Zazwyczaj uczymy takie systemy poprzez minimalizowanie funkcji kosztu mierzącej błędy prognozowania systemu dla zestawu uczącego, z uwzględnieniem kary za złożoność modelu, jeśli jest on regularizowany. W celu uzyskiwania prognoz dostarczamy cechy nowego przykładu do funkcji predykcyjnej modelu za pomocą wartości parametrów określonych przez algorytm uczący.
14. Do głównych wyzwań w dziedzinie uczenia maszynowego należą: brak danych, niska jakość danych, niereprezentatywne dane, nieprzydatne cechy, nadmiernie uproszczone modele ulegające niedotrenowaniu oraz nazbyt złożone modele mające tendencję do przetrenowania.
15. Jeżeli model spisuje się świetnie w przypadku danych uczących, ale niezbyt dobrze generalizuje wyniki z nowych przykładów, to najprawdopodobniej uległ przetrenowaniu (lub mieliśmy wyjątkowe szczęście w doborze zestawu uczącego). Możemy zmniejszyć stopień przetrenowania, dodając więcej danych uczących, upraszczając model (poprzez wybór prostszego algorytmu, zmniejszenie liczby wykorzystywanych parametrów bądź cech lub przeprowadzenie regularizacji modelu) albo redukując zasumienie danych uczących.
16. Zestaw testowy służy do oszacowania błędu uogólniania, który model popełnia w przypadku nowych przykładów, przed jego wprowadzeniem do środowiska produkcyjnego.
17. Za pomocą zestawu walidacyjnego możemy porównywać modele. Umożliwia on wybór najlepszego modelu i dostrojenie hiperparametrów.
18. Zestaw ucząco-rozwojowy używany jest w przypadku pojawienia się ryzyka niedopasowania pomiędzy danymi uczącymi a testowymi lub walidacyjnymi (powinny one być zawsze maksymalnie podobne do danych dostarczanych już w środowisku produkcyjnym). Zestaw ten stanowi odłączony (niewykorzystywany w uczeniu) podzióbior zestawu danych uczących. Model jest uczony za pomocą pozostałej części zestawu danych i oceniany zarówno za pomocą zbioru ucząco-rozwojowego, jak i zbioru walidacyjnego. Jeżeli model radzi sobie dobrze z zestawem uczącym, ale nie ucząco-rozwojowym, to prawdopodobnie jest przetrenowany przez zbiór

uczący. Jeżeli uzyskuje dobre wyniki zarówno dla zestawu danych uczących, jak i ucząco-rozwojowych, ale nie dla zbioru walidacyjnego, to prawdopodobnie występuje znaczne niedopasowanie danych pomiędzy danymi uczącymi a danymi walidacyjnymi i testowymi, więc należy usprawnić dane uczące tak, aby przypominały bardziej dane testowe i walidacyjne.

19. Jeżeli dostroisz hiperparametry za pomocą zestawu testowego, ryzykujesz przetrenowanie modelu wobec niego, a wartość błędu uogólniania będzie optymistycznie mała (możesz wdrożyć model, który będzie sprawował się gorzej, niż można było się spodziewać).

Rozdział 2. Nasz pierwszy projekt uczenia maszynowego

Rozwiązania zadań znajdziesz w notatniku Jupyter dostępnym pod adresem
<ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 3. Klasyfikacja

Rozwiązania zadań znajdziesz w notatniku Jupyter dostępnym pod adresem
<ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 4. Uczenie modeli

1. Jeżeli używamy zestawu uczącego zawierającego miliony cech, możemy skorzystać ze stochastycznego spadku wzdłuż gradientu lub schodzenia po gradiencie z minigrupami, a może nawet wsadowego gradientu prostego, jeśli dane uczące mieszczą się w pamięci. Nie możemy jednak zastosować równania normalnego, ponieważ wraz ze wzrostem liczby cech bardzo szybko rośnie złożoność obliczeniowa (postęp jest większy od kwadratowego).
2. Jeżeli cechy w zbiorze danych uczących występują w bardzo różnych skalach, funkcja kosztu będzie miała wydłużony kształt, zatem algorytmy gradientu prostego będą potrzebować mnóstwo czasu na osiągnięcie zbieżności. W celu rozwiązania tego problemu należy przeskalować dane przed wyuczeniem modelu. Zwrót uwagę, że równanie normalne i rozkład SVD będą działać prawidłowo bez konieczności skalowania cech. Co więcej, w przypadku nieprzeskalowanych cech modele regularyzowane mogą uzyskiwać nie najlepszą zbieżność: regularyzacja w istocie nakłada karę za duże wagę, dlatego cechy o mniejszych wartościach będą ignorowane w porównaniu do cech o większych wartościach.
3. Algorytm gradientu prostego nie może utknąć w minimum lokalnym podczas uczenia modelu regresji logistycznej, ponieważ funkcja kosztu jest wypukła¹.
4. Jeżeli problem optymalizacji jest wypukłej natury (jak w przypadku regresji liniowej lub logistycznej), a współczynnik uczenia nie jest zbyt duży, to algorytmy gradientu prostego osiągną optimum globalne i będą w stanie utworzyć w miarę podobne modele. Jeżeli jednak będziemy stopniowo zmniejszać współczynnik uczenia, takie algorytmy jak stochastyczny spadek wzdłuż gradientu lub

¹ Jeżeli narysujesz linię prostą pomiędzy dowolnymi dwoma punktami na krzywej, linia ta nigdy nie przejdzie przez tę krzywą.

schodzenie po gradiencie z minigrupami nigdy nie osiągną prawdziwej zbieżności — będą ciągle przeskakiwać globalne optimum. Oznacza to, że nawet jeśli pozwolimy im działać przez bardzo długi czas, będą one produkować nieco odmienne modele.

5. Jeżeli błąd walidacji stale rośnie wraz z każdą epoką, to prawdopodobnie współczynnik uczenia jest za duży i algorytm staje się rozbieżny z rozwiązaniem. Jeśli wzrasta również błąd uczenia, to na pewno mamy do czynienia ze wspomnianą przyczyną i należy zmniejszyć wartość współczynnika uczenia. Jeśli jednak błąd uczenia nie rośnie, to model ulega przetrenowaniu i należy przerwać jego uczenie.
6. Zarówno stochastyczny spadek wzduż gradientu, jak i schodzenie po gradiencie z minigrupami z powodu losowości nie gwarantują postępów wraz z każdym przebiegiem uczenia. Zatem jeśli natychmiast przerwiemy uczenie w sytuacji wzrostu błędu walidacji, możemy zbyt szybko zatrzymać ten proces (jeszcze przed osiągnięciem optimum). Lepszym rozwiązaniem jest zapisywanie modelu w regularnych odstępach czasu, a gdy wydajność przestanie wzrastać już przez dłuższy czas (model nie będzie w stanie pobić wcześniejszego rekordu), wystarczy wczytać najlepszy uzyskany model.
7. Stochastyczny spadek wzduż gradientu ma najszybsze przebiegi uczenia, ponieważ każdorazowo analizuje pojedynczy przykład uczący, zatem zazwyczaj jako pierwszy dociera w okolice optimum globalnego (podobnie jest w przypadku schodzenia po gradiencie z bardzo małymi minigrupami). Jednak prawdziwą zbieżność uzyska jedynie wsadowy gradient prosty, jeśli damy mu odpowiednią ilość czasu. Jak już wiemy, stochastyczny spadek wzduż gradientu i schodzenie po gradiencie z minigrupami będą pomijać optimum, dopóki nie będziemy stopniowo zmniejszać współczynnika uczenia.
8. Jeżeli błąd walidacji jest znacznie większy od błędu uczenia, to prawdopodobnie mamy do czynienia z przetrenowaniem modelu. Możemy temu zaradzić, zmniejszając stopień wielomianu: model mający mniejszą liczbę stopni swobody jest bardziej odporny na przetrenowanie. Innym rozwiązaniem jest regularyzacja modelu — na przykład przez dodanie członu ℓ_2 (regularizacja grzbietowa) lub ℓ_1 (regularizacja metodą LASSO) do funkcji kosztu. Metoda ta również redukuje liczbę stopni swobody modelu. Na koniec możemy jeszcze spróbować zwiększyć rozmiar zestawu uczącego.
9. Jeżeli wartości błędów uczenia i walidacji są niemal równe i duże, to najprawdopodobniej model jest niedotrenowany, co oznacza, że ma duże obciążenie. Należy w takim przypadku zmniejszyć wartość hiperparametru regularyzacji α .

10. Pomyślmy:

- Model zawierający jakąś formę regularyzacji zazwyczaj sprawdza się lepiej od modelu nieregularyzowanego, dlatego należy wybierać regresję grzbietową zamiast zwykłej regresji liniowej.
- Regresja metodą LASSO wykorzystuje regularyzację ℓ_1 , która dąży do zerowania wag. Uzyskujemy w ten sposób modele rzadkie, gdzie wszystkie wagi oprócz najistotniejszych mają wartość 0. Jest to mechanizm automatycznego doboru cech, który przydaje się, gdy podejrzewamy, że tylko kilka cech ma znaczenie. Jeśli nie mamy takiej pewności, powinniśmy korzystać z regresji grzbietowej.

- Metoda elastycznej siatki jest preferowana ponad regresję typu LASSO, ponieważ ta druga może w pewnych sytuacjach zachowywać się nieprzewidywalnie (gdy kilka cech jest ze sobą ściśle skorelowanych lub gdy w zestawie danych występuje więcej cech niż przykładów uczących). Jednak ta pierwsza metoda dodaje kolejny hiperparametr do strojenia. Jeśli chcesz używać metody LASSO bez ryzyka nieprzewidywalnego zachowania, możesz wykorzystać technikę elastycznej siatki, w której parametr `l1_ratio` miałby wartość bliską 1.
11. Jeśli chcesz klasyfikować zdjęcia jako wykonane na zewnątrz/w pomieszczeniu, w dzień/w nocy, należy wyuczyć dwa klasyfikatory regresji logistycznej, ponieważ cechy te nie wykluczają się wzajemnie (tzn. możliwe są ich cztery kombinacje).
12. Odpowiedź znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przykłady/uczem2.zip>.

Rozdział 5. Maszyny wektorów nośnych

1. Podstawowa koncepcja opisująca maszyny wektorów nośnych polega na określaniu najszerzej możliwej „ulicy” pomiędzy klasami. Inaczej mówiąc, celem jest ustalenie jak największego marginesu pomiędzy granicą decyzyjną oddzielającą dwie klasy a przykładami uczącymi. W przypadku klasyfikacji miękkiego marginesu maszyna SVM poszukuje kompromisu między perfekcyjnym rozdzieleniem dwóch klas a jak najszerzą „ulicą” (tzn. że kilka próbek może przekroczyć granicę marginesu). Kolejną zasadniczą koncepcją maszyn SVM jest używanie jąder podczas uczenia modelu za pomocą zbiorów nielinowych danych.
2. Po wytrenowaniu maszyny SVM **wektorem nośnym** staje się każdy przykład znajdujący się na „ulicy” (zob. poprzednią odpowiedź), w tym również na granicy marginesu. Granica decyzyjna jest w zupełności określana przez wektory nośne. Każda próbka *niebędąca* wektorem nośnym (tzn. znajdująca się poza „ulicą”) nie ma wpływu na margines — możemy je usuwać, dodawać nowe przykłady lub je przenosić, a dopóki nie będą przekraczały „ulicy”, nie będą wpływać na granicę decyzyjną. Wyliczanie prognoz dotyczy jedynie wektorów nośnych, a nie całego zestawu danych uczących.
3. Maszyny SVM próbują ustalić najszerszą możliwą „ulicę” pomiędzy klasami (zob. odpowiedź 1.), zatem jeśli zestaw danych uczących nie jest wyskalowany, model może ignorować mniejsze cechy (rysunek 5.2).
4. Klasyfikator SVM może obliczać odległość między przykładem testowym a granicą decyzyjną, co może nam posłużyć jako wynik pewności. Jednak wynik ten nie może zostać bezpośrednio przekształcony w oszacowanie prawdopodobieństwa przynależności do klasy. Jeśli wyznaczymy parametr `probability=True` podczas tworzenia maszyny SVM w module Scikit-Learn, to po zakończeniu uczenia model wykalibruje te prawdopodobieństwa za pomocą regresji logistycznej użytej dla wyników maszyny SVM (algorytm regresji jest wytrenowany za pomocą dodatkowego pięciokrotnego sprawdzianu krzyżowego dla danych uczących). W ten sposób zostają dodane metody `predict_proba()` i `predict_log_proba()` do maszyny SVM.
5. Pytanie to dotyczy wyłącznie liniowych maszyn SVM, ponieważ maszyny kernelizowane mogą korzystać tylko z formy dualnej. Złożoność obliczeniowa problemu pierwotnego w maszynach SVM jest proporcjonalna do liczby przykładów uczących m , natomiast w problemie dualnym

wzrasta ona do przedziału od m^2 do m^3 . Zatem w przypadku korzystania z milionów przykładów zdecydowanie należy używać formy pierwotnej, ponieważ forma dualna będzie znacznie wolniejsza.

6. Jeśli klasyfikator SVM wyuczony za pomocą jądra RBF wykazuje oznaki niedotrenowania, może to oznaczać nadmiar regularyzacji. Aby ją zredukować, należy zwiększyć wartość hiperparametru gamma lub C (albo obydwu).
7. Weźmy pod uwagę parametry programowania kwadratowego używane w zagadnieniach twardego marginesu: \mathbf{H} , f , \mathbf{A} i \mathbf{b} (zob. punkt „Programowanie kwadratowe”). W przypadku problemu miękkiego marginesu istnieje m dodatkowych parametrów ($n_p = n+1+m$) i tyle samo ograniczeń ($n_c = 2m$). Możemy zdefiniować je następująco:

- Parametr \mathbf{H} jest równy parametrowi \mathbf{H} z m kolumn zawierających zera dołączone po prawej stronie i m wierszy wypełnionych zerami dodanymi na dole:
$$\mathbf{H} = \begin{pmatrix} \mathbf{H}' & 0 & \cdots \\ 0 & 0 & \\ \vdots & & \ddots \end{pmatrix}.$$
- Parametr f jest równy parametrowi f zawierającemu m dodatkowych elementów, z których wszystkie są równe wartości hiperparametru C .
- Parametr \mathbf{b} jest równy parametrowi \mathbf{b} zawierającemu m dodatkowych elementów, z których wszystkie są równe 0.
- Parametr \mathbf{A} jest równy parametrowi \mathbf{A} zawierającemu dodatkową macierz jednostkową \mathbf{I}_m o rozmiarze $m \times m$ dołączoną po prawej stronie oraz macierz $-\mathbf{I}_m^*$ dodaną na dole, pozostałe elementy zaś mają wartość 0:
$$\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}.$$

Rozwiązania ćwiczeń 8., 9. i 10. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 6. Drzewa decyzyjne

1. Wysokość zrównoważonego drzewa zawierającego m liści jest równa $\log_2(m)$,² gdzie wynik zaokrąglamy w góre. Każde binarne drzewo decyzyjne (podejmujące wyłącznie decyzje binarne — czyli wszystkie rodzaje drzew w module Scikit-Learn) na końcu procesu uczenia będzie mniej lub bardziej zrównoważone, a na każdą próbkę uczącą będzie przypadać jeden liść, jeżeli nie wprowadzimy żadnego ograniczenia. Zatem jeśli zestaw danych uczących składa się z miliona przykładów, drzewo decyzyjne osiągnie wysokość $\log(10^6) \approx 20$ (w rzeczywistości nieco większą, gdyż zazwyczaj nie będzie ono perfekcyjnie zrównoważone).
2. Wskaźnik Giniego dla węzła podzielnego jest zazwyczaj mniejszy niż dla węzła nadzielnego. Wynika to z funkcji kosztu algorytmu uczącego CART, który rozdziela każdy węzeł w sposób minimalizujący sumę ważoną wskaźników Giniego w węzłach potomnych. Jednak istnieje możliwość, że potomek będzie miał większy wskaźnik Giniego od rodzica, pod warunkiem że

² Wyrażenie \log_2 oznacza logarytm binarny, czyli $\log_2(m) = \log(m)/\log(2)$.

zjawisko to będzie równoważone przez zmniejszoną wartość tego wskaźnika w drugim węźle potomnym. Na przykład przeanalizujmy węzeł zawierający cztery przykłady klasy A i jeden przykład klasy B. Jego wskaźnik Giniego wynosi $1 - \left(\frac{1}{5}\right)^2 - \left(\frac{4}{5}\right)^2 = 0,32$. Założymy teraz, że zbiór danych jest jednowymiarowy, a przykłady są umieszczone w nim następująco: A, B, A, A, A. Możemy sprawić, że algorytm ten rozdzieli te przykłady na dwa węzły potomne, z których pierwszy będzie zawierał dane A i B, a w drugim znajdą się przykłady A, A, A. Wskaźnik Giniego dla pierwszego węzła wyniesie $1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0,5$, czyli będzie wyższy niż w przypadku rodzica. Zostaje to zrównoważone przez fakt, że drugi węzeł potomny jest czysty, zatem całkowity ważony wskaźnik Giniego wyniesie w nim $\frac{2}{5} \times 0,5 + \frac{3}{5} \times 0 = 0,2$, czyli ma niższą wartość od węzła nadzawanego.

- 3.** Jeżeli drzewo decyzyjne ulega przetrenowaniu w przypadku danych uczących, dobrym pomysłem jest zmniejszenie wartości parametru `max_depth`, gdyż w ten sposób poprzez regularyzację ograniczymy model.
- 4.** W metodzie drzew decyzyjnych nie jest ważne, czy dane uczące są skalowane lub wyśrodkowane — jest to jedna z zalet tej techniki. Zatem jeśli drzewo decyzyjne wykazuje oznaki niedotrenowania, skalowanie cech wejściowych okaże się stratą czasu.
- 5.** Złożoność obliczeniowa procesu uczenia drzewa decyzyjnego wynosi $O(x \times m \log(m))$. Zatem jeśli powiększymy dziesięciokrotnie rozmiar zbioru danych uczących, czas uczenia wydłuży się o $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. Jeśli $m = 10^6$, to $K \approx 11,7$, czyli możemy oczekwać, że proces uczenia zajmie w przybliżeniu 11,7 godzin.
- 6.** Wstępne sortowanie danych przyśpiesza proces uczenia jedynie wtedy, gdy rozmiar zbioru danych nie przekracza kilku tysięcy przykładów. W przypadku 100 000 próbek wyznaczenie parametru `presort=True` znacznie spowolni uczenie modelu.

Rozwiązańććwiczeń 7. i 8. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 7. Uczenie zespołowe i losowe lasy

- 1.** Po wytrenowaniu pięciu różnych modeli osiągających precyzyję rzędu 95% możemy spróbować połączyć je w zespół głosujący, dzięki któremu możemy nieraz osiągać jeszcze lepsze wyniki. Rozwiązańto sprawdza się tym lepiej, im bardziej różnorodne są poszczególne modele (np. klasyfikator SVM, drzewo decyzyjne, regresja logistyczna itd.). Byłyby również znakomicie, gdyby każdy z tych klasyfikatorów był trenowany za pomocą różnych próbek uczących (na tym bazują metody agregacji i wklejania), ale nie jest to warunek konieczny, dopóki korzystamy z różnych typów modeli.
- 2.** Klasyfikator twardego głosowania zlicza jedynie głosy każdego członka zespołu i wybiera klasę, która uzyskała najwięcej głosów. Z kolei klasyfikator miękkiego głosowania wylicza średnie oszacowanie prawdopodobieństwa przynależności do każdej klasy i wybiera klasę o największym

prawdopodobieństwie. W ten sposób pewniejsze głosy mają większą wagę i często metoda ta daje lepsze rezultaty, jednak działa ona tylko wtedy, gdy wszystkie klasyfikatory są w stanie szacować prawdopodobieństwo przynależności do klasy (tzn. dla klasyfikatorów SVM w module Scikit-Learn musimy wyznaczyć parametr `probability=True`).

3. Przyspieszenie uczenia zespołu agregacji poprzez rozmieszczenie go pomiędzy wiele serwerów jest całkiem możliwe, ponieważ poszczególne predyktry są niezależne od siebie. To samo dotyczy zespołów wklejania i losowych lasów. Jednak każdy predyktor w zespole wzmacniania jest stworzony na podstawie poprzednika, zatem proces uczenia przebiega w tym przypadku sekwencyjnie i umieszczenie poszczególnych składowych na różnych serwerach nic nam nie da. Jeśli chodzi o zespoły kontaminacji, wszystkie predyktry w danej warstwie są wzajemnie niezależne, więc mogą być uczone równolegle na osobnych serwerach. Jednakże predyktry z wyższej warstwy mogą być wyuczone dopiero po wytrenowaniu predyktorów z warstwy poprzedniej.
4. Dzięki ocenie pozatreningowej każdy predyktor w zespole agregacji jest oceniany przy użyciu przykładów, które nie były użyte w procesie uczenia (zostały one odseparowane). Możliwa jest dzięki temu w miarę nieobciążona ewaluacja zespołu bez konieczności tworzenia dodatkowego zespołu walidacyjnego. Zatem mamy w ten sposób więcej przykładów, których możemy użyć do uczenia, a sam zespół osiąga nieco lepszą wydajność.
5. Podczas generowania drzewa w modelu losowego lasu przy podziale węzłów jest brany pod uwagę jedynie losowy podzbior cech. Jest to prawdziwe stwierdzenie również w przypadku algorytmu Extra-Trees, ale tutaj posuwamy się jeszcze dalej: w przeciwieństwie do klasycznych drzew decyzyjnych, w których poszukiwane są najlepsze możliwe progi, tutaj każdej cęsie zostaje wyznaczony losowy próg. Ta dodatkowa losowość pełni funkcję regularyzacji: jeśli losowy las ulega przetrenowaniu wobec danych uczących, algorytm Extra-Trees może lepiej spełniać dane zadanie. Ponadto metoda Extra-Trees nie poszukuje najlepszych możliwych progów, dlatego jej proces uczenia przebiega znacznie szybciej niż w przypadku klasycznego losowego lasu. Jednak algorytm ten nie jest ani szybszy, ani wolniejszy w obliczaniu prognoz.
6. Jeżeli zespół AdaBoost wykazuje oznaki niedotrenowania, możesz zwiększyć liczbę estymatorów lub zmniejszyć wartości hiperparametrów regularyzacji w bazowym estymatorze. Możesz także nieznacznie zwiększyć wartość współczynnika uczenia.
7. Jeżeli zespół wzmacniania gradientowego wykazuje oznaki przetrenowania, należy zmniejszyć wartość współczynnika uczenia. Możesz również wprowadzić metodę wczesnego zatrzymywania, aby określić prawidłową liczbę predyktorów (prawdopodobnie zespół zawiera ich zbyt wiele).

Rozwiązańcia ćwiczeń 8. i 9. znajdziesz w notatniku Jupyter dostępnym na stronie
<ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 8. Redukcja wymiarowości

1. Głównymi przyczynami redukowania wymiarowości są:
 - przyspieszenie następnego algorytmu uczącego (w pewnych sytuacjach skutkuje ono nawet usunięciem szumu i nadmiarowych cech, co zwiększa wydajność algorytmu);

- wizualizacja danych i analiza najistotniejszych cech;
- oszczędność miejsca (kompresja).

Głównymi wadami są:

- utrata części informacji, co może zmniejszyć wydajność kolejnych algorytmów uczących;
 - potrzeba dysponowania znaczną mocą obliczeniową;
 - dodawanie pewnej złożoności do potoków uczenia maszynowego;
 - często utrudniona interpretacja przekształconych cech.
2. Termin „klątwa wymiarowości” odnosi się do faktu, że wiele problemów nieistniejących w przestrzeni małymiarowej ujawnia się w przestrzeni wielowymiarowej. W dziedzinie uczenia maszynowego objawia się to głównie w postaci wielowymiarowych wektorów, które często są bardzo rzadkie, przez co zwiększały ryzyko przetrenowania modelu, a do tego bardzo trudno rozpoznać wzorce danych w przypadku korzystania z niewielkich zestawów przykładów uczących.
3. Po zredukowaniu wymiarowości zbioru danych za pomocą jednego z omówionych algorytmów odwrócenie tej operacji okazuje się niemal niemożliwe, ponieważ w jej trakcie zostaje utracona część informacji. Ponadto niektóre algorytmy (np. analiza PCA) zawierają prostą procedurę odwrotnej transformacji, pozwalającą na w miarę wierne odtworzenie pierwotnego zestawu danych, natomiast inne (takie jak algorytm t-SNE) nie mają takiej możliwości.
4. Analizę PCA możemy wykorzystywać do znacznej redukcji wymiarowości większości zestawów danych, nawet jeśli cechują się nieliniowością, ponieważ dzięki niej możemy pozbyć się przy najmniej nieprzydatnych wymiarów. Jeśli jednak wszystkie wymiary są użyteczne, jak np. w zbiorze danych Swiss roll, to redukcja wymiarowości za pomocą analizy PCA spowoduje nadmierną utratę informacji. Chcemy rozwinąć, a nie zgnieść „roladę”.
5. Podchwytliwe pytanie. To zależy od zbioru danych. Przyjrzymy się dwóm skrajnym przykładom. Najpierw założymy, że mamy do czynienia ze zbiorzem danych tworzących punkty ułożone niemal idealnie w jednej linii. W takim przypadku algorytm PCA może zredukować ten zestaw danych do jednego wymiaru przy zachowaniu 95% wariancji. Teraz wyobraź sobie, że zbiór danych zawiera doskonale losowo rozłożone punkty porozrzucane po 1000 wymiarów. W tej sytuacji potrzebowalibyśmy około 950 wymiarów, żeby zachować wariancję rzędu 95%. Zatem odpowiedź brzmi: to zależy od zbioru danych, a liczba ta może mieścić się w przedziale od 1 do 950. Jednym ze sposobów określenia rzeczywistej wymiarowości zestawu danych jest wygenerowanie wykresu wariancji wyjaśnionej w funkcji liczby wymiarów.
6. Klasyczna analiza PCA stanowi domyślny algorytm, działa jednak tylko wtedy, gdy zbiór danych mieści się w pamięci. Przyrostowa analiza PCA przydaje się do dużych zbiorów danych nieszczepiących się w pamięci, ale jest wolniejsza od klasycznej odmiany, zatem w przypadku zestawów danych, które można przechowywać w pamięci, należy stosować standardową analizę PCA. Jak sama nazwa wskazuje, przyrostowa analiza PCA przydaje się również w zadaniach przyrostowych, gdy trzeba na bieżąco przetwarzać nowe przykłady. Losowa analiza PCA nadaje się do znacznej redukcji wymiarowości danych mieszczących się w pamięci — w takich sytuacjach jest znacznie szybsza od klasycznego algorytmu PCA. Z kolei jądrowa analiza PCA jest używana wobec zestawów danych nieliniowych.

7. Zgodnie z intuicją algorytm redukowania wymiarowości jest wydajny, jeśli jest w stanie usuwać z zestawu danych wiele wymiarów bez utraty zbyt wielu informacji. Jedną z metod sprawdzenia jego wydajności jest wprowadzenie odwrotnej transformacji i zmierzenie błędu rekonstrukcji. Jednak odwrotna transformacja nie jest dostępna dla wszystkich algorytmów redukcji wymiarowości. Ewentualnie jeśli wykorzystujemy redukcję wymiarowości jako jeden z etapów wstępnego przygotowywania danych dla innej metody uczenia maszynowego (np. klasyfikatora losowego lasu), to wystarczy po prostu zmierzyć wydajność tego drugiego algorytmu — jeśli w wyniku redukcji wymiarowości nie utraciliśmy zbyt wiele informacji, to algorytm uczenia maszynowego powinien uzyskiwać taką samą wydajność jak w przypadku nieprzekształconego zbioru danych.
8. Połączenie dwóch różnych algorytmów redukcji wymiarowości ma jak najbardziej sens. Popularnym przykładem jest wykonanie analizy PCA w celu szybkiego pozbycia się dużej liczby niepotrzebnych wymiarów, a następnie zastosowanie znacznie wolniejszego algorytmu redukcji wymiarowości, takiego jak np. LLE. Dzięki takiemu dwuetapowemu rozwiązaniu naprawdopodobniej uzyskamy takie same wyniki jak przy użyciu wyłącznie algorytmu LLE, ale w znacznie krótszym czasie.

Rozwiązania ćwiczeń 9. i 10. znajdziesz w notatniku Jupyter dostępnym na stronie

<ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 9. Techniki uczenia nienadzorowanego

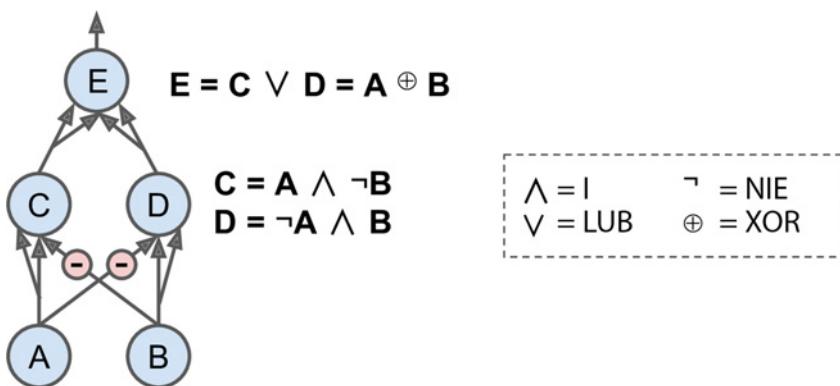
1. W uczeniu maszynowym analiza skupień stanowi zadanie nienadzorowanego grupowania podobnych przykładów. Definicja podobieństwa zależy od wyznaczonego zadania: na przykład w pewnych sytuacjach za podobne będą uznawane dwa przykłady znajdujące się blisko siebie, natomiast w innych przypadkach podobne przykłady mogą być od siebie znacznie oddalone, ale przynależeć do tej samej, gęsto zbitej grupy. Do najpopularniejszych algorytmów analizy skupień należą: algorytm centroidów, DBSCAN, aglomeracyjna analiza skupień, BIRCH, Mean-Shift, propagacja podobieństwa i widmowa analiza skupień.
2. Do głównych zastosowań analizy skupień należą: analiza danych, segmentacja klientów, systemy rekomendacji, silniki wyszukiwarek, segmentacja obrazu, uczenie półnadzorowane, redukcja wymiarowości, wykrywanie anomalii i wykrywanie szczegółów.
3. Reguła „łokcia” to prosta technika dobierania liczby skupień podczas korzystania z algorytmu centroidów: wystarczy utworzyć wykres bezwładności (średniej z kwadratów odległości od każdego przykładu do najbliższego centroidu) jako funkcji liczby skupień, a następnie znaleźć punkt na uzyskanej krzywej, w którym bezwładność przestaje gwałtownie maleć („łokieć”). Wartość ta zazwyczaj odpowiada optymalnej liczbie skupień. Inną metodą jest wygenerowanie wykresu wyniku profilu jako funkcji liczby skupień. Często pojawia się jakaś wartość szczytowa, w pobliżu której mieści się optymalna liczba skupień. Wynik profilu to nic innego jak średni współczynnik profilu ze wszystkich przykładów. Wartość tego współczynnika jest bliska +1 dla przykładów znajdujących się wewnątrz skupienia i oddalonych od innych skupień, a -1 dla przykładów znajdujących się bardzo blisko innego skupienia. Możesz także rysować wykresy profilu i przeprowadzać bardziej zaawansowaną analizę.

4. Oznaczanie zestawu danych jest kosztowne i czasochłonne. Dlatego często spotykamy się z sytuacją, gdy dysponujemy licznymi przykładami nieoznakowanymi, ale tylko kilkoma oznakowanymi. W technice propagacji etykiet kopiujemy część etykiet (lub wszystkie) z przykładów oznakowanych na podobne przykłady nieoznakowane. Rozwiążanie to może znacznie zwiększyć liczbę przykładów oznakowanych, dzięki czemu algorytm uczenia nadzorowanego może uzyskać lepszą skuteczność (jest to jedna z postaci uczenia półnadzorowanego). Można najpierw skorzystać z algorytmu analizy skupień (np. algorytmu centroidów) do pogrupowania wszystkich przykładów, następnie dla każdego skupienia znaleźć najczęściej występującą etykietę lub etykietę przypisaną do najbardziej reprezentatywnego przykładu (np. znajdującego się najbliżej centroidu) i rozprzestrzenić go na wszystkie nieoznakowane przykłady tworzące to skupienie.
5. Algorytmy centroidów i BIRCH skalują się dobrze do dużych zestawów danych, natomiast algorytmy DBSCAN i Mean-Shift poszukują obszarów o dużej gęstości.
6. Uczenie aktywne przydaje się wtedy, gdy masz dużo przykładów nieoznakowanych, ale proces ich oznaczania jest zbyt kosztowny. W takiej sytuacji (jest ona powszechna) nie wybiera się losowych przykładów do oznakowania, lecz przeprowadza się proces uczenia aktywnego, w którym żywici eksperci oddziałują z algorytmem uczenia i w razie potrzeby dostarczają etykiet dla określonych przykładów. Często spotykaną metodą jest próbkowanie niepewności (zob. ramkę „Uczenie aktywne”).
7. Wiele osób wymiennie używa pojęć **wykrywanie anomalii** i **wykrywanie szczegółów**, ale nie są one tożsame. W wykrywaniu anomalii algorytm jest uczony na zestawie danych, który może zawierać elementy odstające, a jego celem jest zazwyczaj ich wykrywanie (w zestawie danych uczących), jak również rozpoznawanie takich anomalii wśród nowych przykładów. Z kolei w wykrywaniu szczegółów algorytm jest trenowany na „czystym” zestawie danych, a jego zadaniem jest wykrywanie odstępstw jedynie wśród nowych przykładów. Niektóre algorytmy sprawują się lepiej w wykrywaniu anomalii (np. las izolacyjny), a inne radzą sobie lepiej z wykrywaniem szczegółów (np. jednoklasowa maszyna SVM).
8. Model mieszanin gaussowskich (GMM) to model probabilistyczny, w którym zakładamy, że przykłady zostały wygenerowane z mieszaniny kilku rozkładów gaussowskich o nieznanych parametrach. Innymi słowy, zgodnie z założeniem dane są pogrupowane w skończoną liczbę skupień, z których każde ma eliptyczny kształt (ale poszczególne skupienia mogą różnić się rodzajem elipsy, rozmiarami, ułożeniem i gęstością), a my nie wiemy, do którego skupienia należą poszczególne przykłady. Model ten przydaje się do szacowania gęstości, analizy skupień i wykrywania anomalii.
9. Jednym ze sposobów określenia właściwej liczby skupień podczas korzystania z modelu mieszanin gaussowskich jest określenie bayesowskiego kryterium informacyjnego (BIC) lub kryterium informacyjnego Akaikiego jako funkcji liczby skupień, a następnie wybór liczby skupień minimalizującej któreś z tych kryteriów. Inną techniką jest użycie bayesowskiego modelu mieszanin gaussowskich, w którym liczba skupień jest dobierana automatycznie.

Rozwiązania ćwiczeń od 10. do 13. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przykłady/uczem2.zip>.

Rozdział 10. Wprowadzenie do sztucznych sieci neuronowych i ich implementacji z użyciem interfejsu Keras

1. Odwiedź stronę TensorFlow Playground (<https://playground.tensorflow.org/>) i pobaw się dostępnymi elementami tak, jak zostało opisane w ćwiczeniu.
2. Mamy tu do czynienia z siecią neuronową opartą na oryginalnych sztucznych neuronach, obliczającą $A \oplus B$ (gdzie \oplus oznacza operację XOR), w której został wykorzystany fakt, że $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. Istnieją także inne rozwiązania, np. $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$ lub $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$.



3. Klasyczny perceptron jest zbieżny jedynie w przypadku zbioru danych rozdzielnych liniowo i nie jest w stanie szacować prawdopodobieństwa przynależności do klas. Natomiast klasyfikator regresji logistycznej dąży do dobrego rozwiązania, nawet jeśli dane nie są rozdzielne liniowo, i oblicza prawdopodobieństwo przynależności do klas. Jeżeli zmienimy w perceptronie funkcję aktywacji na logistyczną (lub typu softmax w przypadku obecności wielu neuronów) i wyuczymy go za pomocą gradientu prostego (lub dowolnego innego algorytmu minimalizującego funkcję kosztu, np. entropii krzyżowej), to stanie się on równoważny klasyfikatorowi regresji logistycznej.
4. Logistyczna funkcja aktywacji stanowiła kluczowy składnik uczenia pierwszych perceptronów wielowarstwowych, gdyż jej pochodna zawsze jest niezerowa, przez co algorytm gradientu prostego zawsze może schodzić wzduż jej nachylenia. W przypadku skokowej funkcji aktywacji algorytm gradientu prostego nie może się w ogóle ruszyć, gdyż nie ma nachylenia, po którym mógłby schodzić.
5. Do popularnych funkcji aktywacji należą: funkcja skoku jednostkowego, funkcja logistyczna (sigmoidalna), funkcja tangensa hiperbolicznego (tanh) oraz funkcja ReLU (rysunek 10.8). W rozdziale 11. zostały opisane inne przykłady, takie jak funkcja ELU i warianty funkcji ReLU.
6. Przeanalizujmy architekturę MLP opisaną w pytaniu: założymy, że mamy do czynienia z perceptronem wielowarstwowym zawierającym warstwę wejściową składającą się z dziesięciu per-

ceptronów przechodnych, następnie warstwę ukrytą tworzoną przez 50 neuronów oraz trójneuronową warstwę wyjściową. Wszystkie sztuczne neurony wykorzystują funkcję aktywacji ReLU.

- a. Wymiary macierzy wejściowej X to $m \times 10$, gdzie m symbolizuje rozmiar grupy danych uczących.
 - b. Wymiary wektora wag warstwy ukrytej W_u to 10×50 , natomiast długość jej wektora obciążenia \mathbf{b}_u to 50.
 - c. Wymiary wektora wag warstwy wyjściowej W_{wy} to 50×3 , a długość jej wektora obciążenia \mathbf{b}_{wy} to 3.
 - d. Wymiary macierzy wyjściowej Y to $m \times 3$.
 - e. $Y^* = \text{ReLU}(\text{ReLU}(XW_u + \mathbf{b}_u)W_{wy} + \mathbf{b}_{wy})$. Przypominam, że funkcja aktywacji ReLU jedynie zeruje każdą ujemną wartość w macierzy. Zwrót także uwagi, że po dodaniu wektora obciążenia do macierzy jest ona dołączana do każdego jej wiersza (tzw. **transmisja** — ang. *broadcasting*).
7. Do klasyfikowania wiadomości e-mail jako spamu lub hamu wystarczy tylko jeden neuron w warstwie wyjściowej — określający prawdopodobieństwo przynależności danej wiadomości do spamu. Podczas szacowania prawdopodobieństwa przynależności do klasy zazwyczaj wykorzystujemy w warstwie wyjściowej logistyczną funkcję aktywacji. Jeśli wolisz skorzystać z zestawu danych MNIST, należy w warstwie wyjściowej umieścić dziesięć neuronów i zastąpić funkcję logistyczną funkcją softmax, która radzi sobie z wieloma klasami naraz i generuje prawdopodobieństwo przynależności do każdej z nich. Gdybyśmy chcieli, żeby nasza sieć neuronowa przewidywała ceny domów (tak jak w rozdziale 2.), to musielibyśmy stworzyć jednoneuronową warstwę wyjściową niezawierającą żadnej funkcji aktywacji³.
8. Technika propagacji wstecznej służy do uczenia sztucznych sieci neuronowych. Najpierw obliczane są gradienty funkcji kosztu w odniesieniu do każdego parametru modelu (wszystkich wag i obciążień), po czym zostaje wykonana metoda gradientu prostego za pomocą tych gradientów. Ten etap propagacji wstecznej jest zazwyczaj powtarzany tysiące albo miliony razy przy użyciu wielu grup danych uczących aż do osiągnięcia zbieżności parametrów modelu minimalizującej funkcję kosztu. W celu wyliczenia gradientów algorytm propagacji wstecznej wykorzystuje tryb odwrotnego różniczkowania automatycznego (choćż gdy algorytm ten został wymyślony — a był wymyślany kilkukrotnie — tak go nie nazywano). Operacja odwrotnego różniczkowania wykonuje przebieg do przodu po grafie obliczeniowym, wylicza wartość każdego węzła dla bieżącej grupy danych uczących, a następnie przeprowadza przebieg wsteczny i jednocześnie oblicza wszystkie gradienty (zob. dodatek D). Na czym więc polega różnica? Technika propagacji wstecznej odnosi się do całego procesu uczenia sztucznej sieci neuronowej za pomocą wielu operacji propagacji wstecznej, z których każda służy do obliczania gradientów, używanych z kolei w algorytmie gradientu prostego. Technika odwrotnego różniczkowania automatycznego służy wyłącznie do wydajnego obliczania gradientów i jest stosowana w metodzie propagacji wstecznej.

³ Gdy prognozowane wartości mogą różnić się wieloma rzędami wielkości, warto obliczać przewidywania wartości docelowej w skali logarytmicznej, a nie bezpośrednio. Już samo obliczenie funkcji eksponencjalnej wyniku sieci neuronowej pozwoli nam uzyskać oszacowaną wartość (gdyż $\exp(\log v) = v$).

9. Oto lista wszystkich parametrów, które można dostrajać w podstawowej architekturze MLP: liczba warstw ukrytych, liczba neuronów w każdej warstwie ukrytej oraz funkcje aktywacji użyte w każdej warstwie ukrytej i wyjściowej⁴. Zasadniczo dobrym rozwiązaniem domyślnym w warstwach ukrytych jest warstwa ReLU (lub jej odmiany opisane w rozdziale 11.). W przypadku warstwy wyjściowej najczęściej używamy funkcji logistycznej podczas klasyfikacji binarnej, funkcji softmax do klasyfikacji wieloklasowej lub zupełnie ją pomijamy w zadaniach regresji. Jeżeli sieć MLP wykazuje oznaki przetrenowania, możesz zmniejszyć liczbę warstw ukrytych i ograniczyć liczbę neuronów w każdej z nich.
10. Rozwiązanie znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 11. Uczenie głębokich sieci neuronowych

1. Nie, wszystkie wagi powinny być losowane niezależnie od siebie; nie powinny zawierać takiej samej wartości początkowej. Ważnym celem losowego inicjowania wag jest uniknięcie symetryczności: gdyby wszystkie wagi miały taką samą wartość początkową, nawet niezerową, to zostaje zachowana symetria (tzn. wszystkie neurony w danej warstwie są takie same), a proces propagacji wstecznej nie zdoła jej złamać. Mówiąc konkretnie, w takiej sytuacji wszystkie neurony w danej warstwie będą zawsze miały takie same wagi. Byłoby to równoznaczne z obecnością w danej warstwie tylko jednego, znacznie powolniejszego neuronu. W takiej konfiguracji uzyskanie dobrej zbieżności jest praktycznie niemożliwe.
2. Nic nie zabrania inicjowania członów obciążenia z wartością 0. Niektórzy lubią inicjować je tak samo jak wagi i jest to równie dobre rozwiązanie — ich wpływ jest niemal taki sam.
3. Kilka zalet funkcji SELU w porównaniu do funkcji ReLU:
 - Może przyjmować wartości ujemne, zatem uśredniony wynik wszystkich neuronów w dowolnej warstwie jest zazwyczaj bliższy zeru w porównaniu do funkcji aktywacji ReLU (która nigdy nie daje w rezultacie ujemnych wartości). W ten sposób możemy złagodzić problem zanikających gradientów.
 - Jej pochodna jest zawsze niezerowa, dzięki czemu jesteśmy w stanie uniknąć problemu śmierci neuronów, który jest spotykany w jednostkach pobudzanych funkcją ReLU.
 - W odpowiednich warunkach (tzn. model jest sekwencyjny, wagi są inicjalizowane za pomocą inicjalizatora LeCuna, dane wejściowe zostały ustandaryzowane oraz nie występuje żadna warstwa niekompatybilna lub regularyzacja, taka jak porzucanie lub regularyzacja ℓ_1) funkcja SELU gwarantuje samonormalizację modelu, dzięki czemu unikamy problemu eksplodujących/zanikających gradientów.

⁴ W rozdziale 11. omawiam wiele technik wprowadzających dodatkowe hiperparametry: rodzaj inicjacji wag, hiperparametry funkcji aktywacji (np. stopień wycieku w „przeciekającej” funkcji ReLU), próg przycinania gradientów, typ optymalizatora i jego hiperparametry (np. hiperparametr momentowy podczas korzystania z optymalizatora MomentumOptimizer), rodzaj regularyzacji stosowanej w każdej warstwie ukrytej oraz hiperparametry regularyzacji (np. współczynnik porzucania w metodzie porzucania) i tak dalej.

- 4.** Funkcja aktywacji SELU stanowi dobre domyślne rozwiązanie. Jeśli chcesz, żeby sieć neuronowa działała możliwie szybko, możesz zastąpić tę funkcję którymkolwiek z „przeciekających” wariantów (np. standardową „przeciekającą” funkcją ReLU wykorzystującą domyślną wartość hiperparametru). Wiele osób wybiera funkcję ReLU z powodu jej prostoty, pomimo faktu, że nie osiąga tak dobrych rezultatów jak funkcja SELU czy „przeciekająca” funkcja ReLU. Jednakże w niektórych sytuacjach przydaje się właściwość funkcji ReLU do generowania wartości 0 na wyjściu (zob. rozdział 17.). Do tego są pod nią zoptymalizowane niektóre rozwiązania programistyczne i sprzętowe. Funkcja tangensa hiperbolicznego (tanh) może być przydatna w warstwie wyjściowej, jeśli chcemy, żeby były generowane wyniki w zakresie od -1 do 1, obecnie jednak rzadko jest spotykana w warstwach ukrytych. Z kolei logistyczna funkcja aktywacji przydaje się do szacowania prawdopodobieństwa (np. w klasyfikacji binarnej), ale również ona rzadko jest wstawiana do warstw ukrytych (istnieją pewne wyjątki, np. warstwa kodowania w autokoderaach wariacyjnych; zob. rozdział 17.). Natomiast funkcja softmax jest wstawiana w warstwie wyjściowej do wyliczania prawdopodobieństwa przynależności do którejś z wzajemnie wykluczających się klas, ale poza tym bardzo rzadko (jeśli w ogóle) występuje w warstwie ukrytej.
- 5.** Jeśli korzystając z optymalizatora MomentumOptimizer wyznaczysz wartość hiperparametru momentum zbyt bliską 1 (np. 0,99999), to algorytm prawdopodobnie znacznie przyspieszy podczas dążenia w kierunku minimum globalnego, ale wskutek uzyskanego „pędzu” znajdzie się po jego przeciwej stronie. Następnie zwolni i skieruje się w przeciwną stronę, znów minie minimum itd. Może w ten sposób długo oscylować, zanim osiągnie zbieżność, dlatego ostatecznie proces ten zajmie znacznie więcej czasu niż w przypadku mniejszej wartości tego hiperparametru.
- 6.** Jedną z metod uzyskania modelu rzadkiego (tzn. zawierającego większość wag równych零) jest wyuczenie modelu w standardowy sposób, a następnie wyzerowanie wszystkich małych wag. W celu otrzymania jeszcze większego rozrzedzenia możesz wprowadzić regularyzację ℓ_1 podczas treningu, dzięki czemu optymalizator będzie dążył do modelu rzadkiego. Trzecim rozwiązaniem jest wykorzystanie narzędzia TensorFlow Model Optimization Toolkit.
- 7.** Tak, metoda porzucania mniej więcej dwukrotnie spowalnia proces uczenia. Jednak nie ma ona wpływu na proces wnioskowania, gdyż jest stosowana wyłącznie podczas treningu. Porzucanie metodą Monte Carlo działa tak samo jak klasyczny algorytm, ale jest przeprowadzane także w trakcie wnioskowania, więc ten proces również zostaje nieznacznie spowolniony. Jednak, co ważniejsze, korzystając z porzucania metodą MC, chcesz zazwyczaj przeprowadzać wnioskowanie co najmniej dziesięć razy, aby uzyskać lepsze prognozy. Oznacza to co najmniej dziesięciokrotne spowolnienie fazy wnioskowania.

Rozwiązanie do ćwiczenia 8. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 12. Modele niestandardowe i uczenie za pomocą modułu TensorFlow

- 1.** TensorFlow to biblioteka o otwartym kodzie przeznaczona do obliczeń numerycznych, szczególnie dobrze dopasowana do wielkoskalowego uczenia maszynowego. Jej podstawowa struktura przypomina moduł NumPy, ale zawiera także obsługę procesorów graficznych, obliczeń

rozproszonych, funkcje analizowania i optymalizowania grafów obliczeniowych (z dostępem do podręcznego formatu grafów, który umożliwia uczenie modelu w jednym środowisku i jego uruchamianie w innym), interfejs optymalizujący oparty na automatycznym różniczkowaniu odwrotnym, a także kilka potężnych API, takich jak `tf.keras`, `tf.data`, `tf.image`, `tf.signal` itd. Do popularnych bibliotek uczenia głębokiego należą też PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2 i Chainer.

2. Moduł TensorFlow ma najczęściej cech wspólnych z biblioteką NumPy, ale z kilku powodów biblioteki te nie są swoimi zamiennikami. Po pierwsze, nie zawsze pokrywają się nazwy funkcji (np. `tf.reduce_sum()` i `np.sum()`). Po drugie, niektóre funkcje nie działają tak samo (np. funkcja `tf.transpose()` tworzy transponowaną kopię tensora, natomiast atrybut `T` z modułu NumPy tworzy widok transponowany, ale w rzeczywistości nie kopiuje danych). Po trzecie, tablice NumPy są mutowalne, w przeciwieństwie do tensorów TensorFlow (ale jeśli potrzebujesz obiektu mutowalnego, możesz użyć `tf.Variable`).
3. Zarówno `tf.range(10)`, jak i `tf.constant(np.arange(10))` zwracają jednowymiarowy tensor zawierający liczby całkowite od 0 do 9. Różnica polega na tym, że pierwsza funkcja wykorzystuje 32-bitowe, a druga 64-bitowe wartości stałoprzecinkowe. Istotnie, domyślnie w module TensorFlow dane są 32-bitowe, a w NumPy — 64-bitowe.
4. Moduł TensorFlow oprócz standardowych tensorów obsługuje także kilka innych struktur danych, w tym takie jak tensory rzadkie, tablice tensorów, tensory nierówne, kolejki, tensory znakowe i zbiory. Dwa ostatnie elementy są w istocie reprezentowane jako standardowe tensory, ale moduł TensorFlow zawiera funkcje specjalne służące do manipulowania nimi (w klasach `tf.strings` i `tf.sets`).
5. Definiując niestandardową funkcję straty, zazwyczaj chcemy ją zaimplementować jako standar-dową funkcję Pythona. Jeśli jednak taka funkcja straty musi obsługiwać jakieś hiperparametry (albo dowolny inny stan), należy utworzyć podklasę klasy `keras.losses.Loss` i zaimplemen-tować metody `_init_()` i `call()`. Jeśli chcesz, aby hiperparametry funkcji straty zostały zapisane wraz z modelem, to musisz również zaimplementować metodę `get_config()`.
6. Podobnie jak w przypadku funkcji straty, większość wskaźników można zdefiniować w posta-ci standardowych funkcji Pythona. Jeśli jednak chcesz, aby wskaźnik obsługiwał jakieś hiperpara-metry (albo dowolny inny stan), to należy utworzyć podklasę klasy `keras.metrics.Metric`. Ponadto jeżeli obliczenie wskaźnika dla całej epoki nie jest równoważne obliczeniu średniego wskaźnika dla wszystkich grup w tej epoce (np. tak jest w przypadku wskaźników precyzji i pełności), należy utworzyć podklasę klasy `keras.metrics.Metric` i zaimplementować meto-dy `_init_()`, `update_state()` i `result()`, aby móc śledzić przebieg wskaźnika w danej epoce. Trzeba także zaimplementować metodę `reset_states()`, chyba że jej jedynym zadaniem byłoby zerowanie wszystkich zmiennych. Jeżeli chcesz, aby stan był zapisywany wraz z modelem, musisz zaimplementować również metodę `get_config()`.
7. Należy odróżnić składniki wewnętrzne modelu (tzn. warstwy lub wielokrotnie używane bloki warstw) od samego modelu (czyli obiektu, który chcesz trenować). W pierwszym przypadku należy utworzyć podklasę klasy `keras.layers.Layer`, a w drugim — podklasę klasy `keras.models.Model`.
8. Tworzenie własnej niestandardowej pętli uczenia jest dość skomplikowane, dlatego trzeba to robić jedynie wtedy, gdy nie ma innego wyjścia. Interfejs Keras zawiera kilka narzędzi umożliwiających

modyfikowanie treningu bez konieczności pisania własnej pętli uczenia: wywołania zwrotne, niestandardowe regularizatory, niestandardowe ograniczenia, niestandardowe funkcje straty itd. Jeśli to możliwe, korzystaj z nich zamiast tworzyć niestandardową pętlę uczenia, własna pętla uczenia jest bowiem narażona na obecność większej liczby błędów i znacznie trudniej jest korzystać z takiego niestandardowego kodu w innych modelach. Jednakże w pewnych sytuacjach tworzenie niestandardowej pętli uczenia okazuje się koniecznością — możesz na przykład chcieć używać różnych optymalizatorów w oddzielnych częściach sieci neuronowej, co zostało wykorzystane w sieci Wide & Deep (<https://arxiv.org/abs/1606.07792>). Niestandardowa pętla uczenia przydaje się także do wykrywania i usuwania błędów oraz do zrozumienia dokładnego mechanizmu uczenia.

9. Niestandardowe składniki interfejsu Keras należy tworzyć w taki sposób, aby można było je przekształcać w funkcje TF, co oznacza, że powinny w maksymalnym stopniu bazować na operacjach TF i przestrzegać wszystkich reguł wymienionych w punkcie „Reguły związane z funkcją TF”. Jeżeli naprawdę musisz umieścić dowolny kod Pythona w niestandardowym składniku, możesz go na przykład wstawić do operacji `tf.py_function()` (ale zmniejszy to wydajność i przenośność modelu) albo wyznaczyć `dynaminc=True` podczas tworzenia niestandardowej warstwy lub modelu (ewentualnie wprowadzić `run_eagerly=True` podczas wywoływania metody `compile()` modelu).
10. Listę reguł, których należy przestrzegać podczas tworzenia funkcji TF, znajdziesz w punkcie „Reguły związane z funkcją TF”.
11. Tworzenie modelu dynamicznego Keras przydaje się do usuwania błędów, ponieważ żaden niestandardowy element nie będzie komplikowany do funkcji TF i aby sprawdzić kod, możesz korzystać z dowolnego debugera Pythona. Można korzystać z niego również wtedy, gdy chcesz wprowadzić dowolny kod Pythona do swojego modelu (lub algorytmu uczącego) i umieszczać wywołania do zewnętrznych bibliotek. Aby uczynić model dynamiczny, wyznacz `dynaminc=True` podczas jego tworzenia. Możesz ewentualnie wprowadzić `run_eagerly=True` podczas wywoływania metody `compile()` modelu. Model dynamiczny uniemożliwia interfejsowi Keras wykorzystywanie funkcji grafowych modułu TensorFlow, co spowoduje spowolnienie procesów uczenia i wnioskowania, Ty zaś nie będziesz mogła/mógl eksportować grafu obliczeniowego, co oznacza ograniczenie przenośności modelu.

Rozwiązań ćwiczeń 9. – 11. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 13. Wczytywanie i przetwarzanie wstępne danych za pomocą modułu TensorFlow

1. Skuteczne wprowadzanie i wstępne przetwarzanie dużej ilości danych bywa skomplikowanym wyzwaniem inżynierijnym. Interfejs danych znacznie ułatwia to zadanie. Zawiera on wiele funkcji, m.in. mechanizm wczytywania danych z różnych źródeł (np. plików tekstowych lub binarnych), równolegle odczytywanie danych z wielu źródeł, przekształcanie tych danych, przeplatanie rekordów, tasowanie danych, ich grupowanie i wstępne wczytywanie.

2. Rozdzielanie dużego zestawu danych na wiele plików umożliwia ogólne przetasowanie tych danych, po którym można przejść do tasowania bardziej szczegółowego za pomocą bufora tasowania. W ten sposób możemy również przechowywać olbrzymie zestawy danych, które nie mieszczą się na jednym komputerze. Łatwiej także manipulować wieloma małymi plikami niż jednym dużym; prościej na przykład rozdzielać dane na wiele podzbiorów. Jeżeli dane są podzielone na wiele plików rozmieszczonego na różnych serwerach, można jednocześnie pobierać wiele plików z poszczególnych serwerów, dzięki czemu optymalizujemy obciążenie łącza.
3. Za pomocą aplikacji TensorBoard możesz wizualizować dane profilu: jeżeli procesor graficzny nie jest w pełni wykorzystywany, to prawdopodobnie w potoku wejściowym będzie występował zator. Możesz rozwiązać ten problem poprzez zrównoleglenie odczytywania i wstępnego przetwarzania danych w wielu wątkach, a także wstępne wczytywanie kilku grup. Jeżeli to nie wystarczy, aby w stu procentach zapiąć GPU do pracy w fazie uczenia, zoptymalizuj kod przetwarzania wstępnego. Możesz też zapisać zestaw danych w wielu plikach TFRecord, a w razie potrzeby przeprowadzić pewne czynności przetwarzania wstępnego jeszcze przed rozpoczęciem treningu, dzięki czemu zaoszczędzisz trochę czasu i mocy obliczeniowej podczas uczenia (przydaje się do tego projekt TF Transform). Jeżeli trzeba, możesz skorzystać z komputera wyposażonego w większą liczbę CPU i więcej pamięci operacyjnej oraz w GPU o odpowiednio dużej przepustowości.
4. Plik TFRecord składa się z sekwencji dowolnych rekordów binarnych: możesz przechowywać wszelkiego rodzaju dane binarne w każdym rekordzie. W praktyce jednak większość plików TFRecord przechowuje sekwencje serializowanych buforów protokołów. Dzięki temu możesz korzystać z zalet tych buforów, np. łatwego ich odczytywania na różnych platformach i w różnych językach, a ich definicje można aktualizować na zasadzie kompatybilności wstępnej.
5. Format Example bufora protokołów ma tę zaletę, że moduł TensorFlow zawiera operacje umożliwiające analizę jego składni (funkcje `tf.io.parse*example()`) bez konieczności definiowania własnego formatu. Jest on wystarczająco elastyczny, aby można było za jego pomocą reprezentować przykłady z większości zestawów danych. Jeżeli jednak nie będzie spełniał Twoich oczekiwani, możesz zdefiniować swój własny bufor protokołów, skompilować go za pomocą narzędzia protoc (argumenty `-descriptor_set_out` i `--include_imports` pozwalają eksportować deskryptora tego bufora protokołów) i użyć funkcji `tf.io.decode_proto()` do analizowania składni serializowanych buforów protokołów (przykład znajdziesz w sekcji „Niestandardowy bufor protokołów” w notatniku Jupyter). Jest to bardziej skomplikowane zadanie i wymaga wdrożenia deskryptora wraz z modelem, ale można tego dokonać.
6. Korzystając z formatu TFRecord, zazwyczaj chcemy stosować kompresję w przypadku pobierania plików TFRecord przez skrypt uczenia, gdyż kompresja zmniejsza rozmiary tych plików i skracą czas pobierania. Jeżeli jednak pliki te znajdują się na tym samym komputerze co skrypt, lepiej pozostawić wyłączoną kompresję, ponieważ rozpakowywanie plików pochłania część mocy obliczeniowej procesora.
7. Przyjrzyjmy się zaletom i wadom poszczególnych opcji przetwarzania wstępnego:
 - Jeżeli przetwarzasz wstępnie dane podczas tworzenia plików danych, skrypt uczenia będzie działał szybciej, ponieważ nie będzie musiał zajmować się tym procesem na bieżąco. W pewnych przypadkach przetworzone dane bywają także znacznie mniejsze od danych pierwotnych, możesz więc zaoszczędzić czas i przyspieszyć pobieranie plików. Niekiedy przydaje

się również materializowanie przetworzonych wstępnie danych, na przykład w celach analitycznych lub archiwizacyjnych. Rozwiążanie to ma jednak też pewne wady. Po pierwsze, nie jest łatwo eksperymentować z różnymi mechanizmami przetwarzania wstępnego, jeśli trzeba generować przetworzony zestaw danych dla każdego wariantu. Po drugie, jeżeli zależy Ci na dogenerowaniu danych, musisz materializować wiele wariantów zestawu danych, co wymaga dużo miejsca na dysku i dużo czasu. Po trzecie, wytrenowany model będzie oczekiwał przetworzonych danych, zatem musisz dołączyć kod przetwarzania wstępnego do aplikacji przed etapem wywołania modelu.

- Jeżeli dane są przetwarzane wstępnie w ramach potoku `tf.data`, znacznie łatwiej dostosować logikę tego procesu i stosować dogenerowanie danych. Poza tym dzięki interfejsowi `tf.data` możemy z łatwością konstruować wydajne potoki przetwarzania wstępnego (np. wykorzystujące wielowątkowość i wstępne wczytywanie danych). Jednakże przetwarzanie wstępne danych w ten sposób spowolni proces uczenia. Ponadto jeżeli dane były przetwarzane wstępnie podczas tworzenia plików danych, to każdy przykład uczący będzie przetwarzany wstępnie raz na epokę. Poza tym model będzie ciągle oczekiwał wstępnie przetworzonych danych.
- Jeżeli dodasz warstwy przetwarzania wstępnego do modelu, to wystarczy wspólny kod zarówno dla fazy uczenia, jak i dla fazy wnioskowania. Jeśli model ma zostać wdrożony do wielu różnych platform, nie będziesz musiał wielokrotnie pisać kodu przetwarzania wstępnego. Ponadto nie będzie istniało ryzyko korzystania z niewłaściwego mechanizmu przetwarzania wstępnego, ponieważ odpowiedni kod będzie stanowił część modelu. Z drugiej strony wstępne przetwarzanie danych spowolni proces uczenia, a każdy przykład uczący będzie wstępnie przetwarzany w każdej epoce. Ponadto domyślne operacje przetwarzania wstępnego będą dla bieżącej grupy realizowane przez procesor graficzny (nie będziesz korzystać z zalet przetwarzania równoległego na procesorze głównym i wczytywania wstępnego). Na szczęście przygotowywane warstwy przetwarzania wstępnego z interfejsu Keras powinny być w stanie uruchamiać operacje przetwarzania wstępnego jako część potoku `tf.data`, dzięki czemu będziesz mogła/mógli wykorzystywać możliwości procesora głównego (wielowątkowość i wczytywanie wstępne).
- Korzystanie z narzędzia TF Transform we wstępny przetwarzaniu danych gwarantuje wiele zalet opisanych w poprzednich podpunktach: przetwarzane dane są materializowane, każdy przykład jest przetwarzany wstępnie tylko raz (co przyspiesza proces uczenia), a warstwy przetwarzania wstępnego są generowane automatycznie, więc musisz stworzyć kod przetwarzania wstępnego tylko raz. Główną wadą jest konieczność nauki korzystania z tego narzędzia.

8. Zobaczmy, jak można kodować dane kategorialne i tekst:

- Aby zakodować cechę kategorialną wyróżniającą się jakąś naturalną kolejnością, np. oceną filmu („dobry”, „przeciętny”, „zły”), najłatwiej jest wykorzystać kodowanie porządkowe: uszereguj kategorie w ich naturalnej kolejności i odwzoruj każdą kategorię za pomocą rangi (np. kategoria „dobry” zostaje odwzorowana jako 0, „przeciętny” jako 1, a „zły” jako 2). Jednak większość cech kategorialnych nie jest uporządkowana w jakiś naturalny sposób. Na przykład kraje czy profesje nie mają naturalnej kolejności. Wówczas możesz skorzystać

z kodowania gorącojedynkowego albo, w przypadku występowania wielu kategorii, wektorów właściwościowych.

- W przypadku tekstu jedną z metod jest reprezentacja „worka słów”: zdanie jest reprezentowane jako wektor zawierający zliczenia wystąpień każdego słowa. Najpowszechniejsze wyrazy zazwyczaj nie są bardzo ważne, dlatego zmniejszamy ich wagę za pomocą techniki TF-IDF. Często zamiast słów zliczane są n -gramy, czyli sekwencje n kolejnych wyrazów — jest to proste i skuteczne rozwiązywanie. Ewentualnie możemy zakodować każde słowo za pomocą reprezentacji właściwościowych słów, które mogą nawet być uprzednio wytrenowane. Zamiast wyrazów możemy również kodować poszczególne litery albo inne elementy składowe wyrazów (np. poprzez podzielenie wyrazu „najlepszy” na „naj” i „lepszy”). Dwa ostatnie rozwiązania zostały omówione w rozdziale 16.

Rozwiązań ćwiczeń 9. i 10. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 14. Głębokie widzenie komputerowe za pomocą splotowych sieci neuronowych

1. Główne zalety splotowej sieci neuronowej w porównaniu do w pełni połączonej sieci neuronowej w zadaniu klasyfikacji obrazów:

- Sieć CNN zawiera znacznie mniej parametrów niż w pełni połączona sieć neuronowa, ponieważ sąsiadujące ze sobą warstwy są tylko częściowo połączone, a do tego wagi są wielokrotnie używane. W ten sposób proces uczenia przebiega o wiele szybciej w sieci splotowej, spada ryzyko przetrenowania oraz wymagana jest znacznie mniejsza ilość danych uczących.
- Gdy sieć CNN nauczy się jądra rozpoznającego określona cechę, będzie ona wykrywana w dowolnym obszarze obrazu. Z drugiej strony, jeśli głęboka sieć neuronowa zacznie rozpoznawać cechę w danym rejonie obrazu, to będzie ją rozpoznawać tylko w tym obszarze. Obrazy najczęściej składają się z bardzo powtarzalnych cech, dlatego sieci splotowe znacznie lepiej generalizują wyniki niż w pełni połączona sieć w zadaniach przetwarzania obrazów, takich jak klasyfikacja, przy jednoczesnym wykorzystaniu mniejszej liczby przykładów uczących.
- W pełni połączona sieć nie „zna” rozmieszczenia pikseli — nie wie, że sąsiadujące piksele są ze sobą powiązane. Wiedza ta stanowi część architektury CNN. Niższe warstwy zazwyczaj rozpoznają cechy w małych obszarach obrazu, natomiast wyższe warstwy składają je w bardziej skomplikowane wzorce. Rozwiązanie to sprawdza się znakomicie w większości rzeczywistych zdjęć, dzięki czemu w kwestiach decyzyjnych sieci splotowe znacznie wyprzedzają głębokie sieci neuronowe.

2. Obliczmy liczbę parametrów w sieci CNN. Jej pierwsza warstwa splotowa składa się z jąder 3×3 , natomiast na wejściu znajdują się trzy kanały (czerwony, zielony i niebieski), zatem każda mapa cech będzie zawierała $3 \times 3 \times 3$ wag i jeden człon obciążenia. Wychodzi więc 28 parametrów na mapę cech. Pierwsza warstwa splotowa ma 100 map cech, czyli 2800 parametrów. W drugiej warstwie występują również jądra 3×3 , a jej wejścia tworzy 100 map cech uzyskanych w poprzedniej warstwie, czyli każda mapa cech w drugiej warstwie zawiera $3 \times 3 \times 100 = 900$ wag i człon obciążenia.

W tej warstwie występuje 200 map cech, co daje nam $901 \times 200 = 180\ 200$ parametrów. W trzeciej, ostatniej warstwie splotowej znowu mamy do czynienia z jądrami 3×3 , dla których na wejściach jest dostarczanych 200 map cech z poprzedniej warstwy, czyli każdą mapę cech opisuje $3 \times 3 \times 200 = 1800$ wag i jeden człon obciążenia. Mieści się tutaj 400 map cech, przez co mamy do czynienia z $1801 \times 400 = 720\ 400$ parametrami. Podsumowując: omawiana sieć CNN zawiera $2800 + 180\ 200 + 720\ 400 = 903\ 400$ parametrów.

Sprawdźmy teraz, ile pamięci RAM (co najmniej) będzie potrzebowała ta sieć neuronowa do wyliczenia prognozy dla jednej próbki. Najpierw przeprowadźmy obliczenia rozmiaru mapy cech w każdej warstwie. Korzystamy z kroku o wartości 2 i uzupełniania zerami w trybie "same", więc szerokość i wysokość mapy cech są dzielone przez 2 w każdej warstwie (i w razie potrzeby zaokrąglane w góre), zatem kanały wejściowe mają rozmiar 200×300 pikseli, mapy cech w pierwszej warstwie ukrytej — 100×150 pikseli, w drugiej warstwie — 50×75 pikseli, a w ostatniej — 25×38 . Pamiętamy, że 32 bity to 4 bajty, a w pierwszej warstwie mamy 100 map cech, zatem będzie ona zajmować $4 \times 100 \times 150 \times 100 = 6$ milionów bajtów (6 MB). Druga warstwa wymaga $4 \times 50 \times 75 \times 200 = 3$ miliony bajtów (3 MB). Trzecia warstwa zaś pochłania $4 \times 25 \times 38 \times 400 = 1\ 520\ 000$ bajtów (1,5 MB). Jednak po wykonaniu obliczeń przez daną warstwę pamięć zajęta przez warstwę poprzednią może zostać zwolniona, więc jeśli cała architektura jest bardzo dobrze zoptymalizowana, będzie wymagane zaledwie $6+3 = 9$ milionów bajtów, czyli 9 MB (w momencie wyliczenia drugiej warstwy, ale jeszcze przed zwolnieniem pamięci w pierwszej warstwie). To jednak nie wszystko: musimy doliczyć jeszcze pamięć zajmowaną przez parametry sieci. Wiemy, że jest ich 903 400, z których każdy wykorzystuje 4 bajty, co oznacza dodatkowych 3 613 600 bajtów (ok. 3,6 MB). Podsumowując: całkowita ilość wymaganej pamięci operacyjnej wynosi (co najmniej) 12 613 600 bajtów (ok. 12,6 MB).

Na koniec obliczymy minimalną ilość pamięci operacyjnej wymaganą podczas uczenia sieci CNN za pomocą minigrupy składającej się z 50 próbek. W trakcie uczenia moduł TensorFlow wykorzystuje algorytm propagacji wstecznej, co wymaga przechowywania wszystkich wartości obliczonych podczas przebiegu do przodu aż do momentu rozpoczęcia przebiegu wstecznego. Musimy więc obliczyć całkowitą przestrzeń pamięci RAM wymaganą przez wszystkie warstwy i pomnożyć ją przez 50! W tym momencie przejdziemy z kilobajtów na megabajty. Wiemy już, że każdy przykład zajmuje w każdej warstwie odpowiednio 6, 3 i 1,5 MB, łącznie 10,5 MB. Zatem 50 przykładów zajmie 525 MB pamięci operacyjnej. Dodajmy do tego pamięć przeznaczoną na obrazy wejściowe: $50 \times 4 \times 200 \times 300 \times 3 = 36$ milionów bajtów (36 MB), wartości parametrów modelu (obliczone wcześniej 3,6 MB) oraz pewną ilość zarezerwowaną dla gradientów (możemy ją pominąć, gdyż może być ona stopniowo uwalniana wraz z przebiegiem propagacji wstecznej). Wychodzi nam zatem w przybliżeniu $525 + 36 + 3,6 = 564,6$ MB, i to przy naprawdę optymistycznym, minimalistycznym założeniu.

3. Jeśli na Twojej karcie graficznej kończy się pamięć podczas uczenia sieci CNN, możesz wykonać pięć następujących czynności w celu rozwiązania problemu (oprócz zakupu karty graficznej mającej więcej pamięci operacyjnej):
 - zmniejszyć rozmiar minigrupy;
 - zredukować wymiarowość, wprowadzając większy krok w przynajmniej jednej warstwie;
 - usunąć co najmniej jedną warstwę;

- stosować 16-bitowe, a nie 32-bitowe liczby zmiennoprzecinkowe;
 - rozdzielić sieć CNN pomiędzy wiele urządzeń.
4. Maksymalizująca warstwa łącząca nie zawiera żadnych parametrów, natomiast warstwa splotowa zawiera ich całkiem dużo (zob. poprzednie odpowiedzi).
5. Warstwa **normalizacji odpowiedzi lokalnej** sprawia, że neurony o największym pobudzeniu hamują działanie neuronów znajdujących się na tej samej pozycji, ale w sąsiadujących mapach cech, dzięki czemu poszczególne mapy cech dążą do specjalizacji i są rozdzielane, co zmusza je do przeszukiwania szerszego zakresu cech. Technika ta jest zazwyczaj stosowana w niższych warstwach w celu uzyskania większej puli podstawowych cech, które będą używane przez wyższe warstwy do łączenia ich w bardziej skomplikowane cechy.
6. Główne usprawnienia sieci AlexNet w porównaniu do LeNet5 to znacznie większy rozmiar i głębia architektury, a także tworzenie stosu warstw splotowych bezpośrednio jedna po drugiej zamiast przeplatania warstw stosowych z buforowymi. Główną nowinką w sieci GoogLeNet jest wprowadzenie **modułów incepcyjnych**, dzięki czemu sieć ta może być znacznie bardziej rozbudowana niż wcześniejsze jej odmiany, przy jednoczesnym korzystaniu z mniejszej liczby parametrów. Z kolei w architekturze ResNet zostały użyte połączenia pomijające, co pozwala uzyskać znacznie powyżej 100 warstw. Prawdopodobnie również jej prostota i jednorodność stanowią innowację. Podstawową nowością w sieci SENet jest wprowadzenie bloku SE (dwuwarstwowej sieci gęstej) po każdym module incepcyjnym lub jednostce rezydualnej w sieci ResNet, co pozwala przekalibrować względną istotność map cech. Z kolei w sieci Xception zastosowano głębokościowo rozdzielne warstwy splotowe, które oddziennie wyszukują wzorce przestrzenne i wzorce głębokościowe.
7. Pełne sieci splotowe to sieci neuronowe składające się wyłącznie z warstw splotowych i łączących. Są one w stanie skutecznie przetwarzać obrazy o dowolnych rozmiarach (przynajmniej przekraczających rozmiar minimalny). Najbardziej przydatą się do wykrywania obiektów i segmentacji semantycznej, ponieważ muszą sprawdzać obraz tylko raz (w przeciwieństwie do klasycznej sieci CNN, która musi przebiegać wielokrotnie przez poszczególne fragmenty obrazu). Jeżeli korzystasz z sieci splotowej zawierającej na końcu warstwy gęste, to po ich przekształcenia w warstwy splotowe otrzymasz pełną sieć konwolucyjną — wystarczy w miejscu najbliższej warstwy gęstej wstawić warstwę splotową o rozmiarze jądra równym rozmiarowi wejść tej warstwy, wyznaczyć po jednym filtrze na każdy neuron warstwy gęstej i wprowadzić uzupełnianie zerami typu "valid". Zazwyczaj krok powinien być równy 1, ale jeśli chcesz, możesz wyznaczyć jego większą wartość. Funkcję aktywacji pozostawiamy taką samą jak w warstwie gęstej. Pozostałe warstwy gęste powinny być przekształcone w taki sam sposób, ale wstawiamy w nich filtry 1×1 . W rzeczywistości istnieje możliwość przekształcenia w ten sposób wytrenowanej sieci CNN poprzez odpowiednie przekształcenie macierzy wag warstw gęstych.
8. Głównym problemem technicznym w segmentacji semantycznej jest fakt, że w sieci splotowej mnóstwo informacji przestrzennych zostaje utraconych podczas przebiegu sygnału przez poszczególne warstwy, zwłaszcza łączące i zawierające krok większy niż 1. Należy jakoś odtworzyć te informacje przestrzenne, aby dokładnie przewidywać klasę każdego piksela.

Rozwiązania ćwiczeń od 9. do 12. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 15. Przetwarzanie sekwencji za pomocą sieci rekurencyjnych i splotowych

1. Kilka zastosowań sieci RNN:

- Sekwencyjne sieci rekurencyjne: prognozowanie pogody (lub dowolnego innego szeregu czasowego), tłumaczenie maszynowe (za pomocą architektury koder – dekoder), podpisывanie filmów, przetwarzanie mowy na tekst, tworzenie muzyki (lub innych sekwencji), rozpoznawanie akordów.
 - Sieci sekwencyjno-wektorowe: przypisywanie fragmentów muzyki do określonego gatunku, analizowanie sentymentów w recenzjach książek, przewidywanie słów, o których myśli afatyczny pacjent, na podstawie odczytów z wszczepów mózgowych, wyliczanie prawdopodobieństwa, że użytkownik będzie chciał oglądać proponowany film, na podstawie historii obejrzanych pozycji (jest to jedna z wielu możliwych implementacji **filtracji wspólnej** — ang. *collaborative filtering*).
 - Sieci wektorowo-sekwencyjne: podpisывanie rysunków, tworzenie listy utworów na podstawie wybranego wykonawcy, generowanie melodii na podstawie określonych parametrów, wyszukiwanie pieszych na zdjęciach (np. na zdjęciu wykonanym przez samochód inteligentny).
2. Warstwa rekurencyjna musi mieć trójwymiarowe wejścia: pierwszy wymiar to rozmiar grupy, drugi wymiar reprezentuje czas (liczbę taktów), natomiast w trzecim wymiarze są przechowywane dane wejściowe w każdym taktie (liczba cech wejściowych w każdym taktie). Na przykład jeżeli chcesz przetwarzać grupę zawierającą pięć szeregów czasowych po 10 taktów każdy, a w każdym takterze dwie wartości (np. temperaturę powietrza i prędkość wiatru), wejście tej sieci będzie miało wymiary [5, 10, 2]. Wyjścia są również trójwymiarowe, gdzie pierwsze dwa wymiary są takie same, ale ostatni jest równy liczbie neuronów. Na przykład jeśli warstwa rekurencyjna składająca się z 32 neuronów przetworzy omówioną grupę danych, to dane wyjściowe będą miały wymiary [5, 10, 32].
3. Aby stworzyć głęboką, sekwencyjną sieć rekurencyjną za pomocą interfejsu Keras, musisz wyznać argument `return_sequences=True` we wszystkich warstwach rekurencyjnych. Jeśli zaś sieć ma być sekwencyjno-wektorowa, umieść `return_sequences=True` we wszystkich warstwach rekurencyjnych oprócz szczytowej, w której argument ten powinien mieć wartość `False` (albo możesz go zupełnie pominąć, gdyż wartość `False` jest domyślna).
4. Jeżeli masz dostęp do codziennego jednowymiarowego szeregu czasowego i chcesz uzyskać prognozy na następnych 7 dni, najprościej utworzyć strukturę RNN składającą się ze stosu warstw rekurencyjnych (mających wyznaczony argument `return_sequences=True` oprócz ostatniej warstwy rekurencyjnej) i ostatniej, siedmioneuronowej warstwy rekurencyjnej. Możesz wytrenować ten model za pomocą losowych ramek z szeregu czasowego (np. sekwencji 30 kolejnych dni i zmiennej docelowej w postaci wektora przechowującego wartości dla 7 następnych dni). Jest to przykład sieci sekwencyjno-wektorowej. Ewentualnie możesz wyznać argument `return_sequences=True` we wszystkich warstwach rekurencyjnych i uzyskać w ten sposób sieć sekwencyjną. Tego typu model możesz uczyć za pomocą losowych ramek z szeregu czasowego, w których sekwencje danych wejściowych mają taką samą długość jak zmienne docelowe.

Każda sekwencja docelowa powinna zawierać po siedem wartości na każdy takt (np. w takcie t sekwencją docelową powinien być wektor zawierający wartości z taków od $t+1$ do $t+7$).

5. Podstawowe trudności podczas uczenia sieci rekurencyjnych to niestabilne gradienty (eksplodujące lub zanikające) oraz bardzo ograniczona pamięć krótkotrwała. Im dłuższe sekwencje, tym problemy te stają się coraz wyraźniejsze. Aby złagodzić kwestię niestabilnych gradientów, możesz zmniejszyć wartość współczynnika uczenia, wprowadzić nasycającą funkcję aktywacji, taką jak funkcję tangensa hiperbolicznego (jest ona domyślna), a także ewentualnie wykorzystać obcinanie gradientów, normalizację wsadową lub porzucanie w każdym taktie. Z kolei problem ograniczonej pamięci krótkotrwałej rozwiązuje warstwy LSTM lub GRU (zmniejszają one również ryzyko niestabilnych gradientów).
6. Struktura komórki LSTM wydaje się skomplikowana, ale w rzeczywistości jest dość prosta, jeżeli rozumiemy zachodzące w niej procesy. Komórka ta zawiera wektor stanu krótkotrwałego i wektor stanu długotrwałego. W każdym taktie dane wejściowe i poprzedni stan krótkotrwały są przekazywane do prostej komórki rekurencyjnej i trzech bramek: bramka zapominająca wyznacza elementy, które mają zostać usunięte ze stanu długotrwałego, bramka wejściowa określa, które informacje z wyjścia prostej warstwy rekurencyjnej mają zostać umieszczone w stanie długotrwałym, natomiast bramka wyjściowa wybiera elementy stanu długotrwałego, które mają zostać przesłane do następnego taktu (po przejściu przez funkcję aktywacji tanh). Nowy stan krótkotrwały jest równoznaczny z danymi wyjściowymi komórki. Zobacz rysunek 15.9.
7. Warstwa rekurencyjna jest zasadniczo sekwencyjna: aby obliczyć wyniki w taktie t , muszą zostać najpierw obliczone rezultaty we wszystkich wcześniejszych taktach. Z tego powodu zrównoleglenie modelu okazuje się niemożliwe. Z drugiej strony jednowymiarowa warstwa splotowa radzi sobie dobrze ze zrównolegleniem, ponieważ nie przechowuje stanów pomiędzy poszczególnymi takami. Innymi słowy, nie zawiera ona pamięci: wynik w każdym taktie może zostać obliczony na podstawie małego przedziału wartości z danych wejściowych bez potrzeby znajomości wcześniejszych wartości. Ponadto jednowymiarowa warstwa splotowa nie jest rekurencyjna, dlatego mniej jej dotyczy problem niestabilnych gradientów. W sieci rekurencyjnej może przydać się co najmniej jedna jednowymiarowa warstwa splotowa, gdyż umożliwia skuteczne przetwarzanie wstępne danych wejściowych, na przykład w celu zmniejszenia ich rozdrobnienia przestrzennej, co pozwala warstwom rekurencyjnym wykrywać wzorce długoterminowe. W istocie możliwe jest wykorzystywanie wyłącznie warstw splotowych, co udowodniono za pomocą architektury WaveNet.
8. Jedna z możliwych architektur klasyfikowania filmów na podstawie zawartości wizualnej mogłaby wyglądać następująco: model przyjmowałby (założymy) jedną klatkę na sekundę, która byłaby następnie przetwarzana przez sieć splotową (może to być np. gotowy model Xception, być może zamrożony, przy założeniu, że Twój zestaw danych nie jest zbyt duży), uzyskany wynik byłby dostarczany do sekwencyjno-wektorowej sieci rekurencyjnej, a ostateczny rezultat byłby uzyskiwany po przejściu przez warstwę zawierającą funkcję aktywacji softmax, dzięki czemu otrzymywaliśmy wszystkie prawdopodobieństwa przynależności do klas. W czasie treningu funkcją kosztu mogłyby być entropia krzyżowa. Gdybyśmy chcieli w celu klasyfikacji wykorzystać również dźwięk, moglibyśmy użyć stosu jednowymiarowych, krokowych warstw splotowych, które zmniejszałyby rozdrobniość czasową z tysiący ramek na sekundę do zaledwie jednej ramki

na sekundę (aby liczba ta zgadzała się z liczbą obrazów na sekundę), po czym połączyć sekwencję wyjściową z sygnałem wejściowym sekwencyjno-wektorowej sieci rekurencyjnej (wzdłuż ostatniego wymiaru).

Rozwiązań ćwiczeń 9. i 10. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 16. Przetwarzanie języka naturalnego za pomocą sieci rekurencyjnych i mechanizmów uwagi

1. Bezstanowa sieć rekurencyjna może wykrywać jedynie wzorce, których długość jest co najwyżej równa rozmiarowi ramek, za pomocą których sieć taka jest uczona. Z kolei stanowa sieć rekurencyjna jest w stanie wykrywać bardziej długoterminowe wzorce. Jednakże implementacja stanowej sieci rekurencyjnej jest znacznie trudniejsza (zwłaszcza faza przygotowywania zestawu danych). Ponadto stanowe sieci rekurencyjne nie zawsze będą działać lepiej, częściowo dlatego, że kolejne grupy danych nie są niezależne i identycznie rozmieszczone. Tego typu zestawy danych nie są zbyt chętnie przetwarzane przez algorytm gradientu prostego.
2. Zasadniczo, jeśli będziemy tłumaczyć pojedynczo wyrazy, wyniki będą okropne. Na przykład francuski zwrot *je vous en prie* oznacza „proszę bardzo”, jeśli jednak przetłumaczymy go słowo po słowie, uzyskamy mniej więcej następujący rezultat: „ja wy w proszę” albo nawet „ja wy w modlę się”, bo czasownik *prier* ma dwa znaczenia. Znacznie lepiej przeczytać najpierw cały zwrot i dopiero wtedy go przetłumaczyć. Standardowa sekwencyjna sieć RNN zaczęłaby tłumaczyć ten zwrot natychmiast po przeczytaniu pierwszego wyrazu, natomiast architektura koder – dekoder najpierw przeczyta cały zwrot, a następnie zajmie się jego przekładem. Podsumowując: możemy sobie łatwo wyobrazić standardową sekwencyjną sieć RSN, która „milczałaby” za każdym razem, gdy nie byłaby pewna, co powiedzieć (podobnie jest w przypadku tłumaczy multilingualnych, gdy muszą dokonywać przekładu w locie).
3. W przypadku sekwencji wejściowych o zmiennej długości możemy krótsze sekwencje uzupełniać zerami w taki sposób, aby wszystkie sekwencje w grupie miały jednakową długość, a następnie wprowadzać maskowanie, dzięki któremu sieć rekurencyjna będzie ignorowała znacznik uzupełniania zerami. Aby uzyskać lepszą wydajność, możesz również tworzyć grupy o sekwencjach mających podobne rozmiary. Tensory nierówne mogą przechowywać sekwencje o zmiennej długości, a interfejs `tf.keras` będzie je najprawdopodobniej obsługiwał w przyszłości, co znacznie uprości proces przetwarzania sekwencji wejściowych o zmiennej długości (w czasie gdy pisałem niniejszą książkę, funkcja ta jeszcze nie była dostępna). W przypadku sekwencji wyjściowych o zmiennej długości, jeżeli znamy z góry długość sekwencji wyjściowej (tzn. wiemy, że będzie ona miała taką samą długość jak sekwencja wejściowa), wystarczy tak skonfigurować funkcję straty, aby ignorowała tokeny znajdujące się za sekwencją. W podobny sposób kod wykorzystujący model powinien ignorować znaczniki znajdujące się po sekwencji. Zasadniczo jednak nie znamy z góry długości sekwencji wyjściowej, dlatego rozwiązaniem okazuje się uczenie modelu w taki sposób, aby na końcu każdej sekwencji umieszczał odpowiedni znacznik końcowy.

- 4.** Przeszukiwanie wiązkowe to technika służąca do poprawienia wydajności wyuczonego modelu typu koder – dekoder, na przykład w systemie neuronowego uczenia maszynowego. Algorytm śledzi krótką listę k najbardziej obiecujących zdań wyjściowych (np. trzech), a w każdym kroku dekoder stara się dodawać do tych zdań po jednym słowie; następnie zachowuje jedynie k najbardziej prawdopodobnych zdań. Parametr k jest nazywany **szerokością wiązki**, a im większa jego wartość, tym będzie wykorzystywana większa moc obliczeniowa i pamięć operacyjna, ale system będzie dokładniejszy. W tej technice system nie musi zachłannie wybierać najbardziej prawdopodobnego następnego słowa w każdym kroku po to, aby rozwinąć zdanie, lecz rozważanych jest jednocześnie kilka najbardziej obiecujących zdań. Ponadto technika ta nadaje się do zrównoleglania. Możesz całkiem łatwo zaimplementować algorytm przeszukiwania wiązkowego za pomocą projektu TensorFlow Addons.
- 5.** Mechanizm uwagi to technika pierwotnie wykorzystywana w modelach typu koder – dekoder, zapewniająca dekoderowi bardziej bezpośredni dostęp do sekwencji wejściowej, a co za tym idzie, do przetwarzania dłuższych sekwencji. W każdym taktie dekodera bieżący stan dekodera i pełny wynik kodera są przetwarzane przez model dopasowania, generujący wynik dopasowania dla każdego taktu wejściowego. Wynik ten wskazuje najistotniejszy fragment sygnału wejściowego dla bieżącego taktu dekodera. Suma ważona sygnału wyjściowego kodera (ważona za pomocą wyniku dopasowania) zostaje następnie przekazana do dekodera, który generuje kolejny stan dekodera, a także rezultat dla tego taktu. Główną zaletą stosowania mechanizmu uwagi jest fakt, że model typu koder – dekoder może w ten sposób przetwarzać dłuższe sekwencje wejściowe. Inną zaletą jest ułatwienie usuwania błędów i interpretowania modelu dzięki wynikom dopasowania: na przykład jeżeli model popełni błąd, możesz sprawdzić, na którą część sygnału wejściowego zwracał uwagę, co może pomóc w rozwiązyaniu problemu. Mechanizm uwagi stanowi również serce architektury transformatora w wieloblokowych warstwach mechanizmu uwagi. Zobacz następną odpowiedź.
- 6.** Najważniejszą warstwą w strukturze transformatora jest wieloblokowa warstwa mechanizmu uwagi (w pierwotnym modelu transformatora znajduje się 18 takich warstw, w tym sześć warstw maskowanych). Stanowi ona główny element w takich modelach językowych jak BERT czy GPT-2. Jej zadaniem jest określenie najściślej powiązanych ze sobą słów, a następnie udoskonalenie reprezentacji każdego wyrazu za pomocą tych kontekstowych poszlakek.
- 7.** Z próbkowanej funkcji softmax korzystamy podczas uczenia modelu klasyfikacji w przypadku występowania wielu klas (np. tysięcy). Obliczana jest aproksymacja entropii krzyżowej na podstawie logitu przewidywanego przez model dla właściwej klasy, a także prognozowanych logitów dla próby niewłaściwych wyrazów. Mechanizm ten znacznie przyspiesza proces uczenia w porównaniu do obliczania funkcji softmax dla wszystkich logitów, a następnie obliczania entropii krzyżowej. Po zakończeniu uczenia model może być normalnie używany, gdzie standar-dowa funkcja softmax służy do obliczania wszystkich przynależności do klas na podstawie logitów.

Rozwiązania ćwiczeń od 8. do 11. znajdziesz w notatniku Jupyter dostępnym na stronie
<ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 17. Uczenie reprezentacji za pomocą autokoderów i generatywnych sieci przeciwnych

1. Oto kilka podstawowych zastosowań autokoderów:

- wydobywanie cech,
- nienadzorowane uczenie wstępne,
- redukowanie wymiarowości,
- modele generatywne,
- wykrywanie anomalii (autokoder najczęściej nie nadaje się do rekonstruowania elementów odstających).

2. Jeśli chcesz stworzyć klasyfikator i masz wiele nieoznaczonych danych i tylko kilka tysięcy zawierających etykiety, to możesz najpierw wyuczyć głęboki autokoder za pomocą pełnego zestawu danych (oznaczonych + nieoznaczonych), a następnie wykorzystać dolną połowę klasyfikatora (tzn. warstwy do warstwy kodującej włącznie), po czym wytrenować klasyfikator za pomocą oznaczonych danych. W przypadku niewielkiej liczby oznaczonych danych prawdopodobnie najlepiej zamrozić ponownie wykorzystywane warstwy podczas uczenia klasyfikatora.

3. To, że autokoder doskonale rekonstruuje dane wejściowe, wcale nie musi oznaczać, iż jest on dobrym modelem — prawdopodobnie mamy po prostu do czynienia z autokoderem zupełionym, który nauczył się kopiować dane wejściowe do warstwy kodowania, a stamtąd na wyjście. W rzeczywistości nawet gdyby warstwa kodowania składała się tylko z jednego neuronu, możliwe byłoby wyuczenie bardzo głębokiego autokodera, tak żeby każda próbka ucząca była rzutowana do innego kodowania (np. pierwsza próbka byłaby rzutowana do kodowania 0,001, druga do 0,002, trzecia do 0,003 itd.), i opanowałby on sztukę rekonstruowania „na pamięć” prawidłowej próbki uczącej z każdego kodowania. Taki autokoder potrafiłby perfekcyjnie odwzorowywać dane wejściowe bez wydobycia z tych próbek prawdziwie przydatnych wzorców. W praktyce tego typu rzutowanie ma bardzo małe szanse na wystąpienie, ale stanowi ono dobre wyjaśnienie, dlaczego doskonała rekonstrukcja nie musi koniecznie oznaczać, że model nauczył się czegoś pożytecznego. Z drugiej strony, fatalne rekonstrukcje niemal zawsze oznaczają kiepski autokoder. W celu określenia wydajności autokodera jednym ze sposobów jest zmierzenie funkcji straty rekonstrukcji (np. wyliczenie błędu MSE, średniego kwadratu różnicy danych wyjściowych i wejściowych). Duża wartość straty rekonstrukcji wskazuje wyraźnie na niską jakość autokodera, ale również mała jej wartość wcale nie musi oznaczać wydajnego autokodera. Należy także ocenić wydajność autokodera pod względem czekającego go typu zadania. Przykładowo jeśli używamy go do nienadzorowanego uczenia wstępnego klasyfikatora, to powinniśmy również zmierzyć wydajność tego klasyfikatora.

4. Autokoder niedopełniony zawiera warstwę kodowania mniejszą od warstw wejściowej i wyjściowej. W odwrotnej sytuacji mamy do czynienia z autokoderem zupełionym. Głównym zagrożeniem w nadmiernie niedopełnionym autokoderze jest problem z rekonstrukcją danych wejściowych. Z kolei w autokoderze zupełionym dane wejściowe mogą być zwyczajnie kopiowane na wyjście bez jakiegokolwiek nauki rozpoznawania wzorców i użytecznych cech przez model.

5. Aby powiązać wagi warstwy kodera z odpowiadającą jej warstwą dekodera, wystarczy sprawić, aby wagi dekodera były równe transponowanym wagom kodera. W ten sposób zmniejszamy o połowę liczbę parametrów modelu, dzięki czemu model staje się znacznie szybciej zbieżny przy użyciu mniejszej liczby próbek uczących, a także ograniczamy ryzyko przetrenowania modelu wobec zbioru danych uczących.
6. Model generatywny jest w stanie losowo generować wyniki przypominające próbki uczące. Na przykład po udanym wyuczeniu modelu za pomocą zbioru danych MNIST model generatywny może całkiem skutecznie losowo generować realistyczne odwzorowania odręcznie pisanych cyfr. Rozkład wyników zazwyczaj bardzo przypomina rozkład danych wejściowych. Przykładowy zestaw danych MNIST zawiera wiele obrazów każdej cyfry, zatem model generatywny będzie w stanie tworzyć mniej więcej taką samą liczbę obrazów każdej klasy. Niektóre modele generatywne mogą być parametryczne — na przykład dostosowane do generowania jedynie określonych rodzajów wyników. Przykładem autokodera generatywnego jest autokoder wariacyjny.
7. Generatywna sieć przeciwna stanowi strukturę sieci neuronowej składającą się z dwóch części: generatora i dyskryminatora, mających sprzeczne cele. Zadaniem generatora jest tworzenie przykładów przypominających zestaw danych uczących po to, aby oszukać dyskryminator. Z kolei dyskryminator musi odróżniać rzeczywiste przykłady od wygenerowanych imitacji. W każdej iteracji uczenia dyskryminator jest uczyony jak normalny klasyfikator binarny, a generator uczy się maksymalizować wartość błędu dyskryminatora. Sieci GAN używane są do zadań zaawansowanego przetwarzania obrazów, takich jak nadrozdzielcość, koloryzowanie, edycja obrazów (zastępowanie obiektów realistycznym tłem), przekształcanie prostych szkiców w fotograficzne obrazy czy przewidywanie kolejnych klatek w obrazach wideo. Za ich pomocą można również dogenerować dane (służące do uczenia innych modeli), generować inne typy danych (np. tekstowe, dźwiękowe czy szeregi czasowe), a także wykrywać słabe strony innych modeli oraz usprawniać te modele.
8. Uczenie sieci GAN bezustannie sprawia problem z powodu skomplikowanych zależności dynamicznych zachodzących pomiędzy generatorem a dyskryminatorem. Największą trudność stanowi załamanie modułu, wskutek którego generator tworzy imitacje o bardzo małym zróżnicowaniu. Ponadto proces uczenia może być wyjątkowo niestabilny: może rozpocząć się we właściwy sposób, po czym niespodziewanie mogą się pojawić oscylacje lub rozbieżności bez wyraźnego powodu. Generatywne sieci przeciwnie są również bardzo wrażliwe na dobór wartości hiperparametrów.

Rozwiązania ćwiczeń 9., 10. i 11. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 18. Uczenie przez wzmacnianie

1. Uczenie przez wzmacnianie stanowi dziedzinę uczenia maszynowego koncentrującą się na tworzeniu agentów zdolnych do wykonywania czynności w danym środowisku w sposób maksymalizujący uzyskiwanie nagród wraz z upływem czasu. Istnieje wiele różnic między uczeniem przez wzmacnianie a standardowymi technikami nadzorowanymi/nienadzorowanymi. Oto kilka z nich:

- W uczeniu nienadzorowanym i nadzorowanym zazwyczaj celem jest wyszukiwanie wzorców w danych i uzyskiwanie prognoz za ich pomocą. Z kolei uczenie przez wzmacnianie dąży do znalezienia dobrej polityki.
- W przeciwnieństwie do uczenia nadzorowanego agent nie otrzymuje jawnie „właściwej” odpowiedzi. Musi sam ją odkryć metodą prób i błędów.
- W przeciwnieństwie do uczenia nienadzorowanego istnieje tu pewna forma nadzorowania — za pomocą nagród. Nie mówimy agentowi, jak ma wykonać dane zadanie, ale dajemy mu znać, gdy czyni postępy lub ponosi porażki.
- Agent uczenia przez wzmacnianie musi znaleźć dobrą równowagę pomiędzy przeszukiwaniem środowiska, szukaniem nowych sposobów uzyskiwania nagród a wykorzystywaniem znanych już źródeł nagród. Z kolei systemy nadzorowane i nienadzorowane nie martwią się kwestią przeszukiwania środowiska — wystarczy im otrzymywanie danych uczących.
- W uczeniu nienadzorowanym i nadzorowanym przykłady uczące są przeważnie wzajemnie niezależne (a najczęściej są nawet tasowane). Z kolei w przypadku uczenia przez wzmacnianie następujące po sobie obserwacje zazwyczaj **nie** są niezależne. Agent może pozostawać w określonym obszarze środowiska przez pewien czas, zanim przejdzie dalej, dlatego następujące po sobie obserwacje są ze sobą ściśle skorelowane. W pewnych sytuacjach wprowadza się pamięć odtwarzania po to, aby zapewnić algorytmowi uczącemu odpowiednio niezależne obserwacje.

2. Oto kilka zastosowań uczenia przez wzmacnianie oprócz wymienionych w rozdziale 18.:

Personalizacja muzyki

Środowiskiem jest spersonalizowane radio internetowe użytkownika. Agentem jest oprogramowanie decydujące, jaki utwór ma zostać odtworzony jako następny. Możliwymi czynnościami są: odtworzenie dowolnego utworu znajdującego się w katalogu (agent musi wybrać ulubioną piosenkę użytkownika) lub odtworzenie reklamy (agent musi wybrać reklamę, która będzie w stanie zainteresować użytkownika). Agent uzyskuje niewielką nagrodę za każdym razem, gdy użytkownik przesłucha utwór, większą nagrodę za wysłuchanie reklamy przez użytkownika, nagrodę ujemną za pominięcie utworu lub reklamy, a bardzo dużą nagrodę ujemną za wyjście z aplikacji.

Marketing

Środowiskiem jest dział marketingu w danej firmie. Agentem jest oprogramowanie określające, co należy wysyłać klientom w trakcie kampanii reklamowej, biorąc pod uwagę ich profil i historię zakupów (dla każdego klienta istnieją dwie czynności: wysłać albo nie wysłać). Agent otrzymuje nagrodę ujemną za koszt kampanii, a nagrodę dodatnią za szacowane przychody uzyskane dzięki niej.

Dostarczanie produktu

Niech agent kieruje flotą pojazdów dostawczych i określa, co mają odbierać z magazynów, gdzie mają się udać, co mają dostarczyć do odbiorców itd. Agent otrzymywały nagrodę dodatnią za każdy produkt dostarczony w terminie, a negatywną za wszelkie opóźnienia.

3. Podczas szacowania wartości danej czynności algorytmy uczenia przez wzmacnianie zazwyczaj sumują wszystkie nagrody, do których doprowadziła dana czynność, przy czym natychmiastowe nagrody mają większą wagę od odległszych w przyszłości (bierzemy pod uwagę fakt, że dana czynność ma większy wpływ na bliższą niż dalszą przyszłość). Aby przekuć tę teorię na praktykę, wprowadza się stopę dyskontową w każdym kroku. Na przykład przy stopie dyskontowej o wartości 0,9 nagroda o wartości 100 otrzymana po dwóch krokach maleje do zaledwie $0,9^2 \times 100 = 81$ podczas szacowania wartości czynności. Możemy interpretować stopę dyskontową jako miarę znaczenia przyszłości w stosunku do czasu teraźniejszego: im bliżej jedności, tym przyszłość ma znaczenie bardziej zbliżone do teraźniejszości. Z kolei im bliżej zera, tym ważniejsze stają się bieżące nagrody. Wpływa to oczywiście bardzo istotnie na rodzaj polityki: jeśli troszczymy się o przyszłość, możemy wytrzymać wiele w oczekiwaniu na dalsze nagrody, a jeśli nie interesuje nas przyszłość, wykorzystujemy każdą nadarzającą się okazję i nie inwestujemy w to, co przyjdzie później.
4. Aby zmierzyć wydajność agenta uczenia przez wzmacnianie, wystarczy zsumować otrzymywane przez niego nagrody. W symulowanym środowisku możesz uruchomić wiele epizodów i sprawdzić uśredzoną wartość wszystkich uzyskiwanych nagród (ewentualnie także obliczyć wartość minimalną, maksymalną, odchylenie standardowe itd.).
5. Problem przydzielania zasługi występuje wtedy, gdy agent uczenia przez wzmacnianie, otrzymując nagrodę, nie ma sposobu sprawdzenia, która z poprzednich czynności przyczyniła się do jej uzyskania. Zazwyczaj pojawia się on wtedy, gdy istnieje duże opóźnienie pomiędzy czynnością a wynikającymi z niej nagrodami (np. w trakcie gry w pong może występować kilkakrotnie kroków pomiędzy odbiciem piłki przez agenta a zdobyciem punktu). Jednym ze sposobów zmniejszenia skutków tego problemu jest przyznawanie agentowi nagród krótkoterminowych, jeśli istnieje taka możliwość. Rozwiążanie to najczęściej wymaga dostarczenia agentowi uprzedniej wiedzy na temat czekającego go zadania. Przykładowo jeśli chcesz nauczyć agenta grać w szachy, należałoby mu przyznawać nagrodę nie tylko za każde zwycięstwo, ale także za zbitie figury przeciwnika.
6. Często agent może pozostawać przez pewien czas w określonym obszarze środowiska, zatem w tym okresie wszystkie doświadczenia będą bardzo do siebie podobne. Zjawisko to może spowodować wzrost obciążenia w algorytmie uczenia. Polityka może zostać dostrojona do tego obszaru środowiska, ale zacznie osiągać dużo gorsze wyniki po dotarciu do innego obszaru. Możemy rozwiązać ten problem za pomocą pamięci odtwarzania: zamiast wykorzystywać wyłącznie bieżące doświadczenia w procesie uczenia, agent będzie trenował przy użyciu bufora składającego się z wcześniejszych doświadczeń, nowych i starszych (być może właśnie dlatego śnimy — aby odtwarzać wspomnienia minionego dnia i lepiej się z nich uczyć).
7. Pozapolicyzny algorytm uczenia przez wzmacnianie uczy się wartości optymalnej polityki (tzn. sumy zmniejszonych nagród, które mogą być spodziewane dla każdego stanu, jeśli agent zachowuje się optymalnie), niezależnie od tego, jak agent zachowuje się w rzeczywistości. Dobry przykład stanowi algorytm Q-uczenia. Natomiast algorytm polityczny (ang. *on-policy algorithm*) uczy się polityki, którą agent rzeczywiście stosuje, w tym również przeszukiwania i wykorzystywania środowiska.

Rozwiązania ćwiczeń 8., 9. i 10. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Rozdział 19. Wielkoskalowe uczenie i wdrażanie modeli TensorFlow

1. Obiekt SavedModel zawiera model TensorFlow: jego strukturę (graf obliczeniowy) i wagę. Jest przechowywany w postaci katalogu zawierającego plik `saved_model.pb` definiujący graf obliczeniowy (reprezentowany przez serializowany bufor protokołów) oraz podkatalogu `variables`, w którym znajdziemy wartości zmiennych. W przypadku modeli składających się z dużej liczby wag wartości tych zmiennych mogą być podzielone pomiędzy wiele plików. Dostępny jest także podkatalog `assets`, zawierający dodatkowe dane, takie jak pliki słowników, nazwy klas czy jakieś przykładowe dane uczące. Mówiąc dokładniej, obiekt SavedModel może zawierać co najmniej jeden **metagraf**. Jest to połączenie grafu obliczeniowego z pewnymi definicjami sygnatur funkcji (włącznie z nazwami, typami oraz wymiarami danych wejściowych i wyjściowych). Każdy metagraf jest określany za pomocą zestawu znaczników. Aby przejrzeć zawartość obiektu SavedModel, możesz skorzystać z narzędzia wiersza polecenia `saved_model_cli` lub wczytać go za pomocą funkcji `tf.saved_model.load()` i przejrzeć w środowisku Python.
2. Serwer TF Serving umożliwia łatwe wdrażanie wielu modeli TensorFlow (lub wielu wersji tego samego modelu) do wszystkich Twoich aplikacji poprzez interfejs gRPC lub REST. Bezpośrednie stosowanie modelu w aplikacjach znacznie utrudniłoby wdrażanie nowych wersji tego modelu. Implementowanie mikrouruchu otaczającej model TF wymagałoby dodatkowej pracy i niełatwo byłoby dopasować ją do funkcji TF. Serwer TF Serving cechuje się wieloma własnościami: może monitorować katalog i automatycznie wdrażać umieszczone w nim modele, a Ty nie będziesz musiał/musiał zmieniać ani nawet ponownie uruchamiać żadnej ze swoich aplikacji, aby móc stosować tę nową wersję; jest szybki, wszechstronnie przetestowany i w znacznym stopniu skalowalny; obsługuje też testowanie A/B modeli eksperymentalnych i umożliwia wdrażanie nowej wersji modelu dla wyznaczonej podgrupy użytkowników (testowanie kanarkowe). Serwer TF Serving może również grupować poszczególne kwerendy i przetwarzanie je łącznie w procesorze graficznym. Aby wdrożyć serwer TF Serving, możesz zainstalować go bezpośrednio ze źródła, ale o wiele łatwiej dokonać tego za pomocą obrazu Docker'a. W celu wdrożenia klastra obrazów TF Serving Docker możesz użyć platformy Kubernetes albo kompleksowego rozwiązania, takiego jak na przykład Google Cloud AI Platform.
3. Do wdrożenia modelu za pomocą wielu instancji TF Serving wystarczy takie skonfigurowanie tych instancji, aby monitorowały ten sam katalog `models`, a następnie wyeksportowanie nowego modelu jako obiektu SavedModel do podkatalogu.
4. Interfejs gRPC jest wydajniejszy od interfejsu REST. Jednak jego biblioteki klienckie nie są rozpowszechnione, a jeżeli podczas korzystania z interfejsu REST włączysz kompresję, osiągniesz niemal taką samą wydajność. Z tego powodu interfejs gRPC jest najbardziej przydatny wtedy, gdy wymagasz maksymalnej wydajności, a klienci nie są ograniczeni do interfejsu REST.

- 5.** Biblioteka TFLite wykorzystuje kilka technik do zmniejszania rozmiaru modelu w przypadku jego wdrażania do urządzenia mobilnego lub wbudowanego:
- Zawiera konwerter optymalizujący obiekt SavedModel: zmniejsza rozmiar modelu i redukuje jego opóźnienie. W tym celu są usuwane wszystkie operacje niepotrzebne do uzyskiwania prognoz (np. operacje uczenia) oraz optymalizowane i łączone wszelkie operacje tam, gdzie to możliwe.
 - Konwerter może również przeprowadzać kwantyzację potreningu: technika ta znacznie zmniejsza rozmiar modelu, zatem można go o wiele łatwiej pobierać i przechowywać.
 - Zapisuje zoptymalizowany model w formacie FlatBuffer, który można bezpośrednio wczytywać do pamięci operacyjnej, bez uprzedniej analizy składniowej kodu. Zmniejszamy w ten sposób czas wczytywania i wymaganą pojemność pamięci.
- 6.** Uczenie w kontekście kwantyzacji polega na dodawaniu udawanych operacji kwantyzacji do modelu podczas treningu. W ten sposób model uczy się ignorować szum kwantyzacji — końcowe wagи będą na nią bardziej uodpornione.
- 7.** Zrównoleglanie modelu to proces dzielenia modelu na wiele części i równoległego ich przetwarzania na różnych urządzeniach, co ma na celu przyspieszyć uczenie lub wnioskowanie. W zrównoleglaniu danych tworzone są dokładne repliki modelu, które zostają wdrożone do wielu urządzeń. W każdej iteracji uczenia poszczególne repliki otrzymują różne grupy danych, a następnie obliczają gradient funkcji straty w odniesieniu do parametrów modelu. W synchronicznym zrównoleglaniu danych gradienty ze wszystkich replik są gromadzone, a optymalizator wykorzystuje metodę gradientu prostego. Parametry mogą być skoncentrowane (np. umieszczone w serwerach parametrów) lub przesyłane do wszystkich replik i synchronizowane za pomocą algorytmu AllReduce. W asynchronicznym zrównoleglaniu danych parametry są skoncentrowane, a repliki działają niezależnie od siebie: każda z nich aktualizuje bezpośrednio parametry centralne na końcu każdej iteracji uczenia, bez konieczności czekania na pozostałe repliki. Okazuje się, że w kontekście przyspieszenia procesu uczenia zrównoleglanie danych spisuje się lepiej od zrównoleglania modelu. Wynika to przede wszystkim z mniejszej komunikacji pomiędzy urządzeniami. Poza tym rozwiązanie to jest o wiele łatwiejsze do zaimplementowania i działa w taki sam sposób dla każdego modelu, natomiast w zrównoleglaniu modelu konieczne jest przeanalizowanie modelu pod kątem określenia optymalnego sposobu jego podziału.
- 8.** Podczas uczenia modelu na wielu serwerach możesz skorzystać z następujących strategii rozpraszania:
- Klasa MultiWorkerMirroredStrategy przeprowadza zrównoleglanie danych zgodne ze strategią duplikowania. Model jest duplikowany na wszystkie dostępne serwery i urządzenia, a każda replika otrzymuje inną grupę danych w każdej iteracji uczenia, po czym oblicza własne gradienty. Zostaje obliczona średnia ze wszystkich gradientów i udostępniona wszystkim replikom za pomocą rozproszonej implementacji algorytmu AllReduce (domyślnie NCCL), następnie wszystkie repliki przeprowadzają ten sam etap gradientu prostego. Strategia ta jest najprostsza w użyciu, ponieważ wszystkie serwery i urządzenia są traktowane w taki sam sposób i jej wydajność jest całkiem niezła. Zasadniczo należy korzystać właśnie z tej strategii. Jej głównym ograniczeniem jest konieczność zmieszczenia się całego modelu w pamięci operacyjnej każdej repliki.

- Klasa ParameterServerStrategy realizuje strategię asynchronicznego zrównoleglania danych. Model jest przesyłany na wszystkie urządzenia we wszystkich roboczych grupach zadań, a parametry zostają podzielone pomiędzy wszystkie serwery parametrów. Każda robocza grupa zadań ma przydzieloną własną pętlę uczenia, która działa asynchronicznie w stosunku do pozostałych roboczych grup zadań; w każdej iteracji uczenia poszczególne robocze grupy zadań otrzymują własną grupę danych i pobierają najnowszą wersję parametrów modelu z serwerów parametrów, a następnie obliczają gradienty funkcji straty w odniesieniu do tych parametrów, które to gradienty zostają odesłane do serwerów parametrów. Na koniec serwery parametrów przeprowadzają etap gradientu prostego za pomocą tych gradientów. Strategia ta jest zazwyczaj nieco powolniejsza i trudniejsza do zaimplementowania od omówionej w poprzednim punkcie, ponieważ trzeba dodatkowo zarządzać serwerami parametrów. Jednak przydaje się ona do trenowania olbrzymich modeli, które nie mieścią się w pamięci operacyjnej kart graficznych.

Rozwiązania ćwiczeń 9., 10. i 11. znajdziesz w notatniku Jupyter dostępnym na stronie <ftp://ftp.helion.pl/przyklady/uczem2.zip>.

Lista kontrolna projektu uczenia maszynowego

Poniższa lista stanowi pomoc podczas tworzenia projektów uczenia maszynowego. Na proces ten składa się osiem etapów:

- 1.** Określenie problemu i przeanalizowanie go w szerszej perspektywie.
- 2.** Pozyskanie danych.
- 3.** Analiza danych w celu wykrycia dodatkowych informacji.
- 4.** Przygotowanie danych w sposób uwidaczniający wzorce wykorzystywane przez algorytmy uczenia maszynowego.
- 5.** Sprawdzenie wielu modeli i stworzenie krótkiej listy najwydajniejszych z nich.
- 6.** Dostrojenie modeli i połączenie ich w zespoły uzyskujące jeszcze lepsze wyniki.
- 7.** Zaprezentowanie rozwiązania.
- 8.** Uruchomienie, monitorowanie i utrzymywanie systemu.

Oczywiście każdą listę należy dowolnie dostosowywać do potrzeb danego projektu.

Określenie problemu i przeanalizowanie go w szerszej perspektywie

- 1.** Zdefiniuj cel w kategoriach biznesowych.
- 2.** W jaki sposób będzie używane Twoje rozwiązanie?
- 3.** Jakie istnieją obecnie rozwiązania/objeścia (jeśli istnieją)?
- 4.** W jakich kategoriach należy zdefiniować problem (nienadzorowany/nadzorowany, przyrostowy/statyczny itd.)?
- 5.** W jaki sposób będzie mierzona wydajność modelu?
- 6.** Czy pomiar wydajności jest powiązany z celem biznesowym?

- 7.** Jaka jest minimalna wydajność wymagana do spełnienia celu biznesowego?
- 8.** Czy istnieją jakieś porównywalne problemy? Czy możesz wykorzystać dostępne doświadczenia lub narzędzi?
- 9.** Czy możesz skorzystać z pomocy ekspertów?
- 10.** W jaki sposób można ręcznie rozwiązać dany problem?
- 11.** Sporządź listę założeń ustalonych przez Ciebie (lub innych).
- 12.** W miarę możliwości zweryfikuj założenia.

Pozyskanie danych

Uwaga: W miarę możliwości zautomatyzuj ten etap, aby móc łatwo uzyskiwać świeże dane.

- 1.** Określ rodzaj i ilość potrzebnych danych.
- 2.** Wyznacz miejsce, w którym możesz uzyskać dane, i udokumentuj je.
- 3.** Sprawdź, jak wiele przestrzeni dyskowej będzie potrzebne na przechowywanie danych.
- 4.** Sprawdź zobowiązania prawne i w razie potrzeby uzyskaj autoryzację.
- 5.** Zdobądź uprawnienia dostępu.
- 6.** Stwórz przestrzeń roboczą (z wystarczającą pojemnością dyskową).
- 7.** Pozyskaj dane.
- 8.** Przekształć dane do formatu, który umożliwi łatwą manipulację nimi (bez zmieniania istoty samych danych).
- 9.** Upewnij się, że wrażliwe dane zostały usunięte lub zabezpieczone (np. zamaskowane).
- 10.** Sprawdź rozmiar i typ danych (szeregi czasowe, próbki, dane geograficzne itd.).
- 11.** Wydziel zestaw testowy, odłóż go i nigdy do niego nie zagładaj (żadnego podglądzania danych!).

Analiza danych

Uwaga: Spróbuj uzyskać na tym etapie wsparcie eksperta z danej dziedziny.

- 1.** Stwórz kopię analizowanych danych (w razie potrzeby przepróbkowując je do rozsądnych rozmiarów).
- 2.** Stwórz notatnik Jupyter, w którym będziesz przechowywać wyniki analizy danych.
- 3.** Określ każdy atrybut i jego parametry:
 - Nazwę.
 - Typ (kategorialne, stało-/zmiennoprzecinkowe, ograniczone/nieograniczone, tekstowe, strukturalne itd.).
 - Odsetek brakujących wartości.
 - Zaszumienie i rodzaj szumu (stochastyczny, elementy odstające, błędy zaokrąglenia itd.).

- Przydatność w określonym zadaniu.
 - Rodzaj rozkładu (gaussowski, jednorodny, logarytmiczny itd.).
- 4.** W przypadku zadań uczenia nadzorowanego określ docelowy atrybut (docelowe atrybuty).
- 5.** Zwizualizuj dane.
- 6.** Przeanalizuj korelacje pomiędzy atrybutami.
- 7.** Zastanów się, w jaki sposób można ręcznie rozwiązać problem.
- 8.** Określ obiecujące przekształcenia, które mogą zostać zastosowane.
- 9.** Określ dodatkowe dane, które mogą okazać się przydatne (zob. etap „Pozyskanie danych”).
- 10.** Udokumentuj zgromadzoną wiedzę.

Przygotowanie danych

Uwagi:

- Pracuj na kopiących danych (oryginalny zestaw danych powinien zostać nietknięty).
- Napisz funkcje dla wszystkich przeprowadzanych przekształceń; wynika to z pięciu powodów:
 - Aby można było łatwiej przygotowywać świeże dane.
 - Aby można było wprowadzać te przekształcenia w przyszłych projektach.
 - Aby oczyścić i przygotować zestaw testowy.
 - Aby oczyszczać i przygotowywać nowe przykłady po wdrożeniu projektu do środowiska produkcyjnego.
 - Aby można było traktować te funkcje przekształceń jako hiperparametry.

1. Oczyszczanie danych:

- Dostosuj lub usuń elementy odstające (nieobowiązkowe).
- Uzupełnij brakujące wartości (np. zerami, średnią, medianą...) lub usuń odpowiednie rzędy (albo kolumny).

2. Dobór cech (nieobowiązkowy):

- Usuń atrybuty, które nie dostarczają użytecznych informacji do wykonania zadania.

3. W miarę możliwości inżynieria cech:

- Zdyskretyzuj cechy ciągłe.
- Dokonaj rozkładu cech (np. kategorialne, data/godzina itd.).
- Dodaj obiecujące przekształcenia cech (np. $\log(x)$, \sqrt{x} , x^2 itd.).
- Połącz cechy w nowe, obiecujące cechy.

4. Skalowanie cech:

- standaryzuj lub normalizuj cechy.

Stworzenie krótkiej listy obiecujących modeli

Uwagi:

- Jeżeli zestaw danych jest bardzo duży, możesz chcieć próbować mniejsze zbiory uczące, dzięki czemu możesz trenować wiele różnych modeli w rozsądnie krótkim czasie (pamiętaj, że to rozwiązanie nie jest dobre dla złożonych modeli, takich jak duże sieci neuronowe lub losowe lasy).
 - Postaraj się zautomatyzować również ten etap.
1. Wyucz wiele różnych testowych wersji modeli (np. liniowe, naiwne bayesowskie, maszyny SVM, losowy las, sieć neuronowa itd.) za pomocą standardowych parametrów.
 2. Zmierz i porównaj wydajność tych modeli.
 - Dla każdego modelu wykonaj N-krotny sprawdzian krzyżowy oraz oblicz średnią i odchylenie standardowe miary wydajności dla N podzbiorów.
 3. Przeanalizuj najistotniejsze zmienne każdego algorytmu.
 4. Przeanalizuj rodzaje błędów popełnianych przez modele.
 - Jakie dane wykorzystałby człowiek do uniknięcia tych błędów?
 5. Wykonaj szybki przebieg doboru i inżynierii cech.
 6. Wykonaj jeszcze jeden albo dwa dodatkowe przebiegi pięciu powyższych czynności.
 7. Sporządź krótką listę od trzech do pięciu najbardziej obiecujących modeli, najlepiej takich, które popełniają różne rodzaje błędów.

Dostrojenie modelu

Uwagi:

- Na tym etapie należy wykorzystać jak największą ilość danych, zwłaszcza pod koniec strojenia.
 - Jak zwykle postaraj się zautomatyzować jak największą część tego procesu.
1. Dostrój hiperparametry za pomocą sprawdzianu krzyżowego.
 - Potraktuj dobrane funkcje przekształceń danych jako hiperparametry modelu, zwłaszcza jeśli nie masz co do nich pewności (np. powinnam/powiniensem zastąpić brakujące wartości zerami czy medianą? A może po prostu usunąć rzędy?).
 - Zawsze wybieraj losowe przeszukiwanie zamiast przeszukiwania siatki, jeśli dostępnych jest niewiele wartości hiperparametrów. Jeżeli proces uczenia trwa bardzo długo, lepszym rozwiążaniem może okazać się optymalizacja bayesowska (np. za pomocą procesów gaussowskich, co zostało opisane przez Jaspera Snoeka, Hugo Larochelle'a i Ryana Adamsa, <https://arxiv.org/abs/1206.2944>)¹.

¹ Jasper Snoek i in., *Practical Bayesian Optimization of Machine Learning Algorithms*, „Proceedings of the 25th International Conference on Neural Information Processing Systems” 2 (2012), s. 2951 – 2959.

2. Wypróbuj metody zespołowe. Zbiór połączonych najlepszych modeli często osiąga lepsze rezultaty od jego poszczególnych składowych.
3. Gdy już będziesz zadowolony ze swojego modelu, zmierz jego wydajność za pomocą zestawu testowego, aby określić błąd uogólnienia.



Nie dostrajaj modelu po zmierzeniu jego błędu uogólniania: spowodowałoby to przetrenowanie wobec zestawu testowego.

Zaprezentowanie rozwiązania

1. Udokumentuj postępy i dokonania.
2. Stwórz elegancką prezentację.
 - Najpierw zaprezentuj problem w szerszej perspektywie.
3. Wyjaśnij, dlaczego Twoje rozwiązanie spełnia cel biznesowy.
4. Nie zapomnij zaprezentować ciekawych spostrzeżeń dokonanych w trakcie pracy nad projektem.
 - Opisz rozwiązań, które zadziałyły i które okazały się nieskuteczne
 - Wymień ustanowione założenia i ograniczenia systemu.
5. Upewnij się, że najważniejsze odkrycia zostaną przekazane za pomocą ślicznych wizualizacji lub przystępnych stwierdzeń (np. „medianą dochodów stanowi główny predyktor cen domów”).

Do dzieła!

1. Przygotuj rozwiązanie pod środowisko produkcyjne (podłącz pod wejścia danych produkcyjnych, napisz jednostki testujące itd.).
2. Napisz kod monitorowania sprawdzający wydajność systemu w regularnych odstępach czasu i wysyłający alerty, gdy ta spadnie.
 - Pamiętaj o zjawisku powolnej degradacji: modele ulegają „rozkładowi” wraz z ewoluowaniem danych.
 - Pomiar wydajności może wymagać czynnika ludzkiego na którymś etapie potoku (np. poprzez usługi źródeł społecznościowych).
 - Monitoruj również jakość danych wejściowych (np. niesprawny czujnik wysyłający losowe wartości lub brak dynamiki danych dostarczanych przez zespół znajdujący się na wcześniejszym etapie potoku). Jest to szczególnie istotne w przypadku systemów uczenia przyrostowego.
3. Trenuj regularnie modele za pomocą świeżych danych (w miarę możliwości zautomatyzuj ten proces).

Problem dualny w maszynach wektorów nośnych

Aby zrozumieć kwestię **dualności**, musimy najpierw przyjrzeć się metodzie **mnożników Lagrange'a**. Podstawowa koncepcja polega tu na przekształceniu celu ograniczonej optymalizacji w postać nie-ograniczoną poprzez przeniesienie ograniczeń do funkcji celu. Weźmy pod uwagę prosty przykład. Założymy, że chcemy znaleźć wartości x i y minimalizujące funkcję $f(x, y) = x^2 + 2y$ podlegającą **ograniczeniu równościowemu**: $3x + 2y + 1 = 0$. Za pomocą metody mnożników Lagrange'a zaczynamy definiować nową funkcję, zwaną **lagranżjanem** (lub **funkcją Lagrange'a**): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Każde ograniczenie (w tym przypadku tylko jedno) zostaje odjęte od pierwotnego celu i pomnożone przez nową zmienną — mnożnik Lagrange'a.

Joseph-Louis Lagrange udowodnił, że jeśli (\hat{x}, \hat{y}) stanowi rozwiązanie problemu ograniczonej optymalizacji, to musi istnieć taka wartość $\hat{\alpha}$, dla której $(\hat{x}, \hat{y}, \hat{\alpha})$ stanowi **punkt stacjonarny** lagranżjana (punktym stacjonarnym nazywamy punkt, w którym wszystkie pochodne cząstkowe są równe zeru). Inaczej mówiąc, możemy obliczyć pochodne cząstkowe funkcji $g(x, y, \alpha)$ w odniesieniu do wartości x , y i α ; możemy znaleźć punkty, dla których te pochodne są równe 0; rozwiązania problemu ograniczonej optymalizacji (jeśli istnieją) muszą znajdować się wśród punktów stacjonarnych.

W omawianym przykładzie mamy do czynienia z następującymi pochodnymi cząstkowymi:

$$\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$$

Gdy wszystkie te pochodne cząstkowe są równe 0, to okazuje się, że $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, dzięki czemu możemy z łatwością stwierdzić, że $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$, $\hat{\alpha} = 1$. To jest jedyny punkt stacjonarny, a do tego spełnia warunki ograniczenia, dlatego musi stanowić rozwiązanie problemu ograniczonej optymalizacji.

Metoda ta działa jednak wyłącznie wobec ograniczeń równościowych. Na szczęście w pewnych warunkach regularizacji (przestrzeganych przez cele maszyn wektorów nośnych) może zostać ona uogólniona również do **ograniczeń nierównościowych** (np. $3x + 2y + 1 \geq 0$). Równanie C.1 przedstawia **uogólniony lagranżjan** dla problemu marginesu twardego, gdzie zmienne $\alpha^{(i)}$ są nazywane **mnożnikami Karusha-Kuhnna-Tuckera** (KKT) — muszą być one większe lub równe 0.

Równanie C.1. Uogólniony lagranżjan dla problemu marginesu twardego

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \mathbf{w}^T \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} \left(t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 \right)$$

gdzie $\alpha^{(i)} \geq 0$ dla $i = 1, 2, \dots, m$

Podobnie jak w metodzie mnożników Lagrange'a, możemy obliczyć pochodne cząstkowe i zlokalizować punkty stacjonarne. Jeżeli istnieje rozwiązanie, musi znajdować się ono wśród punktów stacjonarnych $(\hat{\mathbf{w}}, \hat{b}, \hat{\boldsymbol{\alpha}})$ przestrzegających **warunków KKT**:

- przestrzegają ograniczeń problemu: $t^{(i)} \left((\hat{\mathbf{w}})^T \mathbf{x}^{(i)} + \hat{b} \right) \geq 1 \quad \text{dla } i = 1, 2, \dots, m,$
- sprawdzają $\hat{\boldsymbol{\alpha}}^{(i)} \geq 0 \quad \text{dla } i = 1, 2, \dots, m,$,
- albo $\hat{\boldsymbol{\alpha}}^{(i)} = 0$, albo i -te ograniczenie musi być **ograniczeniem aktywnym**, co oznacza, że musi przestrzegać równości: $t^{(i)} \left((\hat{\mathbf{w}})^T \mathbf{x}^{(i)} + \hat{b} \right) = 1$. Jest to tak zwany warunek **luzu dopełniającego**.

Wynika z niego, że albo $\hat{\boldsymbol{\alpha}}^{(i)} = 0$, albo i -ta próbka znajduje się na granicy (stanowi wektor nośny).

Zwrót uwagę, że warunki KTT muszą być koniecznie spełnione, aby punkt stacjonarny stanowił rozwiązanie problemu ograniczonej optymalizacji. W pewnych sytuacjach są one również warunkami wystarczającymi. Na szczęście maszyny wektorów nośnych spełniają te warunki, zatem dowolny punkt stacjonarny również je spełniający będzie stanowił rozwiązanie problemu ograniczonej optymalizacji.

Mogemy obliczyć pochodne cząstkowe uogólnionego lagranżjana w odniesieniu do \mathbf{w} i b za pomocą równania C.2.

Równanie C.2. Pochodne cząstkowe uogólnionego lagranżjana

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = - \sum_{i=1}^m \alpha^{(i)} t^{(i)}$$

Gdy te pochodne cząstkowe są równe 0, otrzymujemy równanie C.3.

Równanie C.3. Własności punktów stacjonarnych

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} = 0$$

Jeśli wstawimy te wyniki do definicji uogólnionego lagranżjana, pewne człony znikną i otrzymamy równanie C.4.

Równanie C.4. Forma dualna problemu SVM

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \alpha) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

$$\text{gdzie } \alpha^{(i)} \geq 0 \quad \text{dla } i = 1, 2, \dots, m$$

Celem jest teraz znalezienie wektora $\hat{\mathbf{a}}$ minimalizującego tę funkcję, gdzie $\hat{\mathbf{a}}^{(i)} \geq 0$ dla wszystkich przykładów. Taki problem ograniczonej optymalizacji stanowi poszukiwany przez nas problem dualny.

Po znalezieniu optymalnego wektora $\hat{\mathbf{a}}$ możemy obliczyć wektor $\hat{\mathbf{w}}$ za pomocą pierwszego wiersza w równaniu C.3. W celu wyliczenia wartości \hat{b} możemy wykorzystać fakt, że wektor nośny musi sprawdzić, czy $t^{(i)} ((\hat{\mathbf{w}})^T \mathbf{x}^{(i)} + \hat{b}) = 1$, zatem jeśli k-ta próbka jest wektorem nośnym (tj. $\hat{\alpha}^{(k)} > 0$), możemy użyć go do obliczenia $\hat{b} = t^{(k)} - \hat{\mathbf{w}}^T \mathbf{x}^{(k)}$. Często jednak preferowane jest obliczenie średniej ze wszystkich wektorów nośnych, co daje stabilniejszą i precyzyjniejszą wartość (patrz równanie C.5).

Równanie C.5. Oszacowanie członu obciążenia za pomocą formy dualnej

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left[t^{(i)} - \hat{\mathbf{w}}^T \mathbf{x}^{(i)} \right]$$

Różniczkowanie automatyczne

W tym dodatku wyjaśniam funkcję różniczkowania automatycznego dostępną w module TensorFlow, a także porównuję ją z innymi rozwiązaniami.

Założymy, że definiujemy funkcję $f(x, y) = x^2y + y + 2$ i potrzebujemy jej pochodnych cząstkowych $\frac{\partial f}{\partial x}$ i $\frac{\partial f}{\partial y}$, najczęściej do przeprowadzenia gradientu prostego (lub do innego algorytmu optymalizacyjnego). Mamy do wyboru następujące techniki: różniczkowanie ręczne, metodą różnic skończonych, automatyczne oraz odwrotne automatyczne. W module TensorFlow jest dostępne to ostatnie rozwiązanie, ale żeby je dobrze zrozumieć, warto poznać pozostałe metody. Przyjrzyjmy się wszystkim wymienionym możliwościom, począwszy od różniczkowania ręcznego.

Różniczkowanie ręczne

Pierwsza metoda polega na skorzystaniu z kartki, ołówka i własnej wiedzy do ręcznego obliczenia pochodnych cząstkowych. W przypadku zdefiniowanej przez nas funkcji $f(x, y)$ nie jest to trudne zadanie; musimy jedynie skorzystać z pięciu zasad:

- Pochodna ze stałej zawsze wynosi 0.
- Pochodną wyrażenia λx jest λ (gdzie λ symbolizuje stałą).
- Pochodną wyrażenia x^λ jest $\lambda x^{\lambda-1}$, zatem w przypadku x^2 uzyskujemy pochodną $2x$.
- Pochodna sumy funkcji stanowi sumę pochodnych tych funkcji.
- Pochodna iloczynu λ i funkcji stanowi iloczyn λ i pochodnej tej funkcji.

Dzięki powyższym regułom możemy wyliczyć pochodne widoczne w równaniu D.1.

Równanie D.1. Pochodne cząstkowe funkcji $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Metoda ta może stać się bardzo męcząca w przypadku bardziej skomplikowanych funkcji, a do tego jesteśmy narażeni na popełnianie pomyłek. Dobra informacja jest taka, że istnieją inne techniki. Przyjrzyjmy się metodzie różnic skończonych.

Metoda różnic skończonych

Jak pamiętamy, pochodna $h'(x_0)$ funkcji $h(x)$ w punkcie x_0 stanowi nachylenie funkcji w tym punkcie. Mówiąc dokładniej, pochodna jest definiowana jako granica nachylenia linii prostej przechodzącej przez punkt x_0 i inny punkt x tej funkcji w miarę dążenia x nieskończoności blisko ku x_0 (równanie D.2).

Równanie D.2. Definicja pochodnej funkcji $h(x)$ w punkcie x_0

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} = \\ &= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} \end{aligned}$$

Gdybyśmy więc chcieli obliczyć pochodną cząstkową funkcji $f(x, y)$ w odniesieniu do x w punkcie $x = 3$ i $y = 4$, policzylibyśmy $f(3+\varepsilon, 4) - f(3, 4)$ i podzielilibyśmy wynik przez bardzo małą wartość ε . Taki typ aproksymowania pochodnej nazywany jest **metodą różnic skończonych** (ang. *finite difference approximation*), a to szczególne równanie nosi nazwę **ilorazu różnicowego Newtona** (ang. *Newton's difference quotient*). Jego dokładnym odwzorowaniem jest następujący listing:

```
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)
```

Niestety wyniki są nieprecyzyjne (a rozbieżności stają się jeszcze wyraźniejsze wraz ze wzrostem skomplikowania funkcji). Właściwymi wynikami powinny być, odpowiednio, 24 i 10, ale zamiast tego otrzymujemy:

```
>>> print(df_dx)  
24.000039999805264  
>>> print(df_dy)  
10.000000000331966
```

Zwrót uwagi, że w celu obliczenia obydwu pochodnych cząstkowych musimy wywołać funkcję `f()` przynajmniej trzy razy (w powyższym listingu wywołaliśmy ją czterokrotnie, ale algorytm ten można zoptymalizować). Gdybyśmy mieli do czynienia z tysiącem parametrów, należałoby tę funkcję wywołać przynajmniej 1001 razy. W przypadku dużych sieci neuronowych metoda różnic skończonych okazuje się bardzo niewydajna.

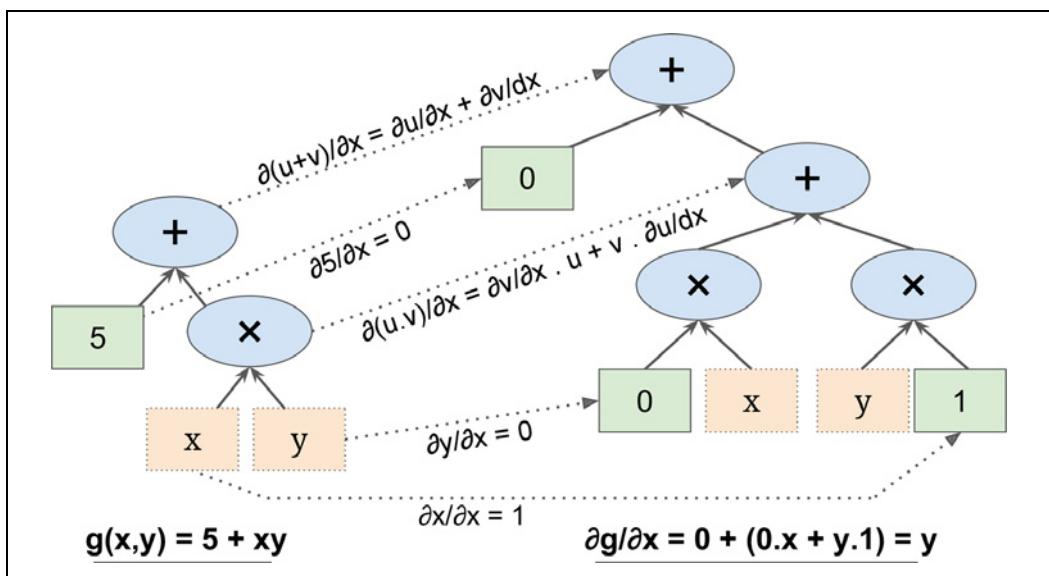
Jednak olbrzymia prostota implementacji tej metody sprawia, że stanowi ona znakomite narzędzie sprawdzania poprawności implementacji innych technik. Na przykład jeżeli uzyskane za jej

pomocą wyniki są niezgodne z rezultatami różniczkowania ręcznego, oznacza to, że najprawdopodobniej w tym drugim przypadku wdał się jakiś błąd.

Dotychczas analizowaliśmy dwa sposoby obliczania gradientów: za pomocą różniczkowania ręcznego i metody różnic skończonych. Niestety żaden z nich nie nadaje się do uczenia wielkoskalowych sieci neuronowych. Przyjrzyjmy się więc różniczkowaniu automatycznemu, najpierw w jego klasycznej odmianie.

Różniczkowanie automatyczne

Rysunek D.1 pokazuje mechanizm działania różniczkowania automatycznego na jeszcze prostszej funkcji, $g(x, y) = 5 + xy$. Graf symbolizujący tę funkcję został zaprezentowany po lewej stronie. Rezultat przeprowadzenia różniczkowania automatycznego został zaprezentowany po prawej stronie: widzimy tam pochodną cząstkową $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (w ten sam sposób moglibyśmy otrzymać pochodną cząstkową w odniesieniu do y).



Rysunek D.1. Różniczkowanie automatyczne

Algorytm będzie realizował operacje grafu obliczeniowego od wejść do wyjść (stąd angielska nazwa *forward-mode autodiff*). Rozpoczyna działanie od uzyskania pochodnych cząstkowych z liści. Węzeł (5) zwraca pochodną 0, gdyż pochodna ze stałej zawsze wynosi zero. Zmienna x zwraca pochodną w postaci stałej 1, gdyż $\frac{\partial x}{\partial x} = 1$, natomiast zmienna y zwraca pochodną 0, gdyż $\frac{\partial y}{\partial x} = 0$ (byłoby na odwrót, gdybyśmy wyliczali pochodną cząstkową w odniesieniu do y).

Możemy teraz przejść na wyższy poziom grafu do węzła iloczynu w funkcji g . Dzięki rachunkowi różniczkowemu wiemy, że pochodna iloczynu dwóch funkcji u i v wynosi $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times v$. Możemy zatem stworzyć znaczną część grafu widoczną po jego prawej stronie, reprezentującą operację $0 \times x + y \times 1$.

W końcu możemy również zająć się węzłem sumowania w funkcji g . Jak już wiemy, pochodna sumy funkcji stanowi sumę pochodnych tych funkcji. Wystarczy więc, że stworzymy węzeł sumowania i połączymy go z już obliczonymi fragmentami grafu. Otrzymujemy prawidłową pochodną cząstkową: $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

Możemy jednak uprościć (znacznie) ten proces. Możemy wyciąć z grafu kilka niepotrzebnych operacji, co pozwala nam uzyskać znacznie mniejszy, zaledwie jednowęzłowy graf: $\frac{\partial g}{\partial x} = y$. W tym przypadku uproszczenie jest dość łatwym procesem, ale w przypadku bardziej skomplikowanej funkcji różniczkowanie automatyczne może wygenerować olbrzymi graf, którego uproszczenie może być bardzo trudne do wykonania i może prowadzić do nieoptymalnej wydajności.

Zwróci uwagę, że rozpoczęliśmy od grafu obliczeniowego, a mechanizm różniczkowania automatycznego wygenerował jeszcze jeden graf. Mamy tu do czynienia z **różniczkowaniem symbolicznym** (ang. *symbolic differentiation*), które cechuje się dwiema przydatnymi własnościami: po pierwsze, po utworzeniu grafu obliczeniowego pochodnej możemy używać go dowolną liczbę razy do obliczania pochodnych danej funkcji dla dowolnej wartości x i y ; po drugie, otrzymany graf możemy przetworzyć za pomocą różniczkowania automatycznego po to, aby w razie potrzeby otrzymać pochodne drugiego rzędu (czyli pochodne z pochodnych). Moglibyśmy nawet obliczyć w ten sposób pochodne trzeciego rzędu itd.

Możliwe jest jednak również różniczkowanie automatyczne bez potrzeby konstruowania grafu (tzn. numerycznie, a nie symbolicznie) poprzez obliczanie wartości pośrednich na bieżąco. Jednym ze sposobów jest wykorzystanie **liczb dualnych**, które są dziwnymi, ale fascynującymi liczbami w postaci $a+b\varepsilon$, gdzie a i b stanowią liczby rzeczywiste, natomiast ε jest nieskończonie małą liczbą taką, że $\varepsilon^2 = 0$ (ale $\varepsilon \neq 0$). Możemy interpretować liczbę dualną $42+24\varepsilon$ jako wartość przypominającą 42,0000...000024 wypełnioną nieskończoną liczbą zer (oczywiście jest to duże uproszczenie, ukazujące koncepcję liczb dualnych w bardzo ogólny sposób). Liczba dualna jest reprezentowana w pamięci jako para wartości zmienoprzecinkowych. Na przykład liczba $42+24\varepsilon$ ma postać pary (42,0, 24,0).

Jak widać w równaniu D.3, możemy wykonywać różne operacje na liczbach dualnych, np. sumowanie, mnożenie itd.

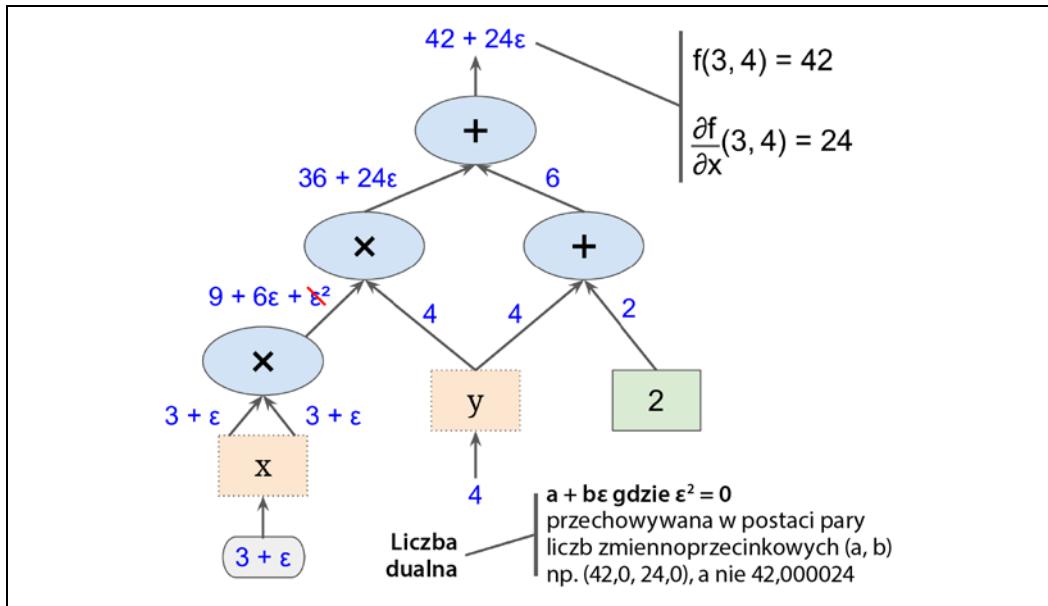
Równanie D.3. Przykłady kilku działań na liczbach dualnych

$$\lambda(a+b\varepsilon) = \lambda a + \lambda b\varepsilon$$

$$(a+b\varepsilon) + (c+d\varepsilon) = (a+c) + (b+d)\varepsilon$$

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

Najważniejsze jednak, że możemy wykorzystać własność $h(a+b\varepsilon) = h(a)+b \times h'(a)\varepsilon$, zatem obliczyszy $h(a+\varepsilon)$, uzyskamy zarówno wartość funkcji $h(a)$, jak i jej pochodną $h'(a)$. Widzimy na rysunku D.2, w jaki sposób różniczkowanie automatyczne oblicza pochodną cząstkową funkcji $f(x, y)$ w odniesieniu do x , przy założeniu, że $x = 3$ i $y = 4$. Wystarczy, że obliczymy $f(3+\varepsilon, 4)$; uzyskamy w ten sposób liczbę dualną, której pierwsza składowa będzie równa wartości $f(3, 4)$, a druga składowa to $\frac{\partial f}{\partial x}(3, 4)$.



Rysunek D.2. Różniczkowanie automatyczne za pomocą liczb dualnych

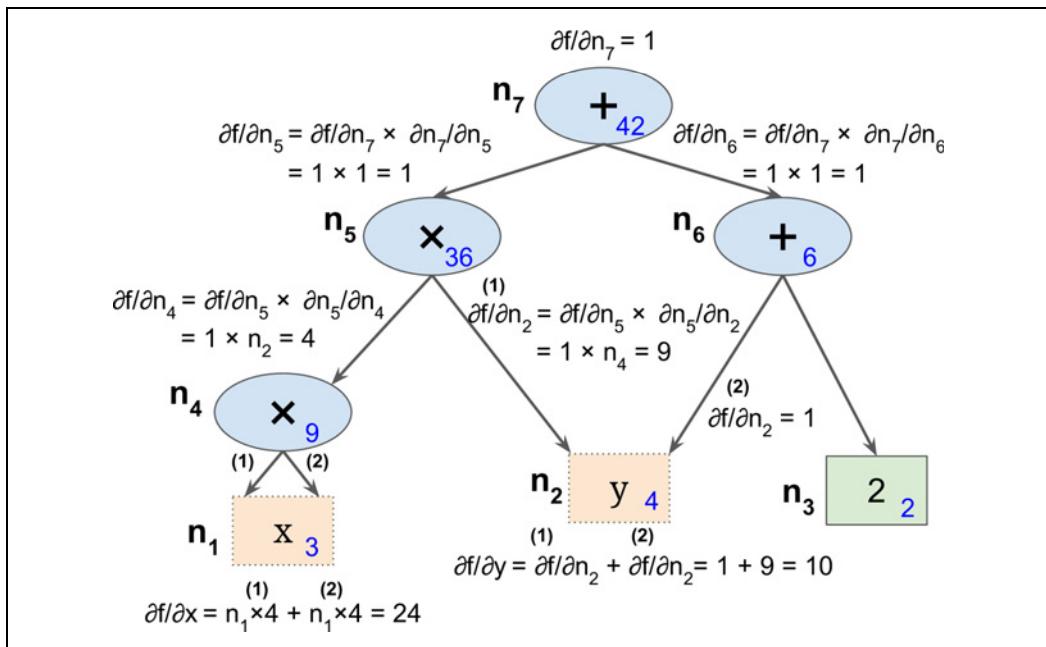
W celu obliczenia pochodnej $\frac{\partial f}{\partial y}(3, 4)$ należałoby wykonać jeszcze jeden przebieg po grafie, tym razem jednak używając punktów $x = 3$ i $y = 4 + \varepsilon$.

Zatem różniczkowanie automatyczne jest o wiele dokładniejsze od metody różnic skończonych, ale ma tę samą wadę. Przynajmniej w przypadku systemów o dużej liczbie wejść i kilku wyjściach (przykładem są sieci neuronowe): obecność tysiąca parametrów oznaczałaby konieczność wykonania 1000 przebiegów w celu obliczenia wszystkich pochodnych cząstkowych. Właśnie w tym momencie wkracza do akcji odwrotne różniczkowanie automatyczne: oblicza ono wszystkie pochodne cząstkowe w zaledwie dwóch przebiegach po grafie. Sprawdźmy, jak wygląda jego mechanizm działania.

Odwrotne różniczkowanie automatyczne

Odwrotne różniczkowanie automatyczne jest rozwiązaniem wykorzystywanym w module TensorFlow. Najpierw wykonuje przebieg w przód po grafie (tj. od wejść do wyjść) w celu obliczenia wartości każdego węzła. Następnie wykonuje drugi przebieg, tym razem wstecz (od wyjścia do wejść), podczas którego

oblicza pochodne cząstkowe. Nazwa angielska *reverse-mode* bierze się właśnie z tego drugiego przebiegu, w którym gradienty przepływają w przeciwnym kierunku. Na rysunku D.3 zostało zaprezentowane drugi przebieg (wsteczny). W trakcie pierwszego przebiegu zostały obliczone wartości wszystkich węzłów, począwszy od punktów $x = 3$ i $y = 4$. Wartości te widzimy w dolnym prawym rogu każdego węzła (np. $\partial f / \partial x = 9$). Dla zachowania przejrzystości oznaczyłem poszczególne węzły nazwami od n_1 do n_7 . Węzeł wyjściowy został oznaczony jako $n_7: f(3, 4) = n_7 = 42$.



Rysunek D.3. Odwrotne różniczkowanie automatyczne

Mechanizm działania polega na stopniowym schodzeniu w dół grafu i wyliczaniu pochodnej cząstkowej funkcji $f(x, y)$ w odniesieniu do każdego następnego węzła, aż do momentu dotarcia do węzłów zmiennych. Ważną rolę odgrywa tu **reguła łańcuchowa** ukazana w równaniu D.4.

Równanie D.4. Reguła łańcuchowa

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Skoro n_7 jest węzłem wyjściowym, $f = n_7$, zatem po prostu $\frac{\partial f}{\partial n_7} = 1$.

Podążajmy dalej wzduż grafu do węzła n_5 : jak bardzo zmienia się funkcja f , gdy zmienia się wartość węzła n_5 ? Odpowiedzią jest $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. Wiemy już, że $\frac{\partial f}{\partial n_7} = 1$, zatem potrzebujemy tylko

$\frac{\partial n_7}{\partial n_5}$. Węzeł n_7 wykonuje jedynie operację $n_5 + n_6$, dlatego dowiadujemy się, że $\frac{\partial n_7}{\partial n_5} = 1$, zatem $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Teraz możemy przejść do węzła n_4 : jak bardzo zmienia się funkcja f , gdy zmienia się wartość węzła n_4 ?

Odpowiedzią jest $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Skoro $n_5 = n_4 \times n_2$, to dowiadujemy się, że $\frac{\partial n_5}{\partial n_4} = n_2$, zatem

$$\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4.$$

Proces ten jest kontynuowany aż do osiągnięcia dna grafu. W tym momencie będziemy mieli obliczone wszystkie pochodne cząstkowe funkcji $f(x, y)$ w punkcie $x = 3$ i $y = 4$. W tym przykładzie okazuje się, że $\frac{\partial f}{\partial x} = 24$ i $\frac{\partial f}{\partial y} = 10$. Wynik jest prawidłowy!

Odwrotne różniczkowanie automatyczne stanowi bardzo potężną i dokładną technikę, zwłaszcza w przypadku dużej liczby wejść i niewielu wyjść, ponieważ wymaga ona tylko jednego przebiegu w przód i jednego wstecznego do wyliczenia wszystkich pochodnych cząstkowych dla wszystkich wyjść w odniesieniu do wszystkich wejść. Podczas uczenia sieci neuronowych chcemy zazwyczaj minimalizować funkcję straty, zatem mamy do czynienia z jednym wyjściem (funkcją straty) i do obliczenia gradientów potrzebne są tylko dwa przebiegi wzduż grafu. Ponadto odwrotne różniczkowanie automatyczne przetwarza również funkcje, które się są całkowicie różniczkowalne, pod warunkiem że będziemy chcieli obliczać pochodne cząstkowe tylko w różniczkowalnych rejonach funkcji.

Na rysunku D.3 wyniki numeryczne obliczane są na bieżąco w każdym węźle. Jednak mechanizm użyty w module TensorFlow wygląda nieco inaczej: generuje on nowy graf obliczeniowy. Innymi słowy, jest tu zaimplementowane **symboliczne**, odwrotne różniczkowanie automatyczne. W ten sposób graf obliczeniowy służący do obliczania gradientów funkcji straty w odniesieniu do wszystkich parametrów sieci neuronowej musi być wygenerowany tylko raz, po czym może być wielokrotnie realizowany, za każdym razem, gdy optymalizator musi obliczyć gradienty. Do tego w razie potrzeby może bez problemu obliczać pochodne wyższych rzędów.



Jeżeli implementujesz nowy rodzaj ogólnych operacji TensorFlow w języku C++ i chcesz, żeby był kompatybilny z techniką automatycznego różniczkowania, musisz wprowadzić funkcję zwracającą pochodne cząstkowe wyjść funkcji w odniesieniu do jej wejść. Założmy na przykład, że implementujesz funkcję obliczającą kwadrat wejść $f(x) = x^2$. W takim przypadku trzeba wprowadzić odpowiadającą jej funkcję pochodną $f'(x) = 2x$.

Inne popularne architektury sieci neuronowych

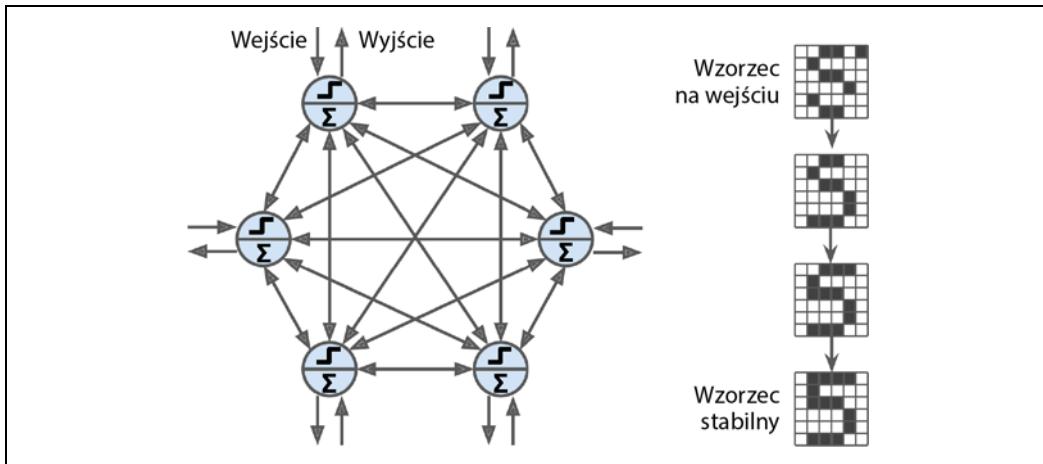
W tym dodatku przyjrzymy się побieżnie kilku historycznym architekturom sieci neuronowych, które straciły na popularności na rzecz perceptronów wielowarstwowych (zob. rozdział 10.), splotowych sieci neuronowych (zob. rozdział 14.), rekurencyjnych sieci neuronowych (zob. rozdział 15.) czy autokoderów (zob. rozdział 17.). Często wspomina się o nich w literaturze fachowej, a niektóre z nich ciągle są spotykane w różnorakich zastosowaniach, dlatego warto się z nimi zapoznać. Ponadto zapoznasz się z **głębokimi sieciami przekonań**, które jeszcze kilkanaście lat temu stanowiły szczyt osiągnięć uczenia głębokiego. Do tej pory są one obiektem intensywnych badań, zatem w niedalekiej przyszłości mogą znowu trafić na szczyty list przebojów.

Sieci Hopfielda

Sieci Hopfielda zostały zaprojektowane w 1974 roku przez W. A. Little'a, a spopularyzowane w 1982 roku przez J. Hopfielda. Stanowią one przykład **sieci pamięci asocjacyjnej**: najpierw uczymy je pewnych wzorców, a następnie po zaprezentowaniu nowego wzoru generują na wyjściu najbliższy (teoretycznie) wyuczony wzorzec. Do czasu pojawienia się lepszych technologii sieci Hopfielda były przydatne przede wszystkim w procesie rozpoznawania znaków. Najpierw sieć jest uczona poprzez prezentowanie jej obrazów zawierających przykładowe znaki (każdemu pikselowi binarnemu odpowiada jeden neuron wejściowy), a następnie po zaprezentowaniu nowego obrazu sieć ta po kilku przebiegach wyświetla najbardziej podobny wyuczony znak.

Każda sieć Hopfielda stanowi w pełni połączony graf (rysunek E.1); oznacza to, że każdy neuron jest połączony z pozostałymi neuronami. Zwrót uwagę, że obrazy cyfr na rysunku mają rozmiar 6×6 , zatem widoczna po lewej stronie sieć neuronowa powinna składać się z 36 neuronów (i zawierać 648 połączeń), ale dla zachowania przejrzystości został ukazany schemat znacznie mniejszej sieci.

W algorytmie uczącym wykorzystywana jest reguła Hebb'a (zob. punkt „Perceptron” w rozdziale 10.): w każdym obrazie uczącym wagę pomiędzy dwoma neuronami są zwiększone, jeżeli obydwa odpowiadające im piksele są włączone lub wyłączone, natomiast ulegają zmniejszeniu, jeśli piksele mają odmienne stany.



Rysunek E.1. Sieć Hopfielda

W celu zaprezentowania nowego obrazu w sieci wystarczy pobudzić neurony odpowiadające aktywnym pikselom. Zostaje następnie obliczony wynik każdego neuronu, przez co uzyskujemy nowy obraz, za pomocą którego możemy powtórzyć cały proces. Po pewnym czasie sieć osiąga stan stabilny. Zazwyczaj oznacza to ukazanie obrazu uczącego przypominającego obraz umieszczony na wejściu sieci.

Sieci Hopfielda są opisywane za pomocą tzw. **funkcji energii**. W każdym przebiegu energia sieci maleje, dlatego po określonym czasie sieć ustabilizuje się w stanie niskoenergetycznym. Algorytm uczący dostosowuje wagę w taki sposób, aby stan energii wzorców uczących był jak najmniejszy, dzięki czemu sieć będzie miała największe prawdopodobieństwo stabilizacji właśnie w którejś z tych wyuczonych konfiguracji. Niestety część wzorców niestanowiących składowych procesu uczenia również cechuje się stanem niskiej energii, dlatego sieć czasami stabilizuje się w takiej niewyuczonej konfiguracji. Są one zwane **wzorcami pasożytniczymi** (ang. *spurious patterns*).

Inną dużą wadą sieci Hopfielda jest ich niewielka skalowalność — ich pojemność pamięci stanowi zaledwie w przybliżeniu 14% liczby neuronów. Na przykład w celu klasyfikowania obrazów o rozmiarze 28x28 pikseli potrzebowalibyśmy 784 w pełni połączone neurony i 306 936 wag. Taka sieć byłaby w stanie wyuczyć się zaledwie około 110 różnych znaków (14% z 784). Mnóstwo parametrów jak na tak niewielką pamięć.

Maszyny Boltzmana

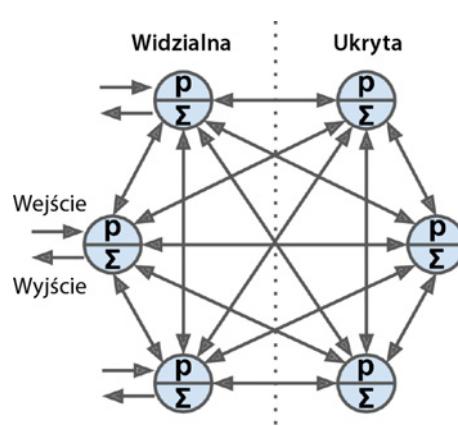
Maszyny Boltzmana zostały zaprojektowane w 1985 roku przez Geoffrey'a Hintona i Terrence'a Sejnowskiego. Podobnie jak w przypadku sieci Hopfielda tutaj mamy również do czynienia z w pełni połączonymi architekturami sieci neuronowych, podstawowymi jednostkami są tu jednak **neurony stochastyczne**: wartość wyjściowa nie jest wyznaczana za pomocą funkcji deterministycznej, lecz prawdopodobieństwa — istnieje pewna szansa, że neuron umieści na wyjściu wartość 1 lub 0. Stosowana tu funkcja prawdopodobieństwa bazuje na rozkładzie Boltzmana (wykorzystywany w mechanice statystycznej); stąd wzięła się nazwa omawianej architektury. Równanie E.1 służy do obliczania prawdopodobieństwa wyświetlenia wartości 1 przez dany neuron.

Równanie E.1. Prawdopodobieństwo, że i-ty neuron umieści na wyjściu wartość 1

$$p(s_i^{(następny_krok)} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{i,j} s_j + b_i}{T}\right)$$

- s_j to stan i-tego neuronu (0 lub 1),
- $w_{i,j}$ jest wagą połączenia pomiędzy i-tym i j-tym neuronem; zwróć uwagę, że $w_{i,i} = 0$,
- b_i oznacza człon obciążenia i-tego neuronu; możemy go zaimplementować, wstawiając neuron obciążający do sieci,
- N wyznacza liczbę neuronów w sieci,
- T to wartość zwana **temperaturą** sieci; wraz ze wzrostem temperatury zwiększa się losowość wyników (tzn. że prawdopodobieństwo zbliża się do wartości 50%),
- σ określa funkcję logistyczną.

Neurony w maszynach Boltzmanna dzielą się na dwie grupy: **jednostki widoczne** i **jednostki ukryte** (rysunek E.2). Wszystkie neurony działają w taki sam stochastyczny sposób, ale to neurony widoczne otrzymują dane wejściowe i dają wyniki na wyjście.



Rysunek E.2. Maszyna Boltzmanna

Z powodu stochastycznej natury maszyny Boltzmanna nigdy nie osiągnie ustalonej, stabilnej konfiguracji, zamiast tego będzie przeskakiwać pomiędzy wieloma konfiguracjami. Jeśli pozostawimy ją uruchomioną przez wystarczająco długi czas, prawdopodobieństwo zaobserwowania określonej konfiguracji będzie stanowiło jedynie funkcję wag połączeń i członów obciążień, a nie pierwotnej konfiguracji (na drodze analogii, jeśli będziemy odpowiednio dugo tasować karty, konfiguracja talii przestanie być zależna od początkowego układu kart). Mówimy, że gdy sieć osiągnie stan, w którym pierwotna konfiguracja zostaje „zapomniana”, znajduje się w **równowadze cieplnej** (mimo że konfiguracja bez przerwy ulega zmianie). Poprzez odpowiedni dobór parametrów, osiągnięcie równowagi cieplnej przez sieć oraz obserwację jej stanu jesteśmy w stanie symulować szeroki zakres rozkładów prawdopodobieństwa. Jest to tak zwany **model generatywny**.

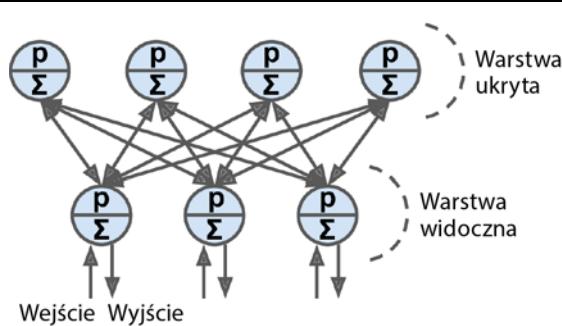
Uczenie maszyny Boltzmanna oznacza znalezienie parametrów pozwalających uzyskać sieci rozkład prawdopodobieństwa przypominający rozkład zbioru danych uczących. Przykładowo jeśli występują w sieci trzy neurony widoczne, a zestaw uczący składa się w 75% z trypletów (0, 1, 1), w 10% z trypletów (0, 0, 1) i w 15% z trypletów (1, 1, 1), to po wytrenowaniu maszyny Boltzmanna moglibyśmy generować losowe tryplety binarne o mniej więcej identycznym rozkładzie prawdopodobieństwa. To znaczy, że w 75% przypadków otrzymywaliśmy tryplet (0, 1, 1).

Taki model generatywny może być wykorzystywany na wiele sposobów. Na przykład, jeśli wytrenujemy go za pomocą obrazów, a do sieci wprowadzimy niepełny lub zaszumiony obraz, to zostanie on w zadowalającym stopniu „zrekonstruowany”. Modele generatywne przydają się również w zadaniach dotyczących klasyfikacji. Wystarczy dodać kilka neuronów widocznych kodujących klasy obrazów uczących (np. wstaw 10 neuronów widocznych i aktywuj tylko piąty neuron podczas uczenia sieci rozpoznawania cyfry 5). Teraz po zaprezentowaniu sieci nowego obrazu sieć automatycznie będzie aktywować odpowiednie neurony widoczne i wskazywać klasę, do jakiej on przynależy (np. będzie pobudzany piąty neuron widoczny, jeśli na danym rysunku będzie znajdować się cyfra 5).

Niestety nie znamy skutecznego sposobu uczenia maszyn Boltzmanna. Jednak zostały zaprojektowane wydajne algorytmy uczące **ograniczone maszyny Boltzmanna** (ang. *restricted Boltzmann machines* — RBM).

Ograniczone maszyny Boltzmanna

W ograniczonej maszynie Boltzmanna nie ma połączeń pomiędzy poszczególnymi neuronami widocznymi lub neuronami ukrytymi; połączone są ze sobą jedynie jednostki widoczne i ukryte. Na przykład na rysunku E.3 widzimy maszynę RBM zawierającą trzy jednostki widoczne i cztery ukryte.



Rysunek E.3. Ograniczona maszyna Boltzmanna

W 2005 roku Geoffrey Hinton i Miguel Á. Carreira-Perpiñán (<http://www.cs.toronto.edu/~fritz/absp/cdmiguel.pdf>) zaprezentowali bardzo wydajny algorytm uczenia zwany **rozbieżnością kontrastową** (ang. *contrastive divergence*)¹. Jego mechanizm działania wygląda następująco: najpierw dostarcza przykład uczący x do sieci, wyznaczając stany neuronów widocznych x_1, x_2, \dots, x_n . Następnie

¹ Miguel Á. Carreira-Perpinan i Geoffrey E. Hinton, *On Contrastive Divergence Learning*, „Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics” (2005), s. 59 – 66.

wyliczane są stany jednostek ukrytych za pomocą wzoru stochastycznego zaprezentowanego w równaniu E.1. W ten sposób uzyskujemy wektor ukryty \mathbf{h} (gdzie wartość h_i jest równa stanowi i-tej jednostki). Teraz przy użyciu tego samego równania stochastycznego obliczamy stan jednostek widocznych. Otrzymujemy dzięki temu wektor \mathbf{x}' . Ponownie obliczamy stan jednostek ukrytych i otrzymujemy wektor \mathbf{h}' . Jesteśmy teraz w stanie zaktualizować wagę każdego połączenia za pomocą reguły zaprezentowanej w równaniu E.2, gdzie η jest współczynnikiem uczenia.

Równanie E.2. Aktualizacja wagi w algorytmie rozbieżności kontrastowej

$$w_{i,j} \leftarrow w_{i,j} + \eta (\mathbf{x}\mathbf{h}^T - \mathbf{x}'\mathbf{h}'^T)$$

Dużą zaletą tego algorytmu jest fakt, że nie musimy czekać na osiągnięcie równowagi cieplnej przez sieć: wykonuje przebieg do przodu, wstecz, jeszcze raz do przodu i to wszystko. Dzięki temu okazuje się on nieporównywalnie wydajniejszy od poprzednich algorytmów; to on był kluczowym składnikiem sukcesu metod uczenia głębokiego bazującego na stosowych maszynach RBM.

Głębokie sieci przekonań

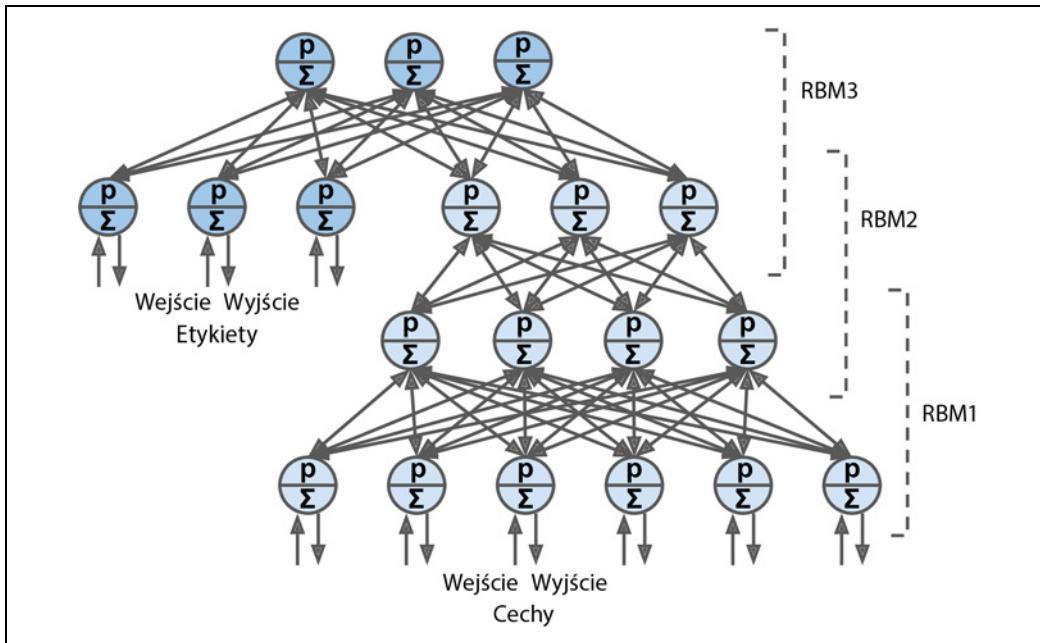
Możemy stworzyć wielowarstwowe maszyny RBM; jednostki ukryte pierwszej ograniczonej maszyny Boltzmanna stają się jednostkami widocznymi drugiej maszyny itd. Tego typu stos maszyn RBM nazywany jest głęboką siecią przekonań (ang. *deep belief network* — DBN).

Jeden ze studentów Geoffreya Hintona, Yee-Whye Teh, zauważył, że jest możliwe uczenie głębokich sieci przekonań po jednej warstwie naraz przy użyciu algorytmu zbieżności kontrastowej, począwszy od dolnej warstwy i stopniowo wspinając się po kolejnych warstwach. Obserwacja ta zaowocowała przełomowym artykułem, który w 2006 roku wywołał trzęsienie ziemi w świecie uczenia głębokiego (<http://www.cs.toronto.edu/~hinton/absps/ncfast.pdf>)².

Głębokie sieci przekonań, podobnie jak maszyny RBM, uczą się odtwarzać rozkład prawdopodobieństwa zestawu danych wejściowych bez żadnej formy nadzorowania. Jednak osiągają znacznie lepsze rezultaty z tego samego powodu, dla którego głębokie sieci neuronowe są wydajniejsze od mniej rozbudowanych architektur: rzeczywiste dane często są uporządkowane w hierarchiczny sposób, z czego potrafią skorzystać sieci DBN. Niższe warstwy uczą się rozpoznawać podstawowe cechy w danych wejściowych, natomiast wyższe warstwy trenują określanie bardziej zaawansowanych wzorców.

Zarówno ograniczone maszyny Boltzmanna, jak i głębokie sieci przekonań są zasadniczo nienadzorowane, możemy jednak uczyć je w sposób nadzorowany, dołączając jednostki widoczne reprezentujące etykiety. Ponadto wspaniała jest właściwość sieci DBN pozwalająca na ich uczenie w sposób częściowo nadzorowany. Rysunek E.4 przedstawia konfigurację częściowo nadzorowanej sieci DBN.

² Geoffrey E. Hinton i in., *A Fast Learning Algorithm for Deep Belief Nets*, „Neural Computation” 18 (2006), s. 1527 – 1554.



Rysunek E.4. Głęboka sieć przekonań skonfigurowana do uczenia częściowo nadzorowanego

Najpierw maszyna RBM 1 jest trenowana w sposób nienadzorowany. Uczy się rozpoznawać ogólne cechy w danych uczących. Dalej następuje proces uczenia maszyny RBM 2, gdzie rolę jednostek wejściowych pełnią neurony ukryte maszyny RBM 1; również w tym przypadku proces uczenia jest nienadzorowany. Są tutaj rozpoznawane bardziej skomplikowane cechy (zwróć uwagę, że na jednostki ukryte maszyny RBM składają się tylko trzy neurony znajdujące się po prawej stronie rysunku, neurony etykiet stanowią zupełnie oddzielną kwestię). Moglibyśmy w ten sposób stworzyć jeszcze kilka warstw ograniczonych maszyn Boltzmann, ale omawiany przykład powinien dobrze ilustrować koncepcję. Jak na razie cały proces uczenia jest całkowicie nienadzorowany. Na koniec maszyna RBM 3 jest uczona za pomocą jednostek ukrytych maszyny RBM 2 pełniących funkcję wejść, a także dodatkowych jednostek widocznych reprezentujących etykiety docelowe (np. wektor „gorącojedynkowy” symbolizujący klasę przykładu). W ten sposób trzecia maszyna uczy się kojarzyć rozbudowane cechy z etykietami uczącymi. Ten etap uczenia jest nadzorowany.

Po zakończeniu treningu, jeśli dostarczymy maszynie RBM 1 nowy przykład, sygnał zostanie przesłany do maszyny RBM 2, dalej na sam szczyt maszyny RBM 3, a następnie w dół do jednostek etykiet; tutaj powinien zostać wzbudzony neuron przechowujący odpowiednią etykietę. W taki właśnie sposób głęboka sieć przekonań może być użyta w zadaniach dotyczących klasyfikacji.

Niezwykłą przydatną zaletą takiej strategii uczenia częściowo nadzorowanego jest fakt, że nie potrzebujemy zbyt wielu oznakowanych danych uczących. Jeśli nienadzorowane maszyny RBM spełniają dobrze swoje zadanie, to wymagana byłaby jedynie niewielka ilość oznakowanych próbek uczących przypadających na każdą klasę. W analogiczny sposób dziecko uczy się rozpoznawać obiekty bez konieczności jego nadzorowania, zatem jeśli wskażesz krzesło i powiesz „krzesło”, dziecko nauczy się kojarzyć wyraz „krzesło” z całą klasą obiektów, które nauczyło się samodzielnie rozpoznawać.

Nie musisz wskazywać każdego krzesła i mówić, że jest krzesłem; wystarczy zaledwie kilka przykładów (tyle, żeby upewnić dziecko, że rzeczywiście masz na myśli dany przedmiot, nie jego kolor lub ktoś z jego części).

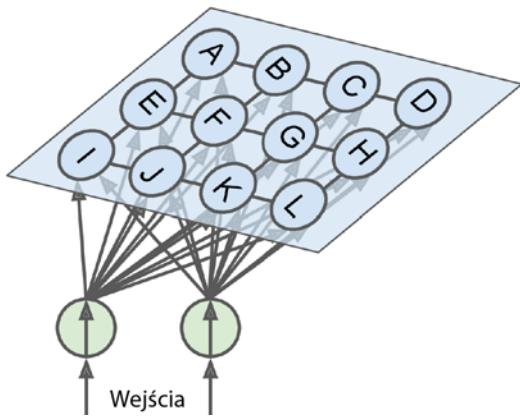
Co zdumiewające, sieci DBN mogą działać również w drugą stronę. Jeśli pobudzimy jedną z jednostek etykiet, sygnał najpierw dotrze do jednostek ukrytych maszyny RBM 3, następnie zejdzie do maszyny RBM 2, a stamtąd do maszyny RBM 1, gdzie nowa próbka zostanie umieszczona na wyjściach neuronów widocznych. Taka nowa próbka będzie przypominała standardową próbkę należącą do klasy wskazywanej przez pobudzony neuron etykiety. Takie zdolności twórcze głębokich sieci przekonań są całkiem spore. Na przykład w ten sposób jesteśmy w stanie generować automatyczne podpisy obrazów i odwrotnie: najpierw sieć DBN jest uczona (w sposób nienadzorowany) do rozpoznawania cech na rysunkach, a druga głęboka sieć przekonań uczy się (również bez nadzoru) określania cech w zestawach podpisów (np. wyraz „samochód” często występuje wraz ze słowem „pojazd”). Ponadobybywa maszynami RBM znajduje się trzecia maszyna, która jest trenowana za pomocą obrazów z dołączonymi podpisami; uczy się ona kojarzyć skomplikowane cechy obrazów z zaawansowanymi cechami podpisów. Teraz jeśli dostarczymy sieci DBN zdjęcie samochodu, sygnał przejdzie przez całą sieć aż do najwyższej maszyny RBM, a stamtąd w dół do maszyny zajmującej się podpisami, po czym wróci na sam dół sieci i wygeneruje podpis. Z powodu stochastycznej natury maszyn RGM i sieci DBN treść podpisu będzie się za każdym razem nieco różnić, jednak najczęściej będzie dopasowana do obrazu. Jeśli wygenerujemy w ten sposób kilkaset podpisów, te najczęściej występujące będą prawdopodobnie stanowiły dobry opis obrazu³.

Mapy samoorganizujące

Mapy samoorganizujące (ang. *self-organizing maps*) różnią się zasadniczo od dotychczas omówionych architektur sieci neuronowych. Służą one do generowania małowymiarowej reprezentacji wielowymiarowego zestawu danych, najczęściej w celu wizualizacji, analizy skupień lub klasyfikacji. Neurony są rozproszone na mapie (zazwyczaj do wizualizacji służy mapa dwuwymiarowa, ale może składać się ona z dowolnej liczby wymiarów), takiej jak widoczna na rysunku E.5, a każdy z nich zawiera ważone połoczenie z każdym wejściem (zwróć uwagę, że na rysunku widnieją tylko dwa wejścia, zazwyczaj jednak jest ich znacznie więcej, gdyż podstawowym zadaniem sieci SOM jest redukcja wymiarowości).

Po wytrenowaniu sieci możemy dostarczyć jej nowy przykład, co spowoduje pobudzenie tylko jednego neuronu (tj. jednego punktu na mapie): mianowicie neuronu, którego wektor wag jest najbliższym wektorowi wejściowemu. Generalnie próbki znajdujące się blisko siebie w pierwotnej przestrzeni wejściowej będą pobudzać neurony znajdujące się blisko siebie na mapie. Z tego właśnie powodu sieć SOM przydaje się do wizualizacji (pozwala ona szczegółowo łatwo wykrywać skupienia na mapie), ale także do takich zastosowań, jak rozpoznawanie mowy. Przykładowo, jeżeli każda próbka reprezentuje nagranie dźwiękowe osoby wymawiającej samogłoskę, to różne sposoby wypowiadania samogłoski „a” będą pobudzać neurony na określonym obszarze mapy, z kolei samogłoska „e” będzie aktywować jej inny rejon, natomiast dźwięk pośredni będzie uaktywniał neurony znajdujące się gdzieś w połowie drogi pomiędzy tymi samogłoskami.

³ W następującym filmiku Geoffrey Hinton objaśnia szczegóły tego rozwiązania i demonstruje przykład: <https://homl.info/137>.



Rysunek E.5. Mapy samoorganizujące



Istotną cechą odróżniającą omawiany algorytm od technik redukcji wymiarowości opisanych w rozdziale 8. jest fakt, że wszystkie przykłady są rzutowane na dyskretną liczbę punktów w małymiarowej przestrzeni (po jednym punkcie na każdy neuron). W przypadku występowania niewielkiej liczby neuronów technika ta bardziej nadaje się do analizy skupień niż do redukowania wymiarowości.

Algorytm uczący jest nienadzorowany. Działa on na zasadzie rywalizacji poszczególnych neuronów pomiędzy sobą. Najpierw wszystkie wagi są losowo inicjowane. Następnie równie losowo jest wybierana próbka ucząca i dostarczana do sieci. Wszystkie neurony obliczają odległość pomiędzy ich wektorem wag a wektorem wejściowym (jest to zupełnie odmienne zachowanie od dotychczas omawianych sztucznych neuronów). Wygrywa neuron uzyskujący w wyniku najmniejszą odległość, po czym modyfikuje wagę w taki sposób, żeby jeszcze bardziej zmniejszyć ten dystans, dzięki czemu zwiększa swoje szanse na „przejmowanie” kolejnych podobnych próbek. „Rekrutuje” on również sąsiadujące neurony i ich wagi również zostają zaktualizowane w sposób zmniejszający nieznacznie odległość do wektora wejściowego (jednak stopień aktualizacji ich wag nie jest tak znaczący, jak w przypadku zwycięskiego neuronu). Teraz algorytm losowo dobiera kolejną próbke i powtarza cały proces. Dzięki tej metodzie sąsiadujące neurony stopniowo dążą do specjalizacji⁴.

⁴ Wyobraź sobie klasę, w której wszystkie dzieci mają podobne umiejętności. Jedno dziecko okazuje się nieco lepsze w grze w koszykówkę. Czuje ono motywację, żeby częściej trenować, zwłaszcza z przyjaciółmi. Po pewnym czasie ta grupka przyjaciół staje się tak dobra, że reszta klasy nie jest w stanie z nimi konkurować. Nic nie szkodzi, gdyż inne dzieci specjalizują się w innych dziedzinach. W końcu całą klasę tworzą grupki małych specjalistów.

Specjalne struktury danych

W tym dodatku przyjrzymy się bardzo pobicie strukturom danych obsługiwanych przez moduł TensorFlow, gdyż nie ograniczają się one wyłącznie do standardowych tensorów stało- i zmiennoprzecinkowych. Do takich specjalnych struktur danych należąła łańcuchy znaków, tensory nierówne, tensory rzadkie, tablice tensorowe, zbiory i kolejki.

Łańcuchy znaków

Tensory mogą przechowywać bajtowe łańcuchy znaków, które przydają się szczególnie w zadaniach przetwarzania języka naturalnego (zob. rozdział 16.):

```
>>> tf.constant(b"uczenie maszynowe")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'uczenie maszynowe'>
```

Jeżeli spróbujesz stworzyć tensor zawierający łańcuch znaków Unicode, moduł TensorFlow automatycznie przekształci go do formatu UTF-8:

```
>>> tf.constant("caf ")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xaa9'>
```

Możliwe jest także tworzenie tensorów reprezentujących łańcuchy znaków Unicode. Wystarczy utworzyć dowolną tablicę zawierającą 32-bitowe wartości stałoprzecinkowe, z których każda będzie odpowiadała punktowi kodowemu Unicode¹:

```
>>> tf.constant([ord(c) for c in "caf "])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
numpy=array([ 99,  97, 102, 233], dtype=int32)>
```



W tensorach typu `tf.string` długość łańcucha znaków nie stanowi jednego z wymiarów tensora. Innymi słowy łańcuchy znaków są traktowane jako wartości atomowe. Jednak w tensorze znakowym Unicode (tzn. tensorze `int32`) długość łańcucha znaków jest częścią wymiarów tensora.

Pakiet `tf.strings` zawiera kilka funkcji umożliwiających operowanie na tensorach znakowych, na przykład `length()`, która oblicza liczbę bajtów w bajtowym łańcuchu znaków (lub liczbę punktów kodowych, jeżeli wyznaczyłaś/wyznaczyłeś `unit="UTF8_CHAR"`), `unicode_encode()`, która przekształca tensor

¹ Jeżeli nie wiesz, czym są punkty kodowe Unicode, zajrzyj pod adres <https://docs.python.org/3/howto/unicode.html>.

znakowy Unicode (tzn. tensor int32) w bajtowy tensor znakowy, czy też `unicode_decode()`, która wykonuje odwrotną operację:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
    numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

Możesz także operować na tensorach zawierających wielełańcuchów znaków:

```
>>> p = tf.constant(["Café", "Coffee", "caffé", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
    numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[ 67 97 102 233 67 111 102 102 101 101 99 97
 102 102 232 21654 21857], shape=(17,), dtype=int32),
row_splits=tf.Tensor([ 0 4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
[99, 97, 102, 102, 232], [21654, 21857]]>
```

Zwróć uwagę, że rozkodowanełańcuchy znaków są przechowywane w obiekcie `RaggedTensor`. Co to jest?

Tensorzy nierówne

Tensor nierówny to specyficzna odmiana tensora, reprezentująca listę tablic o różnych rozmiarach. Mówiąc ogólniej, jest to tensor zawierający co najmniej jeden **wymiar nierówny** (ang. *ragged dimension*), czyli wymiar, którego poszczególne odcinki mogą mieć różne długości. W tensorze nierównym `r` drugi wymiar jest wymiarem nierównym. We wszystkich tensorach nierównych pierwszy wymiar jest zawsze standardowy (tzw. **wymiar jednorodny** — ang. *uniform dimension*).

Wszystkie elementy tensora nierównego `r` są standardowymi tensorami. Spójrzmy przykładowo na drugi element następującego tensora nierównego:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

Pakiet `tf.ragged` zawiera kilka funkcji służących do tworzenia i stosowania tensorów nierównych. Utwórzmy drugi tensor nierówny za pomocą funkcji `tf.ragged.constant()` i połączmy go z pierwszym tensorem nierównym wzduż osi 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

Wynik nie jest zbyt zaskakujący: tensory z obiektu r2 zostały dołączone po tensorach z obiektem r wzdłuż osi 0. A gdybyśmy zrobili to samo, ale wzdłuż osi 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101, 71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

Zwróć uwagę, że tym razem został połączony *i*-ty tensor z obiektem r z *i*-tym tensorem z obiektem r3. Jest to dość niespodziewane, ponieważ wszystkie te tensory mogą mieć różną długość.

Jeżeli wywołasz metodę `to_tensor()`, dane tensory nierówne zostaną przekształcone w tensory standardowe, gdzie wszystkie krótsze tensory zostaną uzupełnione zerami tak, aby zgadzały się rozmiarem z najdłuższym tensorem (możesz zmienić wartość domyślną poprzez wyznaczenie argumentu `default_value`):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[ 67,   97, 102, 233,   0,   0],
       [ 67, 111, 102, 102, 101, 101],
       [ 99,   97, 102, 102, 232,   0],
       [21654, 21857,   0,   0,   0]], dtype=int32)>
```

Wiele operacji TF obsługuje tensory nierówne. Pełną ich listę znajdziesz w dokumentacji klasy `tf.RaggedTensor`.

Tensory rzadkie

Moduł TensorFlow równie skutecznie reprezentuje **tensory rzadkie** (czyli zawierające głównie zera). Wystarczy utworzyć obiekt `tf.SparseTensor`, wyznaczyć indeksy i wartości elementów niezeroowych oraz określić wymiary tensora. Indeksy muszą zostać ułożone w kolejności odczytywania (od lewej do prawej, z góry na dół). Jeżeli nie masz pewności, czy dobrze je ułożyłaś/ułożyłeś, skorzystaj z funkcji `tf.sparse.reorder()`. Możesz przekształcić tensor rzadki w gęsty (standardowy) za pomocą funkcji `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                         values=[1., 2., 3.],
                         dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Zwróć uwagę, że wektory rzadkie nie obsługują tylu operacji co ich gęste odpowiedniki. Możesz na przykład pomnożyć tensor rzadki przez dowolną wartość skalarną i otrzymasz nowy tensor rzadki, ale nie możesz dodać skalara do tensora rzadkiego, ponieważ otrzymany tensor nie byłby już rzadki:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tablice tensorowe

Obiekt `tf.TensorArray` reprezentuje listę tensorów. Przydaje się on w modelach zawierających pętle; zbierane są w nim rezultaty, z których są obliczane statystyki. Możesz odczytywać albo zapisywać tensory w dowolnym miejscu tablicy:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) #=> Zwraca (i usuwa!) tf.constant([3., 10.])
```

Zwróć uwagę, że odczytywany element zostaje usunięty z tablicy i zastąpiony tensorem o takim samym rozmiarze wypełnionym zerami.



Podczas zapisywania danych w tablicy musisz przypisywać do niej otrzymywane wyniki tak, jak pokazałem w powyższym listingu. Jeżeli tego nie zrobisz, kod będzie dobrze działał w trybie pośpiesznym, ale już nie w trybie grafowym (obydwia tryby zostały opisane w rozdziale 12.).

Podczas tworzenia obiektu `TensorArray` musisz podać jego rozmiar (`size`), chyba że korzystasz z trybu grafowego. Możesz ewentualnie zignorować parametr `size`, a zamiast niego wyznaczyć `dynamic_size=True`, ale zmniejszysz w ten sposób wydajność, dlatego jeżeli od początku znasz rozmiar tablicy, zdefiniuj go. Musisz także określić typ danych (`dtype`) i wszystkie elementy muszą mieć takie same wymiary jak pierwszy element zapisany do tablicy.

Możesz umieścić wszystkie elementy w tensorze poprzez wywołanie metody `stack()`:

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Zbiory

Moduł TensorFlow obsługuje zbiory łańcuchów znaków lub liczb stałoprzecinkowych (ale nie zmiennoprzecinkowych). Reprezentuje je za pomocą standardowych tensorów. Na przykład zbiór $\{1, 5, 9\}$ jest reprezentowany po prostu jako tensor $[[1, 5, 9]]$. Zwróć uwagę, że tensor musi mieć co najmniej dwa wymiary, a zbiory muszą mieścić się w ostatnim wymiarze. Na przykład tensor $[[1, 5, 9], [2, 5, 11]]$ przechowuje dwa niezależne zbiory: $\{1, 5, 9\}$ i $\{2, 5, 11\}$. Jeżeli jakieś zbiory są krótsze od innych, należy je uzupełnić zerami (możesz w razie potrzeby zdefiniować inne wartości uzupełniające).

Pakiet `tf.sets` zawiera kilka funkcji służących do operowania na zbiorach. Utwórzmy dwa zbiory i obliczmy ich sumę (w rezultacie otrzymamy wektor rzadki, dlatego wyświetlamy go za pomocą wywołania metody `to_dense()`):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
```

```
>>> u  
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>  
>>> tf.sparse.to_dense(u)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

Możesz także obliczać jednocześnie sumę wielu par zbiorów:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])  
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0]])  
>>> u = tf.sets.union(a, b)  
>>> tf.sparse.to_dense(u)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],  
   [ 0, 10, 13,  0,  0]], dtype=int32)>
```

Jeżeli wolisz korzystać z innych wartości uzupełniających, musisz wyznaczyć `default_value` podczas wywoływania metody `to_dense()`:

```
>>> tf.sparse.to_dense(u, default_value=-1)  
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],  
   [ 0, 10, 13, -1, -1]], dtype=int32)>
```



Domyślną wartością w `default_value` jest 0, dlatego w przypadku zbiorów znakowych trzeba ją odpowiednio zmienić (np. na pusty łańcuch znaków).

Dostępne są tu również inne funkcje, takie jak `difference()`, `intersection()` czy `size()`. Jeżeli chcesz sprawdzić, czy zbiór zawiera jakieś określone wartości, możesz obliczyć część wspólną (`intersection()`) pomiędzy zbiorem a tymi wartościami. Aby dodać jakieś wartości do zbioru, wystarczy obliczyć sumę tego zbioru i dołączanych wartości.

Kolejki

Kolejka jest strukturą danych, w której możesz umieszczać rekordy danych, a później je z niej wydobywać. Moduł TensorFlow implementuje kilka typów kolejek w pakiecie `tf.queue`. Niegdyś stanowiły one bardzo ważną część implementowania wydajnych potoków wczytywania i przetwarzania wstępnego danych, ale pojawienie się interfejsu `tf.data` istotnie zmniejszyło ich znaczenie (nie licząc, być może, jakichś rzadkich sytuacji), ponieważ jest on dużo prostszy w użyciu i zawiera narzędzia bardzo ułatwiające tworzenie potoków. Jednak dla formalności przyjrzyjmy się również tej strukturze danych.

Najprostszym rodzajem jest kolejka FIFO (ang. *First In, First Out* — pierwszy na wejściu, pierwszy na wyjściu). Aby ją utworzyć, musimy określić maksymalną liczbę przechowywanych rekordów. Ponadto każdy rekord stanowi krotkę tensorów, dlatego musimy wyznaczyć typ poszczególnych tensorów oraz ewentualnie ich wymiary. Na przykład za pomocą poniższego listingu tworzymy kolejkę FIFO przechowującą maksymalnie trzy rekordy, z których każdy składa się z krotki w postaci tensora `int32` i tensora znakowego, a następnie wstawiamy do kolejki dwa rekordy, sprawdzamy jej rozmiar (w tym momencie jest równy 2) i wydobywamy z niej rekord:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])  
>>> q.enqueue([10, b"wietrznie"])  
>>> q.enqueue([15, b"bezwietrznie"])  
>>> q.size()
```

```
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'wietrznie'>]
```

Możliwe jest również umieszczanie w kolejce i wydobywanie z niej wielu rekordów jednocześnie (w tym drugim przypadku musimy określić wymiary podczas tworzenia kolejki):

```
>>> q.enqueue_many([[13, 16], [b'pochmurno', b'deszczowo']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'bezwietrznie', b'pochmurno', b'deszczowo'], dtype=object)>]
```

Dostępne są także inne rodzaje kolejek:

PaddingFIFOQueue

To samo co kolejka FIFOQueue, ale jej metoda `dequeue_many()` obsługuje wydobywanie z kolejki wielu rekordów o różnych wymiarach. Automatycznie uzupełnia zerami najkrótsze rekordy, dzięki czemu wszystkie rekordy w grupie mają taki sam rozmiar.

PriorityQueue

Kolejka, z której rekordy są wydobywane według priorytetu. Priorytet ten musi być 64-bitową wartością stałoprzecinkową umieszczoną jako pierwszy element w każdym rekordzie. Co ciekawe, rekordy o mniejszym priorytecie będą usuwane z kolejki jako pierwsze. Rekordy o takim samym priorytecie są usuwane zgodnie z kolejnością FIFO.

RandomShuffleQueue

Kolejka, której rekordy są wydobywane w kolejności losowej. Przed wprowadzeniem interfejsu `tf.data` przydawała się do implementowania bufora tasowania.

Jeżeli kolejka jest już pełna i spróbujesz dodać kolejny rekord, metoda `enqueue*()` zostanie zamrożona aż do momentu usunięcia rekordu przez inny wątek. I podobnie w przypadku próby usunięcia rekordu z pustej kolejki metoda `dequeue*()` zostanie zamrożona, dopóki jakieś rekordy nie zostaną wstawione do kolejki przez inny wątek.

Grafy TensorFlow

W tym dodatku zajmiemy się analizą grafów generowanych przez funkcje TF (zob. rozdział 12.).

Funkcje TF i funkcje konkretne

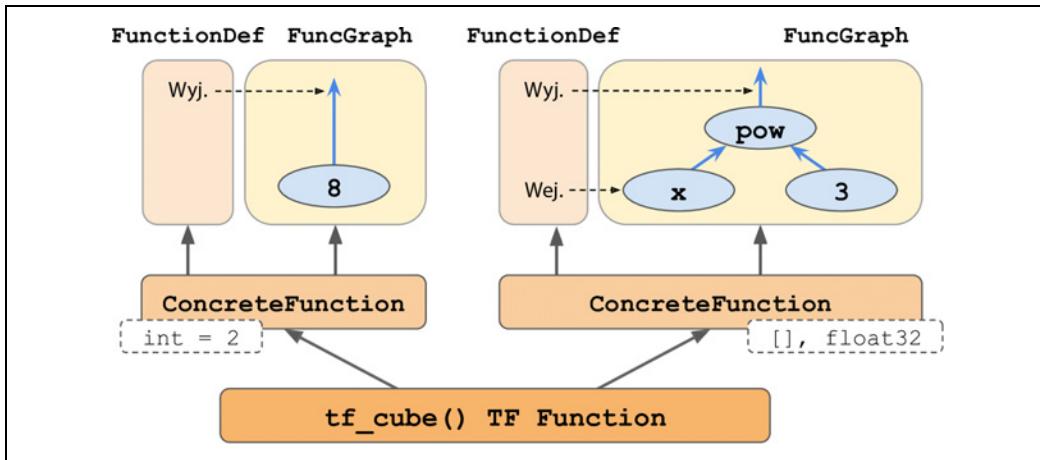
Funkcje TF są polimorficzne, co oznacza, że obsługują dane wejściowe o różnych typach (i wymiarach). Na przykład spójrz na poniższą funkcję `tf_cube()`:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Za każdym razem, gdy wywołujesz funkcję TF z nową kombinacją typów lub wymiarów danych wejściowych, zostaje utworzona nowa **funkcja konkretna** (ang. *concrete function*), zawierająca własny graf wyspecjalizowany pod względem tejże kombinacji. Taka kombinacja typów i wymiarów argumentów to tak zwana **sygnatura wejściowa** (ang. *input signature*). Jeżeli wywołasz funkcję TF z użytą już wcześniej sygnaturą wejściową, zostanie wykorzystana utworzona uprzednio funkcja konkretna. Na przykład jeśli wywołasz `tf_cube(tf.constant(3.0))`, funkcja TF skorzysta z tej samej funkcji konkretnej, która posłużyła do obliczenia `tf_cube(tf.constant(2.0))` (w przypadku tensorów skalarnych typu `float32`). Jeśli jednak wywołasz funkcję `tf_cube(tf.constant([2.0]))` lub `tf_cube(tf.constant([3.0]))` (tensory typu `float32` o wymiarze `[1]`), zostanie wygenerowana nowa funkcja konkretna, a jeszcze inną dla `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (tensory typu `float32` o wymiarach `[2, 2]`). Możesz uzyskać funkcję konkretną dla określonej kombinacji danych wejściowych poprzez wywołanie metody `get_concrete_function()` funkcji TF. Można ją następnie wywołać jak standardową funkcję, ale będzie obsługiwała tylko jedną sygnaturę wejściową (w omawianym przykładzie tensorze skalarny typu `float32`):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

Rysunek G.1 pokazuje funkcję TF `tf_cube()` po wywołaniu funkcji `tf_cube(2)` i `tf_cube →(tf.constant(2.0))`: zostały wygenerowane dwie funkcje konkretne, po jednej dla każdej sygnatury i każda z nich zawiera własny, zoptymalizowany **graf funkcji** (`FuncGraph`) i **definicję funkcji** (`FunctionDef`). Definicja funkcji wyznacza obszary grafu odpowiadające wejściom i wyjściom



Rysunek G.1. Funkcja TF `tf_cube()` zawierająca bloki `ConcreteFunction` i ich grafy funkcji

funkcji. W każdym bloku `FunGraph` węzły (elipsy) symbolizują operacje (np. potęgowanie, stałe lub węzły zastępcze dla argumentów takich jak x itd.), natomiast krawędzie (strzałki ciągle pomiędzy operacjami) to tensory przechodzące przez graf. Widoczna po lewej funkcja konkretna jest wyspecjalizowana dla $x = 2$, zatem moduł TensorFlow uprościł ją i za każdym razem uzyskuje na jej wyjściu wartość 8 (zwróć uwagę, że definicja funkcji nie zawiera nawet wejścia). Z kolei funkcja konkretna widoczna po prawej stronie jest przeznaczona dla tensorów skalarnych typu `float32`, zatem nie można jej uprościć. Jeżeli wywołamy `tf_cube(tf.constant(5.0))`, nastąpi wywołanie drugiej funkcji konkretnej, operacja zastępca dla x wygeneruje wartość 5,0, po czym operacja potęgowania obliczy $5,0^3$, dzięki czemu uzyskamy wynik 125,0.

Tensory w tych grafach są **symboliczne**, co oznacza, że nie przechowują rzeczywistych wartości, lecz jedynie typ, wymiary i nazwę danych. Reprezentują one przyszłe tensory, które będą przepływały przez graf po umieszczeniu rzeczywistej wartości w węźle zastępczym x i po uruchomieniu grafu. Dzięki tensorom symbolicznym możemy zawsze określić mechanizmy łączenia operacji, a także umożliwiają one modułowi TensorFlow rekurencyjne wnioskowanie typów i wymiarów wszystkich tensorów, jeżeli znane są typy i wymiary danych na wejściach.

Zajrzyjmy teraz trochę głębiej i sprawdźmy, jak można uzyskać dostęp do definicji i grafów funkcji oraz jak można analizować grafy i tensory grafu.

Analizowanie grafów i definicji funkcji

Możesz uzyskać dostęp do grafu obliczeniowego funkcji konkretnej za pomocą atrybutu `graph` i użyć listę tworzących go operacji poprzez wywołanie jego metody `get_operations()`:

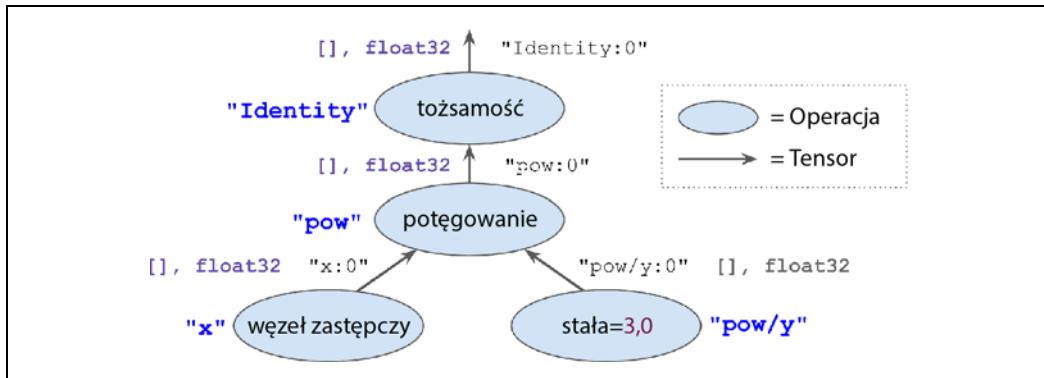
```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
```

```
<tf.Operation 'pow/y' type=Const>
<tf.Operation 'pow' type=Pow>
<tf.Operation 'Identity' type=Identity>]
```

W tym przykładzie pierwsza operacja reprezentuje argument wejściowy x (jest to tzw. **węzeł zastępczy** — ang. *placeholder*), druga „operacja” symbolizuje stałą 3, trzecia operacja to potęgowanie ($**$), natomiast w ostatniej operacji definiujemy wynik funkcji (mamy tu do czynienia z operacją identyczności, co oznacza, że zostaje jedynie skopiowany wynik operacji dodawania¹). Każda operacja zawiera listę tensorów wejściowych i wyjściowych, do której możesz z łatwością uzyskać dostęp za pomocą atrybutów, odpowiednio, `inputs` i `outputs`. Na przykład sprawdźmy wejścia i wyjścia operacji potęgowania:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

Powyższy graf obliczeniowy został zaprezentowany na rysunku G.2.



Rysunek G.2. Przykład grafu obliczeniowego

Zwróć uwagę, że każda operacja ma nazwę. Domyslnie jest to nazwa wykonywanego działania (np. „`pow`”), ale możesz zdefiniować ją własnoręcznie podczas wywoływanego operacji (np. `tf.pow(x, 3, name="inna_nazwa")`). Jeżeli dana nazwa już istnieje, TensorFlow doda automatycznie niepowtarzalny indeks (np. „`pow_1`”, „`pow_2`” itd.). Również każdy tensor ma niepowtarzalną nazwę: jest to zawsze nazwa operacji generującej ten tensor, a także :0, jeżeli jest to pierwszy wynik operacji, :1 w przypadku drugiego wyniku itd. Możesz wyszukać operację lub tensor poprzez nazwę za pomocą wywołania metod grafu `get_operation_by_name()` lub `get_tensor_by_name()`:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

¹ Możesz spokojnie ignorować tę operację, gdyż została ona umieszczona jedynie ze względów technicznych, po to, aby zapobiegać „wyciekaniu” struktur wewnętrznych funkcji TF.

Funkcja konkretna zawiera również definicję funkcji (reprezentowaną w postaci bufora protokołów²), zawierającą sygnaturę funkcji. Dzięki tej sygnaturze funkcja konkretna wstawia do węzłów zastępczych odpowiednie wartości wejściowe i zwraca odpowiednie tensory:

```
>>> concrete_function.function_def.signature
name: "_inference_cube_19068241"
input_arg {
    name: "x"
    type: DT_FLOAT
}
output_arg {
    name: "identity"
    type: DT_FLOAT
}
```

Przyjrzyjmy się teraz dokładniej śledzeniu (ang. *tracing*).

Blższe spojrzenie na śledzenie

Zmodyfikujmy funkcję `tf_cube()` tak, aby były wyświetlane jej dane wejściowe:

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Wywołajmy ją:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

Obiekt `result` wygląda dobrze, ale spójrz, co zostało wyświetlone: `x` jest tensorem symbolicznym! Ma on zdefiniowany typ i wymiary danych, ale brakuje wartości. Do tego ma wyznaczoną nazwę ("`x:0`"). Wynika to z faktu, że funkcja `print()` nie jest operacją TensorFlow, zatem będzie realizowana jedynie wtedy, gdy funkcja Pythona będzie śledzona, czyli w trybie grafowym, gdzie argumenty zostają zastąpione tensorami symbolicznymi (o takim samym typie i wymiarach, ale bez wartości). Funkcja `print()` nie została umieszczona w grafie, dlatego przy kolejnych wywołaniach funkcji `tf_cube()` dla tensorów skalarnych typu `float32` nic nie będzie wyświetlane:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

Jeżeli jednak wywołamy funkcję `tf_cube()` z tensorem o innym typie danych lub wymiarach albo z nową wartością Pythona, zacznie być ponownie śledzona i dlatego zostanie wywołana funkcja `print()`:

```
>>> result = tf_cube(2) # Nowa wartość Pythona: śledź!
x = 2
>>> result = tf_cube(3) # Nowa wartość Pythona: śledź!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # Nowe wymiary: śledź!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
```

² Popularny format binarny omówiony w rozdziale 13.

```
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # Nowe wymiary: śledź!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Stare wymiary: nie śledź
```



Jeżeli Twoja funkcja ma jakieś pythonowe „skutki uboczne” (np. zapisuje jakieś zdarzenia na dysku), to pamiętaj, że kod będzie realizowany jedynie wtedy, gdy ta funkcja będzie śledzona (tzn. za każdym razem, kiedy funkcja TF jest wywoływana z nową sygnaturą wejściową). Najlepiej założyć, że funkcja ta może być śledzona (lub nie), ilekroć zostaje wywołana funkcja TF.

W pewnych przypadkach możesz chcieć ograniczyć funkcję TF do określonej sygnatury wejściowej. Założmy na przykład, że będziesz zawsze wywoływać funkcję TF wyłącznie dla grup obrazów o rozmiarach 28×28 , ale same grupy mogą znacznie różnić się rozmiarami. Możesz nie chcieć, aby moduł TensorFlow generował oddzielną funkcję konkretną dla poszczególnych rozmiarów grup, albo nie chcesz, żeby sam decydował o użyciu wartości `None`. W takim przypadku możesz zdefiniować sygnaturę wejściową w następujący sposób:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # Porzuca połowę rzędów i kolumn
```

Ta funkcja TF będzie akceptować dowolny tensor typu `float32` o wymiarach $[*, 28, 28]$ i będzie za każdym razem korzystać z tej samej funkcji konkretnej:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # Działa właściwie. Śledzi funkcję
preprocessed_images = shrink(img_batch_2) # Działa właściwie. Ta sama funkcja konkretna
```

Jeśli jednak próbujesz wywołać tę funkcję TF z jakąś wartością Pythona albo tensorem o nieoczekiwany typie lub wymiarach danych, zostanie wyświetlony komunikat o wyjątku:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3) # ValueError! Nieoczekiwana sygnatura
```

Analiza przepływu sterowania za pomocą narzędzia AutoGraph

Jeżeli Twoja funkcja zawiera prostą pętlę `for`, to jak myślisz, co się stanie? Utwórzmy na przykład funkcję dodającą 10 do danych wejściowych poprzez zwiększenie ich wartości o 1 przez dziesięć przebiegów:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

Algorytm działa tak, jak należy, ale jeżeli spojrzymy na graf, to przekonamy się, że nie zawiera on pętli: znajdziemy tu po prostu dziesięć operacji dodawania!

```
>>> add_10(tf.constant(0))
```

```
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...]
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

W istocie ma to sens: gdy rozpoczął się proces śledzenia funkcji, pętla wykonała dziesięć przebiegów, zatem operacja `x += 1` została przeprowadzona dziesięciokrotnie, a ponieważ znajdowała się w trybie grafowym, została zarejestrowana dziesięć razy w grafie. Możemy traktować taką pętlę `for` jako pętlę „statyczną”, która jest rozwijana w miarę tworzenia grafu.

Jeżeli wolisz skorzystać z pętli „dynamicznej” (realizowanej po uruchomieniu grafu), możesz ją utworzyć własnoręcznie za pomocą operacji `tf.while_loop()`, ale nie jest to zbyt intuicyjne rozwiązanie (przykład znajdziesz w sekcji „Analiza przepływu sterowania za pomocą narzędzia AutoGraph” w notatniku Jupyter dołączonym do rozdziału 12.). Zamiast tego znacznie łatwiej jest użyć omówionego w rozdziale 12. narzędzia **AutoGraph**, stanowiącego część modułu TensorFlow. W rzeczywistości narzędzie AutoGraph jest domyślnie uaktywnione (jeżeli będziesz musiał/musiało kiedyś je wyłączyć, możesz przekazać `autograph=False` do funkcji `tf.function()`). Skoro więc jest domyślnie włączone, dlaczego nie zajmuje się pętlą `for` w funkcji `add_10()`? Wychwytuje ono pętle `for` wykorzystywane jedynie w ramach funkcji `tf.range()`, a nie `range()`. Otrzymujesz w ten sposób wybór:

- Jeżeli skorzystasz z funkcji `range()`, pętla `for` pozostanie statyczna, co oznacza, że będzie realizowana jedynie w przypadku śledzonej funkcji. Jak już wiemy, pętla zostanie „rozwinęta” w zbiór operacji dla każdej iteracji.
- Jeżeli skorzystasz z funkcji `tf.range()`, pętla będzie dynamiczna, co oznacza, że będzie stanowić część samego grafu (ale nie będzie realizowana podczas śledzenia).

Spójrzmy na graf, który zostanie wygenerowany po zastąpieniu funkcji `range()` funkcją `tf.range()` wewnętrz funkcji `add_10()`:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

Jak widać, graf zawiera teraz operację pętli `While`, zupełnie jakbyśmy wywołali funkcję `tf.while_loop()`.

Obsługa zmiennych i innych zasobów w funkcjach TF

W module TensorFlow zmienne i inne obiekty stanowe, takie jak kolejki czy zestawy danych, są nazywane **zasobami** (ang. *resources*). Funkcje TF traktują je ze szczególną troską: każda operacja odczytująca lub aktualizująca zasób jest uznawana za stanową, a funkcje TF gwarantują, że operacje stanowe są przetwarzane w kolejności występowania (w przeciwieństwie do operacji bezstanowych, które mogą być realizowane równolegle, zatem mogą nie być przetwarzane zgodnie z kolejnością). Ponadto przekazywanie zasobu jako argumentu do funkcji TF następuje za pomocą odwołania, dzięki czemu może on być modyfikowany. Na przykład:

```

counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter jest teraz równy 1
increment(counter) # counter jest teraz równy 2

```

Jeśli spojrzesz na definicję funkcji, zauważysz, że pierwszy argument jest oznaczony jako zasób:

```

>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE

```

Możliwe jest także korzystanie z zasobu `tf.Variable` zdefiniowanego poza funkcją, bez konieczności jego jawnego przekazywania jako argumentu:

```

counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)

```

Funkcja TF będzie traktować go jako niejawny pierwszy argument, dlatego ostatecznie uzyskamy tę samą sygnaturę (nie licząc nazwy argumentu). Jednak korzystanie ze zmiennych globalnych szybko może wywołać bałagan, dlatego zazwyczaj należy umieszczać zmienne (i inne zasoby) wewnętrz klas. Dobra wiadomość jest taka, że `@tf.function` współdziała również z metodami:

```

class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)

```



W przypadku funkcji TF nie używaj operatorów `=`, `+=`, `-=` ani żadnego innego pythonowego operatora przypisania. Zamiast nich musisz korzystać z metod `assign()`, `assign_add()` lub `assign_sub()`. Jeżeli wprowadzisz pythonowy operator przypisania, podczas wywołania metody zostanie wyświetlony komunikat o wyjątku.

Dobrym przykładem takiej strategii obiektowej jest, rzecz jasna, interfejs `tf.keras`. Sprawdźmy, jak można korzystać z funkcji TF wraz z interfejsem `tf.keras`.

Korzystanie (lub niekorzystanie) z funkcji TF wraz z interfejsem `tf.keras`

Domyślnie każde niestandardowe warstwa, funkcja lub model używane w interfejsie `tf.keras` będą automatycznie przekształcane w funkcję TF — nie musisz nic robić! W pewnych przypadkach jednak należy wyłączyć tę funkcję automatycznej konwersji, na przykład wtedy, gdy niestandardowego

kodu nie da się przekształcić w funkcję TF albo gdy po prostu chcesz wychwytywać błędy w kodzie, co jest znacznie łatwiejsze w trybie pośpiesznym. W tym celu wystarczy przekazać `dynamic=True` podczas tworzenia modelu lub którejkolwiek z jego warstw:

```
model = MyModel(dynamic=True)
```

Jeżeli niestandardowe model/warstwa będą zawsze dynamiczne, możesz skorzystać z innego rozwiązania i wywołać konstruktor klasy bazowej z argumentem `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Ewentualnie możesz po prostu przekazać `run_eagerly=True` podczas wywoływania metody `compile()`:

```
model.compile(loss=my_mse, optimizer="adam", metrics=[my_mae],
               run_eagerly=True)
```

Teraz już wiesz, jak funkcje TF obsługują polimorfizm (z wieloma funkcjami konkretnymi), jak są automatycznie generowane grafy za pomocą narzędzia AutoGraph i śledzenia, jak wyglądają grafy, jak analizować operacje i tensory symboliczne, jak obsługiwac zmienne i zasoby, a także jak korzystać z funkcji TF w interfejsie `tf.keras`.

Informacje o autorze

Aurélien Géron jest konsultantem do spraw uczenia maszynowego i wykładowcą. W latach 2013 – 2016 jako pracownik firmy Google prowadził zespół zajmujący się klasyfikowaniem filmów w serwisie YouTube. Jest również założycielem i dyrektorem ds. technicznych w kilku firmach: Wifirst (wiodący dostawca bezprzewodowych usług internetowych we Francji), Polyconseil (biuro doradcze w zakresie telekomunikacji, środków masowego przekazu i strategii) oraz Kiwisoft (biuro doradcze w zakresie uczenia maszynowego i ochrony danych).

Autor książki, zanim stał się inżynierem, pracował w wielu branżach: finansowej (JP Morgan i Société Générale), obronnej (kanadyjskie Ministerstwo Obrony) i medycznej (transfuzje krwi). Napisał kilka książek o zagadnieniach technicznych (o języku C++, sieciach WiFi oraz architekturach sieci internetowych), a także prowadził zajęcia z informatyki na francuskiej politechnice.

Kilka ciekawostek z życia: nauczył trójkę swoich dzieci liczenia w systemie dwójkowym za pomocą palców (aż do liczby 1023), studiował mikrobiologię i genetykę ewolucyjną, a w czasie drugiego skoku nie otworzył się mu spadochron.

Kolofon

Widocznym na okładce zwierzęciem jest salamandra plamista (*Salamandra salamandra*), płaz występujący na większości terytorium Europy. Są to jaszczurki mające czarne ciało upstrzone dużymi żółtymi plamami, które ostrzegają przed obecnością toksycznych alkaloidów w ich organizmach. Prawdopodobnie stąd pochodzi druga nazwa tego zwierzęcia, jaszczur ognisty, wszak kontakt z tymi toksynami (które mogą być również rozprzestrzeniane na niewielkie odległości) wywołuje konwulsje i hiperwentylację. Obecność toksyn wywołujących ból lub znaczna wilgotność skóry (a może obie te cechy naraz) doprowadziły do błędnego przekonania, że zwierzęta te potrafią nie tylko przeżyć w płomieniach, lecz wręcz je gasić.

Salamandry plamiste żyją w zacienionych lasach, ukrywając się w wilgotnych szczelinach i pod pniami w pobliżu stawów lub innych zbiorników wodnych, które stanowią ich naturalne środowisko lęgowe. Większość życia spędzały na lądzie, ale rodzą młode w wodzie. Źródło ich pożywienia stanowią głównie owady, pajęczaki, ślimaki i inne płazy. Mogą osiągnąć długość do 32 cm, a w niewoli dożywają nawet 50 lat.

Liczebność tego gatunku systematycznie maleje w wyniku niszczenia ich siedlisk leśnych i wyłapywania na rynek hodowlany, jednak największe zagrożenie stanowi dla nich wrażliwość przepuszczalnej skóry na zanieczyszczenia i drobnoustroje. Od 2014 roku uznaje się, że salamandry plamiste wygineały w niektórych obszarach Holandii i Belgii wskutek wprowadzenia nowego gatunku grzybów.

Wiele gatunków zwierząt umieszczanych na okładkach wydawnictwa O'Reilly jest zagrożonych; każdy z nich stanowi wielki skarb dla świata.

Autorką rysunku na okładce jest Karen Montgomery, która wzorowała się na szkicu z książki *Wood's Illustrated Natural History*.

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
<http://program-partnerski.helion.pl>

KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ

BLENDDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie z dostępem do nowoczesnych narzędzi - videokursów, e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL