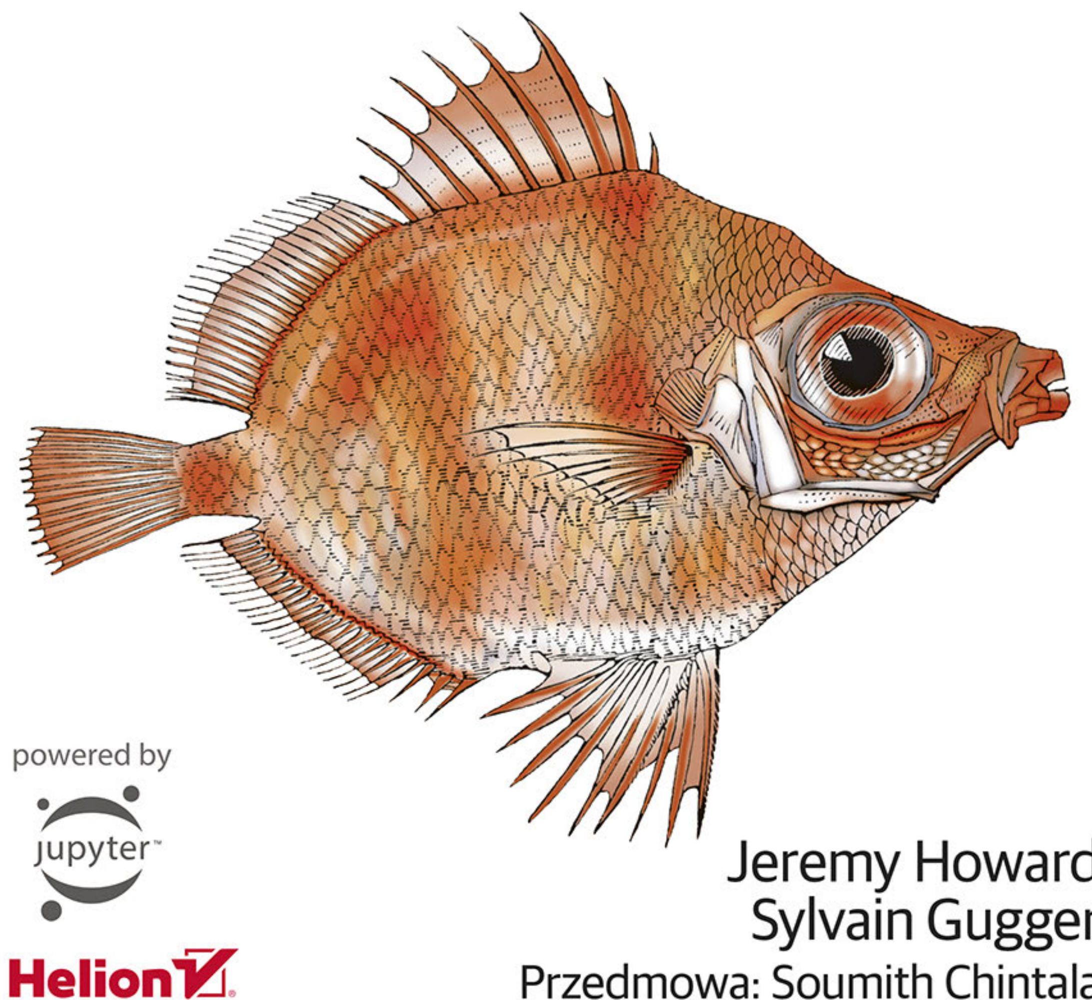
O'REILLY

Deep learning dla programistów

Budowanie aplikacji AI za pomocą fastai i PyTorch



powered by

jupyter

Jeremy Howard
Sylvain Gugger

Przedmowa: Soumith Chintala

Helion

Tytuł oryginału: Deep Learning for Coders with fastai and PyTorch: AI Applications Without a PhD

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-7509-3

© 2021 Helion S.A.
Authorized Polish translation of the English edition of Deep Learning for Coders with fastai and PyTorch ISBN 9781492045526 © 2020 Jeremy Howard and Sylvain Gugger

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 właścicieli.

Autor oraz wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion S.A.
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/delepr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 17

Przedmowa 21

Część I. Uczenie głębokie w praktyce 23

1. Podróż po świecie uczenia głębokiego 25
Uczenie głębokie jest dla każdego 25
Sieci neuronowe — krótka historia 27
Kim jesteśmy? 29
Jak zdobyć wiedzę o uczeniu głębokim? 31
Twoje projekty i Twój sposób myślenia 33
Oprogramowanie: PyTorch, fastai i Jupyter (i dlaczego nie ma to znaczenia) 34
Twój pierwszy model 35
Uzyskanie dostępu do serwera z procesorem graficznym i możliwością realizowania uczenia głębokiego 36
Uruchomienie pierwszego notatnika 37
Co to jest uczenie maszynowe? 41
Co to jest sieć neuronowa? 44
Trochę słownictwa związanego z uczeniem głębokim 46
Ograniczenia związane z uczeniem maszynowym 46
Jak działa nasz program do rozpoznawania obrazów 48
Czego nauczył się program do rozpoznawania obrazów? 54
Systemy do rozpoznawania obrazów mogą radzić sobie z zadaniami innymi niż analiza obrazów 57
Podsumowanie słownictwa 60
Uczenie głębokie to nie tylko klasyfikowanie obrazów 62

Kup ksi k

Pole ksi k

Zbiory walidacyjne i testowe 68
Użycie oceny w definiowaniu zbiorów testowych 70
Moment, w którym wybierasz swoją własną przygodę 73
Pytania 74
Dalsze badania 75

2. Od modelu do produkcji 77
Praktyczne zastosowanie uczenia głębokiego 77
Rozpoczęcie projektu 78
Stan uczenia głębokiego 79
Metoda układu napędowego 83
Gromadzenie danych 85
Od danych do obiektu DataLoaders 89
Generowanie sztucznych danych 92
Trenowanie modelu i używanie go do czyszczenia danych 93
Przekształcanie modelu w aplikację internetową 96
Korzystanie z modelu do wnioskowania 97
Tworzenie w notatniku aplikacji na podstawie modelu 98
Zamień notatnik w prawdziwą aplikację 100
Wdrażanie aplikacji 101
Jak uniknąć katastrofy 104
Nieprzewidziane konsekwencje i pętle sprzężenia zwrotnego 106
Zapisuj! 107
Pytania 108
Dalsze badania 109

3. Etyka danych 111
Kluczowe przykłady etyki danych 112
Błędy i regresja: wadliwy algorytm używany do świadczeń opieki zdrowotnej 113
Pętle sprzężenia zwrotnego: system rekomendacji YouTube 113
Uprzedzenie: wykładowca Latanya Sweeney „aresztowana” 113
Dlaczego ma to znaczenie? 114
Integracja uczenia maszynowego z projektowaniem produktu 117
Zagadnienia związane z etyką danych 119
Regres i odpowiedzialność 119
Pętle sprzężenia zwrotnego 120
Uprzedzenie 123
Dezinformacja 133
Identyfikowanie i rozwiązywanie problemów etycznych 134
Przeanalizuj projekt, nad którym pracujesz 135
Procesy do zaimplementowania 136

Kup ksi k Pole ksi k

Potęga różnorodności 137
Uczciwość, odpowiedzialność i przejrzystość 139
Rola polityki 140
Skuteczność przepisów 140
Prawa i polityka 141
Samochody — historyczny precedens 142
Wnioski 142
Pytania 143
Dalsze badania 144
Uczenie głębokie w praktyce — to wszystko! 145

Część II. Zrozumienie aplikacji fastai 147

4. Jak to wygląda od środka — trenowanie klasyfikatora cyfr 149
Piksele — podstawa widzenia komputerowego 149
Podejście pierwsze: podobieństwo pikseli 153
Tablice NumPy i tensory PyTorch 158
Wyznaczanie wskaźników z wykorzystaniem rozgłaszania 160
Stochastyczny spadek wzdłuż gradientu 163
Wyznaczanie gradientów 167
Stopniowanie ze współczynnikiem uczenia 169
Kompleksowy przykład użycia stochastycznego spadku wzdłuż gradientu 171
Podsumowanie procesu stochastycznego spadku wzdłuż gradientu 176
Funkcja straty MNIST 177
Sigmoida 181
Stochastyczny spadek wzdłuż gradientu i minipaczki 182
Złożenie wszystkiego w całość 184
Tworzenie optymalizatora 186
Wprowadzanie nieliniowości 188
Bardziej rozbudowane modele 191
Podsumowanie słownictwa 192
Pytania 194
Dalsze badania 195

5. Klasyfikowanie obrazów 197
Od psów i kotów do ras zwierząt domowych 197
Dobór wstępny 200
Sprawdzanie i debugowanie obiektu DataBlock 202
Entropia krzyżowa 205
Przeglądanie aktywacji i etykiet 205
Softmax 206

Kup ksi k Pole ksi k

Logarytm prawdopodobieństwa 209
Obliczanie logarytmu 210
Interpretacja modelu 212
Poprawianie modelu 214
Wyszukiwarka współczynnika uczenia 214
Odmrażanie i uczenie transferowe 216
Dyskryminatywne współczynniki uczenia 218
Wybór liczby epok 221
Bardziej złożone architektury 221
Podsumowanie 223
Pytania 224
Dalsze badania 224

6. Inne zagadnienia związane z widzeniem komputerowym 225
Klasyfikacja wieloetykietowa 225
Dane 226
Tworzenie obiektu DataBlock 228
Binarna entropia krzyżowa 231
Regresja 235
Gromadzenie danych 236
Trenowanie modelu 238
Podsumowanie 241
Pytania 241
Dalsze badania 242

7. Trenowanie supernowoczesnego modelu 243
Imagenette 243
Normalizacja 245
Progresywna zmiana rozmiaru 246
Wydłużenie czasu testu 248
Mixup 249
Wygładzanie etykiet 252
Podsumowanie 254
Pytania 254
Dalsze badania 255

8. Szczegółowa analiza filtrowania zespołowego 257
Pierwszy kontakt z danymi 258
Czynniki ukryte 260
Tworzenie obiektu DataLoaders 261

Kup ksi k   Pole ksi k

Filtrowanie zespołowe od podstaw 263
Wygaszanie wag 266
Tworzenie własnego modułu osadzania 268
Interpretacja osadzeń i przesunięć 269
Użycie aplikacji fastai.collab 270
Odległość osadzania 271
Uruchamianie modelu filtrowania zespołowego 272
Uczenie głębokie w filtrowaniu zespołowym 273
Podsumowanie 275
Pytania 276
Dalsze badania 277

9. Szczegółowa analiza modelowania tabelarycznego 279
Osadzenia skategoryzowane 279
Poza uczeniem głębokim 284
Zbiór danych 285
Konkursy Kaggle 285
Sprawdzenie danych 286
Drzewa decyzyjne 288
Obsługa dat 290
Użycie obiektów TabularPandas i TabularProc 290
Tworzenie drzewa decyzyjnego 292
Zmienne skategoryzowane 296
Lasy losowe 297
Tworzenie lasu losowego 298
Błąd out-of-bag 300
Interpretacja modelu 301
Wariancja drzewa dla pewności prognozy 301
Ważności cech 302
Usuwanie zmiennych o niskiej ważności 303
Usuwanie zbędnych cech 304
Częściowa zależność 305
Wyciek danych 308
Interpreter drzewa 309
Ekstrapolacja i sieci neuronowe 311
Problem ekstrapolacji 311
Wyszukiwanie danych spoza domeny 312
Użycie sieci neuronowej 314

Kup ksi k Pole ksi k

Łączenie w zespoły 317
Wzmacnianie 318
Łączenie osadzeń z innymi metodami 319
Podsumowanie 320
Pytania 321
Dalsze badania 322

10. Szczegółowa analiza przetwarzania języka naturalnego — rekurencyjne sieci neuronowe ... 323
Wstępne przetwarzanie tekstu 325
Tokenizacja 326
Tokenizacja słów przy użyciu biblioteki fastai 327
Tokenizacja podłańcuchów 329
Zamiana na liczby przy użyciu biblioteki fastai 331
Umieszczanie tekstu w paczkach dla modelu językowego 332
Trenowanie klasyfikatora tekstu 335
Użycie klasy DataBlock w modelu językowym 335
Dostrajanie modelu językowego 336
Zapisywanie i wczytywanie modeli 337
Generowanie tekstu 338
Tworzenie klasyfikatora DataLoaders 339
Dostrajanie klasyfikatora 341
Dezinformacja i modele językowe 342
Podsumowanie 344
Pytania 345
Dalsze badania 346

11. Przygotowywanie danych dla modeli za pomocą interfejsu API pośredniego poziomu z biblioteki fastai ... 347
Szczegółowa analiza warstwowego interfejsu programistycznego biblioteki fastai 347
Transformacje 348
Tworzenie własnej transformacji 349
Klasa Pipeline potoku transformacji 351
TfmdLists i Datasets — kolekcje przekształcone 351
TfmdLists 351
Datasets 353
Zastosowanie interfejsu API pośredniego poziomu — SiamesePair 355
Podsumowanie 359
Pytania 359
Dalsze badania 360
Zrozumienie aplikacji fastai — podsumowanie 360

Kup ksi k Pole ksi k

Część III. Podstawy uczenia głębokiego 361

12. Tworzenie od podstaw modelu językowego 363

Dane 363
Tworzenie od podstaw pierwszego modelu językowego 364
Obsługa modelu językowego w bibliotece PyTorch 365
Pierwsza rekurencyjna sieć neuronowa 368
Ulepszanie sieci RNN 369
Obsługa stanu sieci RNN 370
Tworzenie większej liczby sygnałów 372
Wielowarstwowe rekurencyjne sieci neuronowe 374
Model 375
Eksplodujące lub zanikające aktywacje 376
Architektura LSTM 377
Tworzenie modelu LSTM od podstaw 378
Trenowanie modelu językowego wykorzystującego architekturę LSTM 380
Regularyzacja modelu LSTM 381
Dropout 381
Regularyzacja aktywacji i czasowa regularyzacja aktywacji 383
Trening regularyzowanego modelu LSTM z wiązanymi wagami 384
Podsumowanie 385
Pytania 386
Dalsze badania 388

13. Konwolucyjne sieci neuronowe 389

Magia konwolucji 389
Odwzorowywanie jądra splotu 392
Konwolucje w bibliotece PyTorch 394
Kroki i dopełnienie 396
Zrozumienie równań konwolucji 397
Pierwsza konwolucyjna sieć neuronowa 399
Tworzenie konwolucyjnej sieci neuronowej 399
Zrozumienie arytmetyki konwolucji 402
Pola receptywne 403
Kilka uwag o Twitterze 405
Obrazy kolorowe 407
Ulepszanie stabilności trenowania 409
Prosty model bazowy 410
Zwiększenie wielkości paczki 412

Kup ksi k   Pole ksi k

Trenowanie jednocykliczne 413
Normalizacja wsadowa 418
Podsumowanie 420
Pytania 421
Dalsze badania 423

14. Sieci ResNet 425
Powrót do Imagenette 425
Tworzenie nowoczesnej konwolucyjnej sieci neuronowej — ResNet 429
Pomijanie połączeń 429
Model sieci ResNet na poziomie światowym 434
Warstwy z wąskim gardłem 437
Podsumowanie 439
Pytania 439
Dalsze badania 440

15. Szczegółowa analiza architektur aplikacji 441
Widzenie komputerowe 441
Funkcja cnn_learner 441
Funkcja unet_learner 443
Model syjamski 445
Przetwarzanie języka naturalnego 447
Dane tabelaryczne 448
Podsumowanie 449
Pytania 450
Dalsze badania 451

16. Proces trenowania 453
Tworzenie modelu bazowego 453
Ogólny optymalizator 455
Momentum 456
RMSProp 459
Adam 460
Dwie metody wygaszania wag 461
Wywołania zwrotne 461
Tworzenie wywołania zwrotnego 464
Kolejność wywołań zwrotnych i wyjątki 468
Podsumowanie 469
Pytania 469
Dalsze badania 470
Podstawy uczenia głębokiego — podsumowanie 471

Kup ksi k Pole ksi k

Część IV. Uczenie głębokie od podstaw 473

17. Sieć neuronowa od podstaw 475
Tworzenie od podstaw warstwy sieci neuronowej 475
Modelowanie neuronu 475
Mnożenie macierzy od podstaw 476
Arytmetyka składowych 477
Rozgłaszanie 479
Konwencja sumacyjna Einsteina 483
Przejścia w przód i wstecz 484
Definiowanie i inicjalizowanie warstwy 484
Gradienty i przejście wstecz 488
Modyfikowanie modelu 491
Implementacja przy użyciu biblioteki PyTorch 492
Podsumowanie 494
Pytania 495
Dalsze badania 497

18. Interpretacja sieci CNN przy użyciu mapy aktywacji klas 499
Mapa aktywacji klas i punkty zaczepienia 499
Gradientowa mapa aktywacji klas 502
Podsumowanie 504
Pytania 505
Dalsze badania 505

19. Klasa Learner biblioteki fastai od podstaw 507
Dane 507
Klasa Dataset 509
Klasy Module i Parameter 511
Prosta konwolucyjna sieć neuronowa 514
Funkcja straty 515
Klasa Learner 516
Wywołania zwrotne 518
Harmonogram modyfikowania współczynnika uczenia 519
Podsumowanie 521
Pytania 521
Dalsze badania 522

20. Uwagi końcowe 523

A Tworzenie bloga 525

B Lista kontrolna projektu dotyczącego danych 533

Kup ksi k   Pole ksi k

Kup książkę Poleć książkę

ROZDZIAŁ 13.
Konwolucyjne sieci neuronowe

W rozdziale 4. dowiedziałeś się, jak można stworzyć sieć neuronową rozpoznającą obrazy. Podczas rozróżniania cyfr „3" i „7" udało się osiągnąć nieco ponad 98-procentową dokładność. Wiemy jednak, że wbudowane klasy biblioteki *fastai* były w stanie zbliżyć się do wartości 100%. Zacznijmy od zniwelowania tej luki.

Najpierw przeanalizujemy konwolucje, a także od podstaw stworzymy sieć CNN. Następnie omówimy szereg technik poprawiających stabilność trenowania i pokażemy ulepszenia, które zazwyczaj pozwalają osiągnąć świetne wyniki.

Magia konwolucji

Jednym z najpotężniejszych narzędzi, którymi dysponują specjaliści zajmujący się w praktyce uczeniem maszynowym, jest *inżynieria cech*. *Cecha* to transformacja danych, która ma na celu ułatwienie modelowania. Na przykład funkcja `add_datepart`, której używaliśmy do wstępnego przetwarzania tabelarycznego zbioru danych w rozdziale 9., uzupełniła ten zbiór o cechy daty. Jakie rodzaje cech moglibyśmy stworzyć na podstawie obrazów?

Słownictwo: inżynieria cech

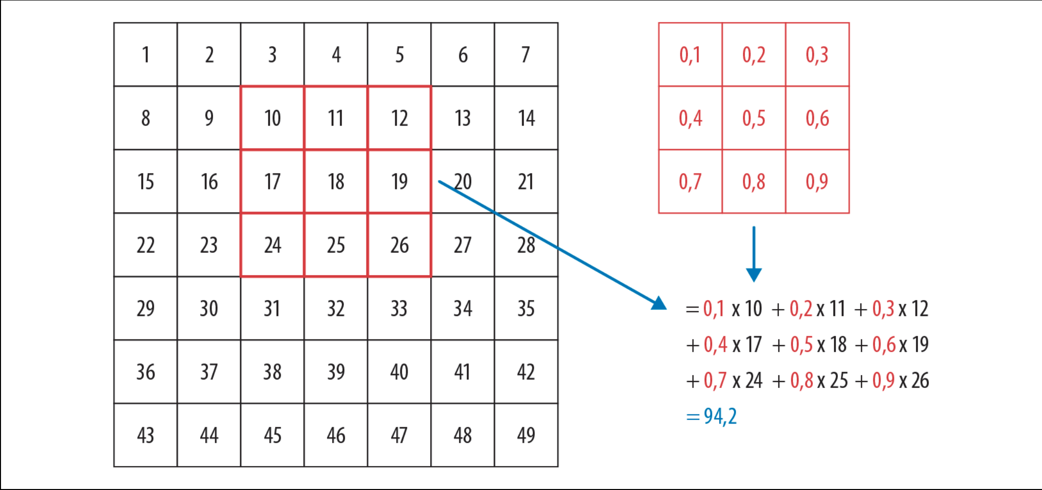
Tworzenie transformacji danych wejściowych w celu ułatwienia modelowania.

W przypadku obrazów cecha jest wizualnie wyróżniającym się atrybutem. Na przykład liczba „7" charakteryzuje się poziomą krawędzią w górnej części znaku i ukośną linią prowadzącą od prawego górnego do lewego dolnego narożnika. Z drugiej strony cyfra „3" jest o wiele bardziej skomplikowana: charakteryzuje się dwiema ukośnymi krawędziami skierowanymi w stronę środka znaku, poziomymi, krótkimi liniami na górze i dole, kolejnymi ukośnymi krawędziami skierowanymi w przeciwną stronę itd. Czy moglibyśmy więc wyodrębnić informacje o tym, w których miejscach obrazu występują krawędzie, a następnie użyć jako cech nie pikseli, ale tych informacji?

Okazuje się, że wyszukiwanie krawędzi obrazu jest bardzo częstym zadaniem w widzeniu komputerowym, a na dodatek zaskakująco prostym. W tym celu używamy operacji zwanej *konwolucją* (inne nazwy to *splot*, *splot całkowy* lub *mnożenie splotowe*). Wykorzystuje ona jedynie mnożenie i dodawanie. Te dwie operacje są odpowiedzialne za zdecydowaną większość pracy, jaką wykonuje każdy model uczenia głębokiego zaprezentowany w tej książce!

Kup ksi k | Pole ksi k

Operacja konwolucji polega na zastosowaniu w obrazie tak zwanego *jądra splotu*. Jądro splotu to mała macierz, taka jak ta o rozmiarze 3×3 widoczna na rysunku 13.1 w prawym górnym narożniku.



Rysunek 13.1. Stosowanie jądra splotu w pojedynczej lokalizacji

Macierz 7×7 po lewej stronie to obraz, w którym zamierzamy zastosować jądro splotu. Operacja konwolucji mnoży element jądra splotu przez odpowiedni element bloku 3×3 z obrazu. Iloczyny są następnie sumowane. Na rysunku 13.1 pokazano przykład użycia jądra splotu z pojedynczym obszarem będącym blokiem 3×3, który otacza pole o numerze 18.

Napiszmy kod odpowiadający tej operacji. Najpierw utworzymy niewielką macierz 3×3:

```
top_edge = tensor([[-1,-1,-1],
                   [ 0, 0, 0],
                   [ 1, 1, 1]]).float()
```

Nazwijmy tę macierz jądrem splotu (ponieważ tak nazywają je mądrale zajmujący się widzeniem komputerowym). Oczywiście będziemy też potrzebować obrazu:

```
path = untar_data(URLs.MNIST_SAMPLE)
im3 = Image.open(path/'train'/'3'/'12.png')
show_image(im3);
```

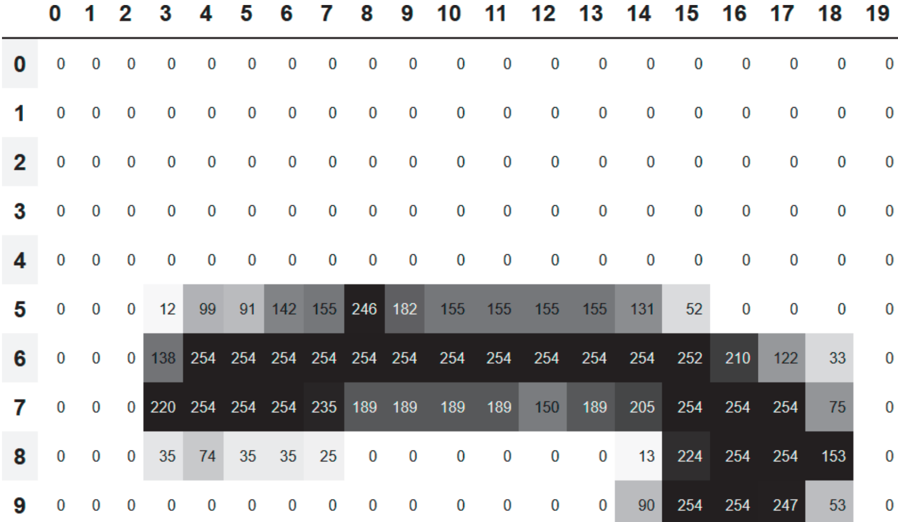
Teraz pobierzemy z górnej części obrazu kwadrat o wymiarach 3×3 i pomnożymy każdą z jego składowych przez odpowiedni element jądra splotu. Otrzymane iloczyny zsumujemy:

```
im3_t = tensor(im3)
im3_t[0:3,0:3] * top_edge
tensor([[-0., -0., -0.],
        [0., 0., 0.],
        [0., 0., 0.]])
(im3_t[0:3,0:3] * top_edge).sum()
tensor(0.)
```


Kup ksi k Pole ksi k

Uzyskaliśmy niezbyt interesujące wyniki — wszystkie piksele w lewym górnym narożniku są białe. Wybierzmy jednak kilka bardziej interesujących miejsc:

```
df = pd.DataFrame(im3_t[:10,:20])
df.style.set_properties(**{'font-size':'6pt'}).background_gradient('Greys')
```



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	12	99	91	142	155	246	182	155	155	155	155	131	52	0	0	0	0
6	0	0	0	138	254	254	254	254	254	254	254	254	254	254	254	252	210	122	33	0
7	0	0	0	220	254	254	254	235	189	189	189	189	150	189	205	254	254	254	75	0
8	0	0	0	35	74	35	35	25	0	0	0	0	0	0	13	224	254	254	153	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	90	254	254	247	53	0

Górna krawędź cyfry znajduje się w komórce o adresie 5,7. Powtórzmy obliczenia:

```
(im3_t[4:7,6:9] * top_edge).sum()
tensor(762.)
```

Prawa krawędź znajduje się w komórce 8,18. Co uzyskamy?

```
(im3_t[7:10,17:20] * top_edge).sum()
tensor(-29.)
```

W przypadku gdy kwadrat o wymiarach 3×3 piksele obejmuje górną krawędź cyfry (czyli miejsce, gdzie występują niskie wartości, a zaraz pod nimi duże), konwolucja zwraca dużą liczbę. Dzieje się tak, ponieważ wartości `-1` z jądra splotu mają niewielki wpływ na wynik. Dużą rolę odgrywają za to wartości równe `1`.

A teraz trochę matematyki. Filtr wykorzystuje dowolny obszar o rozmiarze 3×3. Załóżmy, że odpowiednie piksele kwadratu nazwaliśmy w następujący sposób:

$a1$ $a2$ $a3$

$a4$ $a5$ $a6$

$a7$ $a8$ $a9$

W takim przypadku filtr zwróci wartość równą $a1 + a2 + a3 - a7 - a8 - a9$. Jeśli wybierzemy obszar na obrazie, w którym wartości $a1$, $a2$ i $a3$ zostaną dodane do takich samych wartości komórek $a7$, $a8$ i $a9$, wyniki zniosą się i otrzymamy 0. Jednakże jeśli wartość $a1$ będzie większa niż $a7$, $a2$ większa niż $a8$, a $a3$ większa niż $a9$, w rezultacie uzyskamy wyższą liczbę. Zatem filtr wykrywa krawędzie

Kup ksi k Pole ksi k

poziome, a dokładniej: takie, w których przechodzimy od jasnych górnych fragmentów do ciemniejszych dolnych.

Gdy zmienimy filtr tak, aby na górze znajdowały się wartości 1, a na dole –1, zacznie wykrywać krawędzie przechodzące z obszaru ciemnego do jasnego. Umieszczenie wartości 1 i –1 w kolumnach zamiast w wierszach da w rezultacie filtr wykrywający krawędzie pionowe. Każdy zestaw wag będzie związany z innym rodzajem wyniku.

Utwórzmy funkcję, która będzie stosowana do określonego obszaru obrazu, a następnie sprawdźmy, czy uzyskamy taki sam wynik jak poprzednio:

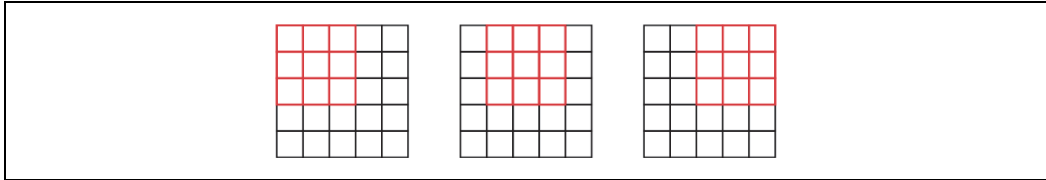
```
def apply_kernel(row, col, kernel):
    return (im3_t[row-1:row+2,col-1:col+2] * kernel).sum()
apply_kernel(5,7,top_edge)

tensor(762.)
```

Pamiętaj jednak, że nie możemy jej zastosować do narożnika (na przykład do lokalizacji 0,0), ponieważ w tym miejscu nie można uzyskać pełnego kwadratu 3×3.

Odwzorowywanie jądra splotu

Funkcji `apply_kernel()` możemy użyć w całej siatce współrzędnych. Oznacza to, że jądro splotu 3×3 zastosujemy dla każdego z obszarów o rozmiarach 3×3 należących do obrazu. Na rysunku 13.2 pokazano użycie jądra 3×3 w przypadku górnej krawędzi obrazu o rozmiarach 5×5.



Rysunek 13.2. Stosowanie jądra splotu w siatce współrzędnych

Aby uzyskać siatkę współrzędnych, możemy użyć zagnieżdżonej listy składanej, na przykład:

```
[[(i,j) for j in range(1,5)] for i in range(1,5)]

[[(1, 1), (1, 2), (1, 3), (1, 4)],
 [(2, 1), (2, 2), (2, 3), (2, 4)],
 [(3, 1), (3, 2), (3, 3), (3, 4)],
 [(4, 1), (4, 2), (4, 3), (4, 4)]]
```

Zagnieżdżone listy składane

Listy zagnieżdżone są często używane w języku Python, więc jeśli jeszcze nie miałeś z nimi do czynienia, poświęć kilka minut, aby się upewnić, że rozumiesz, jak działają. Postaraj się również poeksperymentować i utworzyć przykłady zagnieżdżonych list składanych.

Kup ksi k Pole ksi k

Oto wynik nałożenia jądra splotu na siatkę współrzędnych:

```
rng = range(1,27)
top_edge3 = tensor([[apply_kernel(i,j,top_edge) for j in rng] for i in rng])

show_image(top_edge3);
```

Wygląda nieźle! Górne krawędzie są czarne, a dolne — białe (gdyż są *przeciwieństwem* górnych). Ponieważ obraz zawiera także liczby ujemne, biblioteka *matplotlib* automatycznie zmieniła kolory, aby obszarom białym odpowiadały liczby najmniejsze, czarnym największe, a szarym zera.

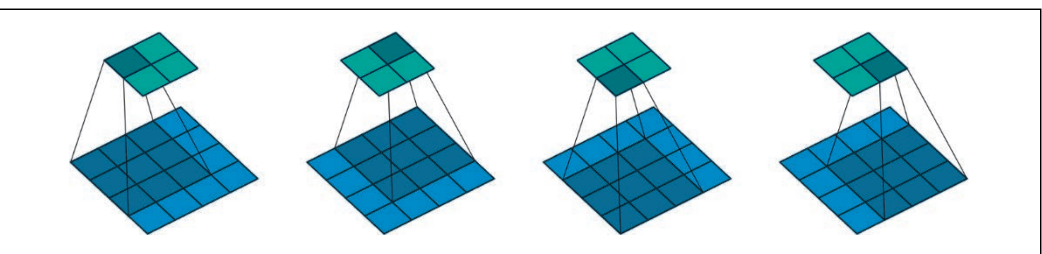
Spróbujmy wykonać tę samą operację dla krawędzi lewych:

```
left_edge = tensor([[-1,1,0],
                    [-1,1,0],
                    [-1,1,0]]).float()

left_edge3 = tensor([[apply_kernel(i,j,left_edge) for j in rng] for i in rng])

show_image(left_edge3);
```

Jak już wspominaliśmy, konwolucja to operacja nakładania jądra splotu na siatkę współrzędnych. Artykuł Vincenta Dumoulina i Francesco Visina *A Guide to Convolution Arithmetic for Deep Learning* („Jak używać arytmetyki konwolucyjnej w uczeniu głębokim?" — *https://oreil.ly/les1R*) zawiera wiele świetnych schematów pokazujących, w jaki sposób można stosować jądro splotu. Rysunek 13.3 prezentuje przykład wzięty z artykułu, pokazujący jasnoniebieski obraz siatki 4×4 z nałożonym ciemnoniebieskim jądrem splotu 3×3, co w rezultacie tworzy zieloną mapę aktywacji wyjściowej o rozmiarze 2×2.



Rysunek 13.3. Wynik zastosowania jądra 3×3 w obrazie 4×4 (dzięki uprzejmości Vincenta Dumoulina i Francesco Visina)

Spójrz na kształt wyniku. Jeśli obraz ma wysokość `h` i szerokość `w`, ile można w nim znaleźć okien o rozmiarach 3×3? Jak widać na rysunku, okna mają rozmiary `h-2` × `w-2`, więc otrzymany obraz ma wysokość `h-2` i szerokość `w-2`.

Kup ksi k   Pole ksi k

Nie zaimplementujemy funkcji konwolucyjnej od podstaw, ale użyjemy gotowego rozwiązania dostępnego w bibliotece PyTorch (działa o wiele szybciej niż jakikolwiek kod Pythona).

Konwolucje w bibliotece PyTorch

Konwolucja jest tak ważną i szeroko stosowaną operacją, że została wbudowana w bibliotekę PyTorch i występuje pod nazwą `F.conv2d` (pamiętaj, że `F` to import biblioteki *fastai* z `torch.nn.functional`, zgodnie z zaleceniami PyTorch). Z dokumentacji biblioteki PyTorch możemy się dowiedzieć, że funkcja `F.conv2d` zawiera następujące parametry:

`input`
: wejściowy tensor o kształcie `(minibatch, in_channels, iH, iW)`

`weight`
: filtr o kształcie `(out_channels, in_channels, kH, kW)`

Symbole `iH` i `iW` oznaczają wysokość i szerokość obrazu (w naszym przypadku 28,28), a `kH` i `kW` jest wysokością i szerokością jądra splotu (w naszym przypadku 3,3). Najwyraźniej jednak biblioteka PyTorch oczekuje, by oba parametry były tensorami 4. rzędu, podczas gdy obecnie mamy tylko tensory 2. rzędu (czyli macierze lub tablice dwuwymiarowe).

Wymóg dostarczenia dodatkowych wymiarów wynika stąd, że biblioteka PyTorch ma kilka asów w rękawie. Pierwsza sztuczka polega na tym, że konwolucja może zostać zastosowana w tym samym czasie do wielu obrazów. Oznacza to, że możemy jej od razu użyć ze wszystkimi elementami zawartymi w minipaczce!

Po drugie, biblioteka PyTorch może w tym samym czasie zastosować wiele jąder splotu. Utwórzmy więc również jądra o krawędziach w postaci przekątnej, a następnie umieśćmy wszystkie cztery jądra splotu w jednym tensorze:

```
diag1_edge = tensor([[ 0,-1, 1],
                     [-1, 1, 0],
                     [ 1, 0, 0]]).float()
diag2_edge = tensor([[ 1,-1, 0],
                     [ 0, 1,-1],
                     [ 0, 0, 1]]).float()

edge_kernels = torch.stack([left_edge, top_edge, diag1_edge, diag2_edge])
edge_kernels.shape

torch.Size([4, 3, 3])
```

Aby przeprowadzić testy, będziemy potrzebować obiektu `DataLoader` i przykładowej minipaczki. Skorzystajmy z interfejsu API bloków danych:

```
mnist = DataBlock((ImageBlock(cls=PILImageBW), CategoryBlock),
                  get_items=get_image_files,
                  splitter=GrandparentSplitter(),
                  get_y=parent_label)

dls = mnist.dataloaders(path)
xb,yb = first(dls.valid)
xb.shape

torch.Size([64, 1, 28, 28])
```


Kup ksi k   Pole ksi k

Podczas korzystania z bloków danych biblioteka *fastai* domyślnie umieszcza dane w karcie graficznej. W tym przypadku wybierzmy jednak procesor komputera:

```
xb,yb = to_cpu(xb),to_cpu(yb)
```

Jedna minipaczka składa się z 64 obrazów, z których każdy zawiera 1 kanał. Obrazy mają wymiary 28×28 pikseli. Funkcja `F.conv2d` obsługuje również obrazy wielokanałowe (kolorowe). *Kanał* to pojedynczy, podstawowy kolor obrazu — w przypadku zwykłych kolorowych obrazów dostępne są trzy kanały: czerwony, zielony i niebieski. Biblioteka PyTorch reprezentuje obraz jako tensor 3. rzędu o następujących wymiarach:

```
[liczba_kanałów, liczba_wierszy, liczba_kolumn]
```

W dalszej części rozdziału dowiesz się, jak można obsługiwać więcej niż jeden kanał. Jądra splotu przekazywane do funkcji `F.conv2d` muszą być tensorami 4. rzędu:

```
[liczba_kanałów_wejściowych, liczba_cech_wyjściowych, liczba_wierszy, liczba_kolumn]
```

Zmiennej `edge_kernels` brakuje obecnie jednego rzędu, Musimy poinformować bibliotekę PyTorch, że liczba kanałów wejściowych w jądrze wynosi 1. Możemy to zrobić przez umieszczenie osi o rozmiarze 1 (zwanej *osią jednostkową*) w miejscu oznaczonym *`liczba_kanałów_wejściowych`*. Aby wstawić oś jednostkową do tensora, używamy metody `unqueeze`:

```
edge_kernels.shape,edge_kernels.unsqueeze(1).shape
(torch.Size([4, 3, 3]), torch.Size([4, 1, 3, 3]))
```

Zmienna `edge_kernels` ma już odpowiedni kształt. Wywołajmy więc funkcję `conv2d`:

```
edge_kernels = edge_kernels.unsqueeze(1)

batch_features = F.conv2d(xb, edge_kernels)
batch_features.shape

torch.Size([64, 4, 26, 26])
```

Uzyskany kształt oznacza, że w minipaczce mamy 64 obrazy, a także używamy 4 jąder splotu i map krawędzi o rozmiarach 26×26 (zaczynaliśmy od obrazów 28×28, ale jak wspomnieliśmy, z każdej krawędzi należało odjąć po jednym pikselu). Otrzymaliśmy taki sam rezultat jak w przypadku samodzielnie stworzonego kodu:

```
show_image(batch_features[0,0]);
```

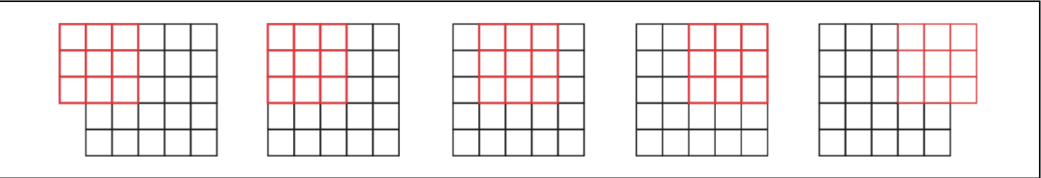
Najważniejsza sztuczka, jaką PyTorch ma w zanadrzu, polega na tym, że te zadania (czyli nakładanie wielu jąder splotu na wiele obrazów z wieloma kanałami) mogą być wykonywane równolegle w karcie graficznej. Procesory graficzne pracują najwydajniej, gdy wykonują współbieżnie wiele działań. Gdybyśmy uruchamiali każdą z operacji pojedynczo, wszystko działałoby setki razy wolniej (a nawet miliony razy wolniej w przypadku użycia samodzielnie stworzonej pętli konwolucji z poprzedniego punktu!). Dlatego warto przydzielać kartom graficznym jak najwięcej zadań wykonywanych jednocześnie.

Kup ksi k Pole ksi k

Byłoby miło, gdybyśmy nie musieli dla każdej osi tracić dwóch pikseli. Rozwiązanie polega na dodaniu *dopełnienia*, czyli po prostu dodatkowych pikseli umieszczonych na zewnątrz obrazu. Najczęściej dodawane są piksele z zerami.

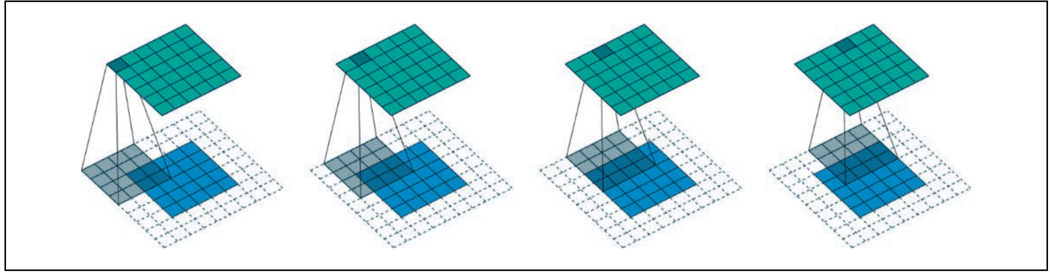
Kroki i dopełnienie

Dzięki odpowiedniemu dopełnieniu zapewnimy, że mapa aktywacji wyjściowej będzie miała taki sam rozmiar jak obraz oryginalny, co znacznie uprości konstruowanie architektur. Na rysunku 13.4 pokazano, w jaki sposób dopełnienie pozwala zastosować jądro splotu w narożnikach obrazu.



Rysunek 13.4. Konwolucja z dopełnieniem

Jak widać na rysunku 13.5, jeżeli mamy dane wejściowe o rozmiarze 5×5, jądro 4×4 i 2 piksele dopełnienia, otrzymujemy mapę aktywacji 6×6.

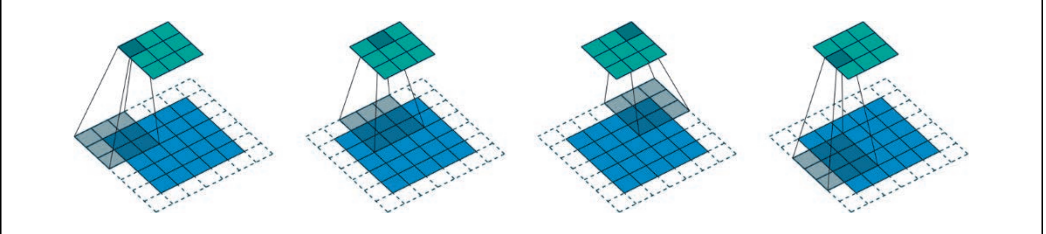


Rysunek 13.5. Jądro splotu 4×4 używane z danymi wejściowymi o rozmiarze 5×5 i dwoma pikselami wypełnienia (dzięki uprzejmości Vincenta Dumoulina i Francesco Visina)

Jeśli użyjemy jądra splotu o rozmiarze `ks` na `ks` (przy czym `ks` jest liczbą nieparzystą), wartość dopełnienia wymagana do zachowania takiego samego kształtu będzie równa `ks//2`. Gdyby `ks` był liczbą parzystą, wymagałoby to innego dopełnienia na górze (dole) i po lewej (prawej). W praktyce prawie nigdy jednak nie używamy parzystego rozmiaru filtra.

Po nałożeniu jądra splotu na siatkę współrzędnych przesuwaliśmy je następnie o jeden piksel. Ale możemy zrobić coś więcej — na przykład przesuwać jądro o dwa piksele po każdym użyciu, jak pokazano na rysunku 13.6. Taka operacja jest znana pod nazwą *konwolucji z krokiem 2* (ang. *stride-2 convolution*). W praktyce najpowszechniejszym rozmiarem jądra jest 3×3, a najczęstszym dopełnieniem jest wartość 1. Jak zobaczysz, konwolucje z krokiem 2 przydają się do zmniejszania rozmiaru wartości wynikowych, a konwolucje z krokiem 1 są przydatne do dodawania warstw bez zmiany rozmiaru wyjściowego.

Kup ksi k Pole ksi k



Rysunek 13.6. Jądro 3×3 używane z danymi wejściowymi o rozmiarze 5×5, konwolucją z krokiem 2 i jednym pikselem wypełnienia (dzięki uprzejmości Vincenta Dumoulina i Francesco Visina)

Użycie dopełnienia równego 1 i kroku o wartości 2 z obrazem o rozmiarze `h` × `w` pozwoli uzyskać rozmiar `(h+1)//2` × `(w+1)//2`. Oto ogólny wzór dla dowolnego rozmiaru:

```
(n + 2*pad - ks) // stride + 1
```

W powyższym wzorze `pad` oznacza dopełnienie, `ks` rozmiar jądra splotu, a `stride` krok.

A teraz przyjrzyjmy się, w jaki sposób są obliczane wartości pikseli w przypadku wyników operacji konwolucji.

Zrozumienie równań konwolucji

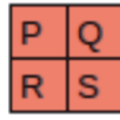
Próbując objaśnić operacje matematyczne dotyczące konwolucji, Matt Kleinsmith, student fast.ai, wpadł na bardzo sprytny pomysł zaprezentowania konwolucyjnej sieci neuronowej z różnych punktów widzenia (*https://oreil.ly/wZuBs*). To rozwiązanie było tak ciekawe i pomocne, że wykorzystaliśmy je w książce!

Oto obraz o wymiarach 3 na 3 piksele. Każdy z pikseli został oznaczony określoną literą:

A	B	C
D	E	F
G	H	J

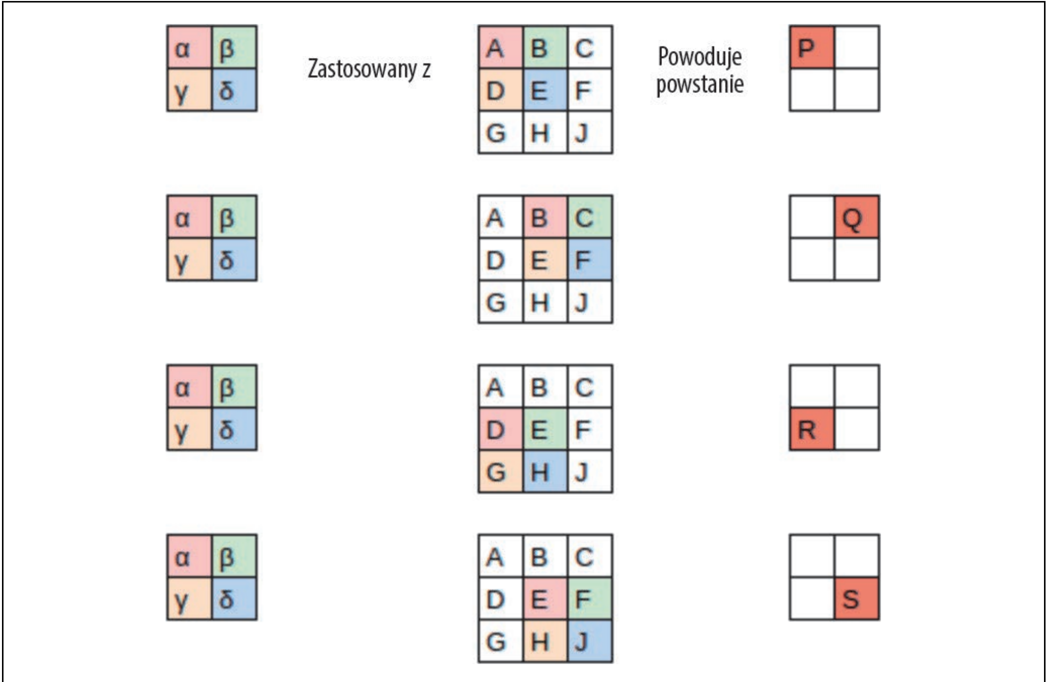
A oto jądro splotu. Każda waga została oznaczona literą grecką:

Ponieważ filtr może zostać użyty cztery razy z obrazem, uzyskujemy cztery wyniki:



Kup ksi k Pole ksi k

Rysunek 13.7 pokazuje, w jaki sposób zastosowaliśmy jądro do każdej części obrazu, by uzyskać odpowiedni wynik.



Rysunek 13.7. Stosowanie jądra splotu

Na rysunku 13.8 zaprezentowano widok równań.

$$\alpha * A + \beta * B + \gamma * D + \delta * E + b = P$$

$$\alpha * B + \beta * C + \gamma * E + \delta * F + b = Q$$

$$\alpha * D + \beta * E + \gamma * G + \delta * H + b = R$$

$$\alpha * E + \beta * F + \gamma * H + \delta * J + b = S$$

Rysunek 13.8. Równania

Zwróć uwagę, że wartość przesunięcia *b* jest taka sama dla każdej części obrazu. Przesunięcie, podobnie jak wagi (α, β, γ, δ), można potraktować jako część filtra.

Kup ksi k Pole ksi k

Oto interesujące spostrzeżenie — konwolucję można przedstawić jako specjalny rodzaj mnożenia macierzy, co pokazano na rysunku 13.9. Macierz wag jest taka sama jak w przypadku tradycyjnych sieci neuronowych. Jednak ma ona dwie szczególne właściwości:

1. Zera w kolorze szarym nie mogą być trenowane. Oznacza to, że nie zmienią wartości podczas całego procesu optymalizacji.
2. Niektóre z wag są sobie równe i chociaż można je wytrenować (czyli zmienić), wciąż muszą pozostać takie same. Są one nazywane *wagami współdzielonymi* (ang. *shared weights*).

Zera odpowiadają pikselom, których filtr nie może zmienić. Każdy wiersz macierzy wag odpowiada jednej operacji użycia filtra.

Rysunek 13.9. Konwolucja jako mnożenie macierzy

Skoro już wiesz, czym są konwolucje, użyjmy ich do stworzenia sieci neuronowej.

Pierwsza konwolucyjna sieć neuronowa

Nie ma powodu, by sądzić, że określone filtry krawędziowe są najbardziej użytecznymi jądrami splotu służącymi do rozpoznawania obrazów. Wiesz już, że jądra splotu stają się w dalszych warstwach złożonymi transformacjami cech z niższych poziomów. Nie potrafimy ich jednak samodzielnie tworzyć.

Najlepiej byłoby, gdyby model mógł się nauczyć wartości jąder splotu. Już wiemy, jak można to zrobić — za pomocą stochastycznego spadku wzdłuż gradientu! Dzięki temu model pozna cechy przydatne do klasyfikacji. Gdy używamy konwolucji zamiast (lub oprócz) zwykłych warstw liniowych, tworzymy *konwolucyjną sieć neuronową* (CNN).

Tworzenie konwolucyjnej sieci neuronowej

Wróćmy do podstawowej sieci neuronowej, której używaliśmy w rozdziale 4. Została ona zdefiniowana w następujący sposób:

```
simple_net = nn.Sequential(
    nn.Linear(28*28,30),
    nn.ReLU(),
    nn.Linear(30,1)
)
```


Kup ksi k Pole ksi k

Możemy wyświetlić definicję modelu:

```
simple_net
Sequential(
  (0): Linear(in_features=784, out_features=30, bias=True)
  (1): ReLU()
  (2): Linear(in_features=30, out_features=1, bias=True)
)
```

Chcemy stworzyć architekturę podobną do powyżej przedstawionego modelu, ale z wykorzystaniem warstw konwolucyjnych zamiast liniowych. Moduł `nn.Conv2d` jest odpowiednikiem modułu `F.conv2d`. Podczas tworzenia architektury jest on jednak wygodniejszy niż `F.conv2d`, ponieważ automatycznie tworzy macierz wag, gdy go konkretyzujemy.

Oto możliwa architektura:

```
broken_cnn = sequential(
    nn.Conv2d(1,30, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.Conv2d(30,1, kernel_size=3, padding=1)
)
```

Należy zauważyć, że nie musieliśmy określać wartości `28*28` jako rozmiaru wejściowego. Wynika to stąd, że dla każdego piksela warstwa liniowa musi określić wagę w macierzy wag. Powinna więc znać informację o tym, ile jest pikseli. Konwolucja zostaje jednak automatycznie zastosowana do każdego z nich. Jak już wiesz, wagi zależą tylko od liczby kanałów wejściowych i wyjściowych oraz rozmiaru jądra splotu.

Zastanów się przez chwilę nad tym, jaki może być kształt wyjściowy. Sprawdźmy zatem, czy dobrze odgadłeś:

```
broken_cnn(xb).shape
torch.Size([64, 1, 28, 28])
```

Takiego modelu nie możemy jeszcze użyć do klasyfikacji, ponieważ potrzebujemy pojedynczej aktywacji wyjściowej na obraz, a nie mapy aktywacji o rozmiarze 28×28. Jednym ze sposobów rozwiązania tego problemu jest użycie wystarczającej liczby konwolucji z krokiem 2, tak aby ostatnia warstwa miała rozmiar 1. Po jednej konwolucji z krokiem 2 rozmiar będzie wynosił 14×14; po dwóch 7×7; następnie 4×4, 2×2 i wreszcie 1.

Przetestujmy to. Najpierw zdefiniujemy funkcję z podstawowymi parametrami, których będziemy używać w każdej konwolucji:

```
def conv(ni, nf, ks=3, act=True):
    res = nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)
    if act: res = nn.Sequential(res, nn.ReLU())
    return res
```

Modyfikacja

Właściwa modyfikacja fragmentów sieci neuronowych znacznie zmniejsza prawdopodobieństwo wystąpienia błędów wynikających z niespójności w architekturach i powoduje, że bardziej oczywiste staje się, które elementy warstw faktycznie się zmieniają.

Kup ksi k Pole ksi k

Gdy używamy konwolucji z krokiem 2, często jednocześnie zwiększamy liczbę cech. Dzieje się tak, ponieważ czterokrotnie zmniejszamy liczbę aktywacji w mapie — nie chcemy od razu zbytnio zmniejszać pojemności warstwy.

Słownictwo: kanały i cechy

Te dwa terminy są w dużej mierze używane zamiennie i odnoszą się do rozmiaru drugiej osi macierzy wag, czyli liczby aktywacji na komórkę siatki po wykonaniu operacji konwolucji. Cechy nigdy nie są używane w odniesieniu do danych wejściowych, jednak kanały mogą dotyczyć takich danych (zazwyczaj oznaczają wtedy kolory) albo aktywacji wewnątrz sieci.

W taki sposób możemy zbudować prosty model konwolucyjnej sieci neuronowej:

```
simple_cnn = sequential(
    conv(1 ,4),            #14x14
    conv(4 ,8),            #7x7
    conv(8 ,16),           #4x4
    conv(16,32),           #2x2
    conv(32,2, act=False), #1x1
    Flatten(),
)
```

Mówi Jeremy

W wierszach kodu odpowiadających operacjom konwolucji lubię dodawać komentarze, które informują, jaki będzie rozmiar mapy aktywacji po każdej warstwie. W powyższym przypadku założyłem, że rozmiar wejściowy wynosi 28×28.

Teraz sieć generuje dwie aktywacje, które są odwzorowywane na dwa możliwe poziomy w etykietach:

```
simple_cnn(xb).shape
torch.Size([64, 2])
```

Utwórzmy obiekt `Learner`:

```
learn = Learner(dls, simple_cnn, loss_func=F.cross_entropy, metrics=accuracy)
```

Aby poznać szczegóły związane z modelem, możemy użyć funkcji `summary`:

```
learn.summary()
Sequential (Input shape: ['64 x 1 x 28 x 28'])
================================================================
Layer (type)         Output Shape         Param #    Trainable
================================================================
Conv2d               64 x 4 x 14 x 14     40         True
________________________________________________________________
ReLU                 64 x 4 x 14 x 14     0          False
________________________________________________________________
Conv2d               64 x 8 x 7 x 7       296        True
________________________________________________________________
ReLU                 64 x 8 x 7 x 7       0          False
________________________________________________________________
Conv2d               64 x 16 x 4 x 4      1,168      True
________________________________________________________________
ReLU                 64 x 16 x 4 x 4      0          False
```


Kup ksi k   Pole ksi k

```
________________________________________________________________
Conv2d               64 x 32 x 2 x 2      4,640      True
________________________________________________________________
ReLU                 64 x 32 x 2 x 2      0          False
________________________________________________________________
Conv2d               64 x 2 x 1 x 1       578        True
________________________________________________________________
Flatten              64 x 2               0          False
________________________________________________________________

Total params: 6,722
Total trainable params: 6,722
Total non-trainable params: 0

Optimizer used: <function Adam at 0x7fbc9c258cb0>
Loss function: <function cross_entropy at 0x7fbca9ba0170>

Callbacks:
  - TrainEvalCallback
  - Recorder
  - ProgressCallback
```

Zwróć uwagę, że wynik końcowej warstwy `Conv2d` jest równy `64×2×1×1`. Musimy usunąć dodatkowe osie `1×1` — to właśnie wykonuje funkcja `Flatten`. Działa ona w zasadzie tak samo jak metoda `squeeze` z biblioteki PyTorch, ale w postaci modułu.

Sprawdźmy, jak trenuje się właśnie utworzoną sieć! Ponieważ jest głębsza niż ta, którą wcześniej budowaliśmy od podstaw, użyjemy niższej wartości współczynnika uczenia i większej liczby epok:

```
learn.fit_one_cycle(2, 0.01)
```

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	0,072684	0,045110	0,990186	00:05
1	0,022580	0,030775	0,990186	00:05

Sukces! Wyniki są zbliżone do tych, które osiągnęliśmy w przypadku modelu `resnet18`, chociaż trenowanie zajmuje więcej epok i używamy niższego współczynnika uczenia. Powinieneś więc jeszcze poznać kilka innych sztuczek. Jesteśmy jednak coraz bliżej momentu, w którym stworzymy od podstaw nowoczesną konwolucyjną sieć neuronową.

Zrozumienie arytmetyki konwolucji

W podsumowaniu informacji o modelu mogliśmy zobaczyć, że dane wejściowe mają rozmiar `64×1×28×28`. Poszczególne osie to `batch` (minipaczka), `channel` (kanał), `height` (wysokość) i `width` (szerokość). Taki zestaw jest często przedstawiany w postaci skrótu `NCHW` (gdzie `N` oznacza wielkość paczki). Z drugiej strony biblioteka TensorFlow używa kolejności osi `NHWC`. Oto pierwsza warstwa:

```
m = learn.model[0]
m
Sequential(
  (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (1): ReLU()
)
```


Kup ksi k Pole ksi k

Mamy więc 1 kanał wejściowy, 4 kanały wyjściowe i jądro splotu 3×3. Sprawdźmy wagi pierwszej konwolucji:

```
m[0].weight.shape
torch.Size([4, 1, 3, 3])
```

W podsumowaniu widać, że mamy 40 parametrów, ale przecież `4*1*3*3 = 36`. Jakie są więc pozostałe cztery parametry? Zobaczmy, co zawiera przesunięcie:

```
m[0].bias.shape
torch.Size([4])
```

Możemy teraz wykorzystać powyższą informację, aby wyjaśnić wcześniejsze stwierdzenie: „Gdy używamy konwolucji z krokiem 2, często zwiększamy liczbę cech. Dzieje się tak, ponieważ czterokrotnie zmniejszamy liczbę aktywacji w mapie — nie chcemy od razu zbytnio zmniejszać pojemności warstwy".

Każdy kanał zawiera jedno przesunięcie (gdy nie mamy do czynienia z danymi wejściowymi, kanały są czasem nazywane *cechami* lub *filtrami*). Kształt wyjściowy wynosi `64×4×14×14`, a zatem stanie się on kształtem wejściowym dla następnej warstwy. Następna warstwa, zgodnie z wynikiem funkcji `summary`, ma 296 parametrów. Aby uprościć obliczenia, zignorujmy oś odpowiadającą minipaczce. Tak więc dla każdej z `14*14 = 196` lokalizacji mnożymy `296-8 = 288` wag (dla uproszczenia pomijając przesunięcie), czyli dla tej warstwy uzyskujemy `196*288 = 56448` operacji iloczynów. Następna warstwa również będzie miała `7*7*(1168-16) = 56448` operacji iloczynów.

Okazuje się, że konwolucja z krokiem 2 zmniejszyła o połowę *rozmiar siatki*, z `14×14` na `7×7`. Jednocześnie *liczba filtrów* podwoiła się, z 8 do 16, co spowodowało, że ilość obliczeń się nie zmieniła. Gdybyśmy zostawili taką samą liczbę kanałów w każdej warstwie z krokiem 2, ilość wykonywanych obliczeń byłaby coraz mniejsza w miarę przemieszczania się w głąb sieci. Wiemy jednak, że głębsze warstwy muszą rozpoznawać semantycznie złożone cechy (takie jak oczy lub futro), więc nie spodziewamy się, że wykonywanie *mniejszej* ilości obliczeń ma sens.

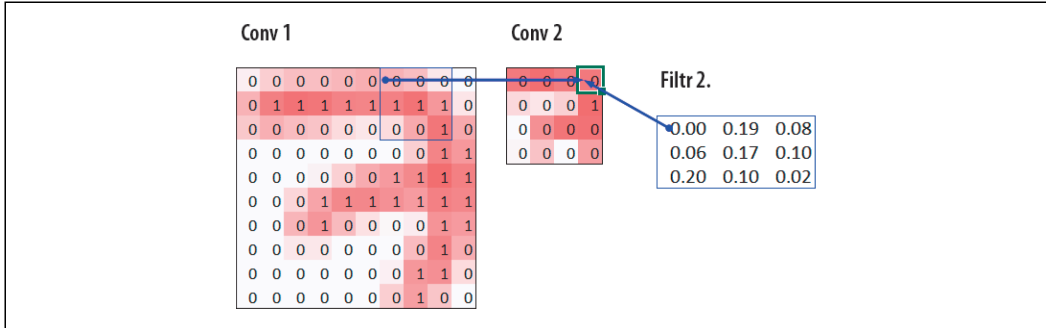
Innym podejściem są pola receptywne.

Pola receptywne

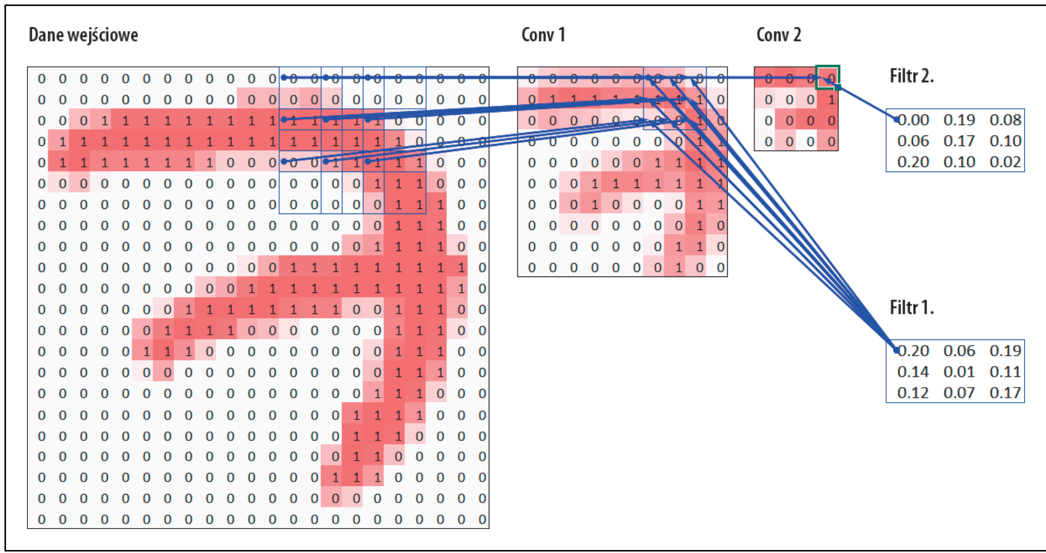
Pole receptywne to obszar obrazu, który bierze udział w obliczaniu warstwy. Zamieszczony na naszej stronie internetowej (*https://book.fast.ai/*) arkusz kalkulacyjny Excel o nazwie *conv-example.xlsx* przedstawia obliczenia dwóch konwolucyjnych warstw z krokiem 2 wykorzystujące cyfry ze zbioru MNIST. Każda warstwa ma jedno jądro splotu. Na rysunku 13.10 pokazano, co uzyskamy, gdy klikniemy jedną z komórek obszaru *conv2*, który przedstawia dane wyjściowe drugiej warstwy konwolucyjnej, a następnie wybierzemy opcję *Śledź poprzedniki*.

Komórka, którą kliknęliśmy, została otoczona zieloną ramką. Komórki podświetlone na niebiesko są *poprzednikami*, czyli polami, które są używane do jej obliczenia. Tym polom odpowiada obszar 3×3 komórek z warstwy wejściowej (po lewej) i komórki z filtra (po prawej). Kliknijmy teraz ponownie opcję *Śledź poprzedniki*, aby zobaczyć, jakie komórki są używane do obliczania danych wejściowych. To działanie przedstawiono na rysunku 13.11.

Kup książkę Poleć książkę



Rysunek 13.10. Bezpośrednie poprzedniki warstwy Conv2



Rysunek 13.11. Drugie poprzedniki warstwy Conv2

W tym przykładzie mamy tylko dwie warstwy konwolucyjne z krokiem 2, dlatego śledząc, wracamy do obrazu wejściowego. Możemy zauważyć, że obszar komórek o wymiarach 7×7 w warstwie wejściowej jest używany do obliczenia pojedynczej zielonej komórki w warstwie *Conv2*. Ten obszar 7×7 jest *polem receptywnym* dla tej zielonej komórki aktywacji. Widzimy również, że w filtrze jest potrzebne drugie jądro splotu, ponieważ używamy dwóch warstw.

Okazuje się, że im bardziej zagłębimy się w sieć (a dokładniej: im więcej konwolucji z krokiem 2 znajduje się przed warstwą), tym większe będzie pole receptywne dla aktywacji w danej warstwie. Duże pole receptywne oznacza, że znacząca ilość informacji pochodzącej z obrazu wejściowego zostanie użyta do obliczenia każdej z aktywacji. W głębszych warstwach sieci dostępne są semantycznie bogate cechy odpowiadające większym polom receptywnym. Dlatego też dla każdej z tych cech będziemy potrzebować większej liczby wag, aby model mógł sobie poradzić z rosnącą złożonością. To inny sposób wyrażenia tego samego wniosku, który przedstawiliśmy w poprzednim punkcie: gdy w sieci zaczniemy używać konwolucji z krokiem 2, powinniśmy zwiększyć liczbę kanałów.

Kup ksi k
Pole ksi k

Tworząc ten rozdział, stanęliśmy w obliczu wielu pytań, na które musieliśmy odpowiedzieć, aby jak najlepiej wyjaśnić działanie konwolucyjnych sieci neuronowych. Co ciekawe, większość odpowiedzi znaleźliśmy na Twitterze. Zanim przejdziemy do obrazów wykorzystujących kolory, zrobimy sobie krótką przerwę i potweetujmy.

Kilka uwag o Twitterze

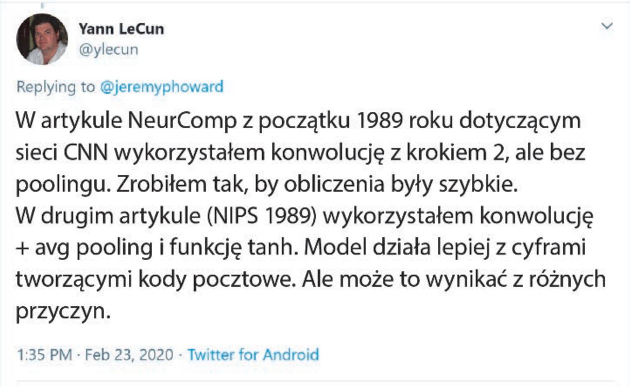
Ogólnie mówiąc, nie jesteśmy zbyt aktywnymi użytkownikami sieci społecznościowych. Celem tej książki jest sprawienie, abyś został świetnym specjalistą wykorzystującym w praktyce uczenie głębokie. Koniecznie trzeba więc wspomnieć, jak ważny był Twitter podczas poszukiwań wiedzy związanej z tą dziedziną nauki.

Istnieje zupełnie odmienny zakamarek Twittera, bardzo odległy od Donalda Trumpa i Kardashianów, w którym naukowcy i specjaliści zajmujący się uczeniem głębokim codziennie rozmawiają o swoich sprawach. Gdy pisaliśmy ten podrozdział, Jeremy chciał jeszcze raz sprawdzić, czy to, co stwierdziliśmy o konwolucji z krokiem 2, było poprawne, więc zadał pytanie:

Kilka minut później pojawiła się odpowiedź:

Christian Szegedy jest głównym twórcą architektury Inception (*https://oreil.ly/hGE_Y*), zwycięzcy konkursu ImageNet z 2014 roku. Wygłasza również wiele ważnych uwag dotyczących współczesnych sieci neuronowych. Dwie godziny później pojawiła się następna wypowiedź:

Kup ksi k Pole ksi k



Czy rozpoznajesz tego naukowca? Spotkałeś go w rozdziale 2., gdy mówiliśmy o zdobywcach nagrody Turinga, którzy stworzyli podstawy uczenia głębokiego!

Jeremy poprosił na Twitterze również o pomoc w sprawdzeniu, czy opis wygładzania etykiet zaprezentowany w rozdziale 7. jest poprawny. Ponownie otrzymał odpowiedź od Christiana Szegedy (wygładzanie etykiet zostało pierwotnie wprowadzone w artykule dotyczącym architektury Inception):

Wielu znanych naukowców jest stałymi użytkownikami Twittera i chętnie komunikuje się z szerszą społecznością. Warto odwiedzić profil Jeremy'ego (*https://oreil.ly/sqOI7*) lub Sylvaina (*https://oreil.ly/VWYHY*). W ten sposób możesz zapoznać się z listą użytkowników Twittera, którzy naszym zdaniem prowadzą ciekawe dyskusje.

To przede wszystkim dzięki Twitterowi jesteśmy na bieżąco z interesującymi artykułami, nowymi wersjami oprogramowania i innymi wiadomościami dotyczącymi uczenia głębokiego. Aby nawiązać kontakt ze społecznością, warto zaangażować się zarówno na forach fast.ai (*https://forums.fast.ai/*), jak i na Twitterze.

A teraz wróćmy do głównego zagadnienia poruszanego w tym rozdziale. Do tej pory przykładowe obrazy prezentowaliśmy tylko w odcieniach szarości, z jedną wartością przypadającą na piksel. W praktyce większość kolorowych obrazów używa trzech wartości składających się na piksel i określających jego kolor.

Kup ksi k   Pole ksi k

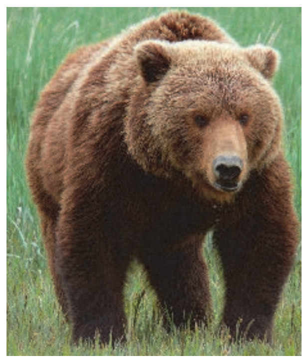
Obrazy kolorowe

Obraz kolorowy to tensor 3. rzędu:

```
im = image2tensor(Image.open('images/grizzly.jpg'))
im.shape
```

```
torch.Size([3, 1000, 846])
```

```
show_image(im);
```



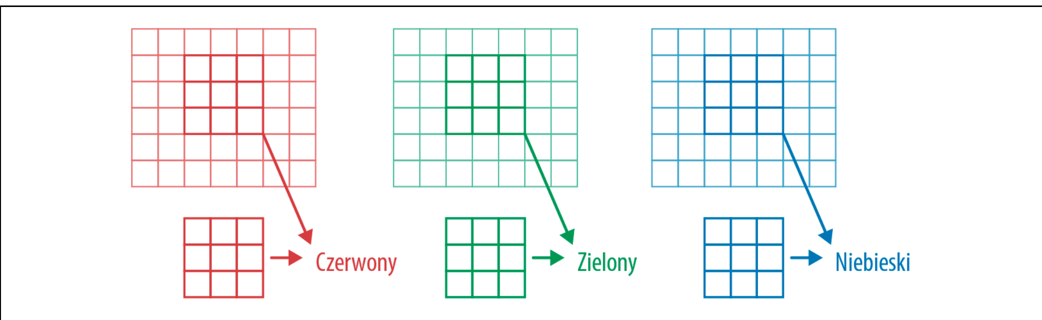
Pierwsza oś zawiera kanały czerwony, zielony i niebieski:

```
_,axs = subplots(1,3)
for bear,ax,color in zip(im,axs,('Reds','Greens','Blues')):
    show_image(255-bear, ax=ax, cmap=color)
```

Widziałeś, jak przebiegała operacja konwolucji w przypadku jednego filtra użytego z jednym kanałem obrazu (przykłady wykorzystywały obszar będący kwadratem). Warstwa konwolucyjna pobiera obraz z pewną liczbą kanałów (trzema przypadającymi na pierwszą warstwę, jeśli plik to zwykły obraz z kolorami RGB), a następnie generuje obraz zawierający inną ich liczbę. Podobnie jak w przypadku rozmiaru ukrytego, który reprezentuje liczbę neuronów w warstwie liniowej, możemy również mieć tyle filtrów, z których każdy w czymś się specjalizuje (część wykrywa krawędzie poziome, inne wykrywają pionowe itd.), ile chcemy. Coś podobnego widziałeś już na przykładach, które analizowaliśmy w rozdziale 2.

Kup ksi k   Pole ksi k

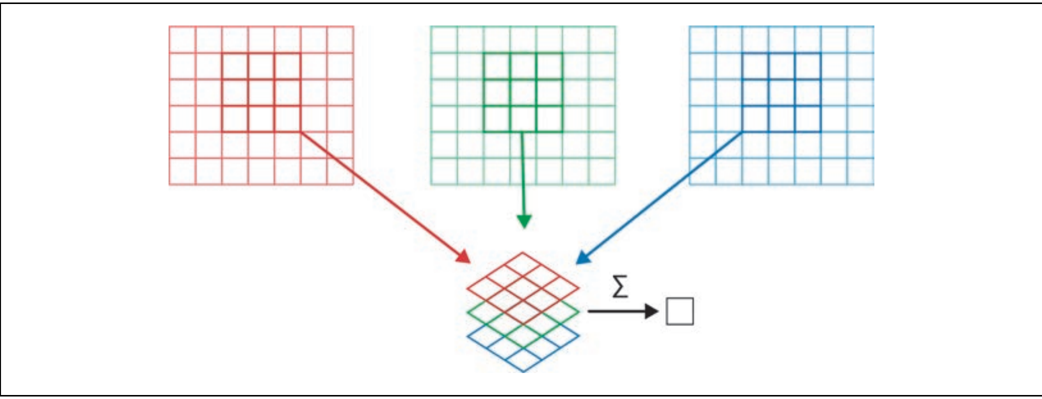
W przesuwającym się oknie używamy określonej liczby kanałów, dlatego potrzebujemy tyle samo filtrów (nie używamy tego samego jądra splotu dla wszystkich kanałów). Jądro nie ma więc rozmiaru 3 × 3, ale ch_in (dla kanałów wejściowych) na 3 × 3. Dla każdego z kanałów mnożymy elementy okna przez elementy odpowiedniego filtra, a następnie (jak już widziałeś) sumujemy wyniki i wszystkie filtry. W przykładzie podanym na rysunku 13.12 wynikiem warstwy konwolucyjnej dla okna jest suma czerwony + zielony + niebieski.



Rysunek 13.12. Konwolucja obrazu RGB

Aby więc zastosować konwolucję do kolorowego obrazu, potrzebujemy tensora jądra o rozmiarze pasującym do pierwszej osi. Dla każdej lokalizacji mnożone są ze sobą odpowiednie części jądra splotu i elementy obrazu.

Jak pokazano na rysunku 13.13, dla każdej cechy wyjściowej iloczyny są sumowane w celu uzyskania jednej wartości dla pojedynczego punktu siatki.



Rysunek 13.13. Dodawanie filtrów RGB

Uzyskujemy więc filtry, których liczba wynosi ch_out. Końcowym wynikiem warstwy konwolucyjnej będzie paczka obrazów zawierających ch_out kanałów. Wysokość i szerokość będą wynikały z podanego wcześniej wzoru. Otrzymamy ch_out tensorów o rozmiarze ch_in × ks × ks, które odwzorujemy w jednym dużym tensorze o czterech wymiarach. W przypadku biblioteki PyTorch kolejność wymiarów dla wag jest następująca: ch_out × ch_in × ks × ks.

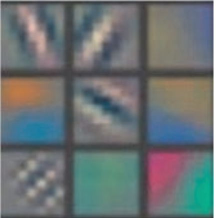
Kup ksi k Pole ksi k

Dodatkowo dla każdego z filtrów możemy użyć przesunięcia. W przypadku wcześniejszego przykładu wynikiem warstwy konwolucyjnej byłaby wartość $y_R + y_G + y_B + b$. Podobnie jak w warstwie liniowej istnieje tyle przesunięć, ile jąder splotu, więc całkowite przesunięcie jest wektorem o rozmiarze `ch_out`.

Podczas przygotowywania konwolucyjnej sieci neuronowej do trenowania z użyciem obrazów kolorowych nie są wymagane żadne specjalne mechanizmy. Należy się jedynie upewnić, że pierwsza warstwa będzie miała trzy wejścia.

Istnieje wiele sposobów przetwarzania obrazów kolorowych. Na przykład możesz je zamienić na czarno-białe, przekształcić przestrzeń kolorów RGB na HSV (barwa, nasycenie i wartość) itd. Eksperymentalnie dowiedziono, że zmiana kodowania kolorów nie będzie miała żadnego wpływu na wyniki modelu, o ile podczas transformacji nie stracisz jakichś informacji. Zatem przekształcenie obrazów na czarno-białe jest złym pomysłem, ponieważ całkowicie usuwa informacje o kolorach (co może mieć duże znaczenie — na przykład pewne rasy zwierząt domowych mogą mieć charakterystyczny kolor sierści). Jednak zmiana przestrzeni kolorów na HSV zwykle nie robi żadnej różnicy.

Teraz wiesz, co oznaczają zdjęcia (pochodzące z artykułu Zeilera i Fergusa — *https://oreil.ly/Y6dzZ*), które umieściliśmy w rozdziale 1. i które mają związek z tym, czego uczy się sieć neuronowa! Dla przypomnienia, oto zdjęcie przedstawiające niektóre wagi warstwy 1.:



Powyższy rysunek oznacza pobranie dla każdej cechy wyjściowej trzech wycinków jądra splotu i wyświetlenie ich jako obrazów. Widać, że chociaż twórcy sieci neuronowej nigdy nie stworzyli specjalnych jąder splotu służących do wyszukiwania krawędzi, sieć neuronowa automatycznie odkryła tę cechę za pomocą stochastycznego spadku wzdłuż gradientu.

A teraz zobaczmy, jak można wytrenować konwolucyjną sieć neuronową. Postaramy się zaprezentować wszystkie techniki, których biblioteka *fastai* używa w tle, by osiągnąć dobrą wydajność trenowania.

Ulepszanie stabilności trenowania

Ponieważ potrafimy już bardzo dobrze odróżniać cyfrę „3” od „7”, przejdźmy do czegoś trudniejszego — rozpoznawania wszystkich 10 cyfr. Oznacza to, że zamiast `MNIST_SAMPLE` będziemy musieli użyć bazy `MNIST`:

```
path = untar_data(URLs.MNIST)
path.ls()
(#2) [Path('testing'),Path('training')]
```


Kup ksi k Pole ksi k

Dane znajdują się w dwóch folderach o nazwach *training* i *testing*, więc musimy o tym poinformować obiekt `GrandparentSplitter` (domyślnie wywołuje metody `train` i `valid`). Operację tę przeprowadzimy w funkcji `get_dls`, którą zdefiniujemy, aby uprościć późniejszą zmianę rozmiaru paczki:

```
def get_dls(bs=64):
    return DataBlock(
        blocks=(ImageBlock(cls=PILImageBW), CategoryBlock),
        get_items=get_image_files,
        splitter=GrandparentSplitter('training','testing'),
        get_y=parent_label,
        batch_tfms=Normalize()
    ).dataloaders(path, bs=bs)

dls = get_dls()
```

Pamiętaj, że zawsze warto przejrzeć dane przed ich użyciem:

```
dls.show_batch(max_n=9, figsize=(4,4))
```

Gdy mamy już dane, możemy wytrenować na nich prosty model.

Prosty model bazowy

Wcześniej w tym rozdziale zbudowaliśmy model oparty na funkcji `conv`:

```
def conv(ni, nf, ks=3, act=True):
    res = nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)
    if act: res = nn.Sequential(res, nn.ReLU())
    return res
```

Stwórzmy prostą konwolucyjną sieć neuronową i potraktujmy ją jako punkt odniesienia. Wykorzystamy to samo rozwiązanie co wcześniej, ale z jedną poprawką: użyjemy większej liczby aktywacji. Ponieważ mamy więcej cyfr do rozróżnienia, prawdopodobnie będziemy musieli wytrenować więcej filtrów.

Jak stwierdziliśmy, chcemy podwajać liczbę filtrów za każdym razem, gdy używamy warstwy z krokiem 2. Jednym ze sposobów na zwiększenie liczby filtrów w sieci jest podwojenie liczby aktywacji w pierwszej warstwie. Wówczas każda kolejna warstwa powiększy się dwukrotnie.

Spowoduje to jednak pojawienie się ledwie uchwytnego problemu. Rozważmy jądro splotu, które jest stosowane do każdego piksela. Domyślnie używamy jądra o wymiarach 3 na 3 piksele. W związku z tym w każdej lokalizacji istnieje w sumie $3 \cdot 3 = 9$ pikseli, do których stosowane jest jądro.

Kup ksi k Pole ksi k

Wcześniej pierwsza warstwa miała cztery filtry wyjściowe, a zatem wyznaczyliśmy cztery wartości na podstawie dziewięciu pikseli. Pomyśl, co się stanie, jeśli podwoimy liczbę filtrów wyjściowych. Gdy w takim przypadku zastosujemy jądro, użyjemy dziewięciu pikseli do wyznaczenia ośmiu liczb. Oznacza to, że model niewiele się nauczy: rozmiar wyjściowy będzie prawie taki sam jak rozmiar wejściowy. Sieci neuronowe będą tworzyć użyteczne cechy tylko wtedy, gdy będą do tego zmuszone — to znaczy jeśli liczba wyjść z operacji będzie znacznie mniejsza niż liczba wejść.

Aby rozwiązać ten problem, możemy w pierwszej warstwie użyć większego jądra splotu. Jeśli użyjemy jądra 5×5 pikseli, każda operacja obejmie 25 pikseli. Utworzenie w przypadku tej konfiguracji ośmiu filtrów sprawi, że sieć neuronowa będzie musiała znaleźć kilka przydatnych cech:

```
def simple_cnn():
    return sequential(
        conv(1 ,8, ks=5),        #14x14
        conv(8 ,16),             #7x7
        conv(16,32),             #4x4
        conv(32,64),             #2x2
        conv(64,10, act=False),  #1x1
        Flatten(),
    )
```

Okazuje się, że podczas trenowania możemy zajrzeć do wnętrza modeli. Dzięki temu uzyskujemy dodatkowe informacje pozwalające lepiej przeprowadzać proces trenowania. Aby to zrobić, używamy wywołania zwrotnego `ActivationStats`, które rejestruje średnią, odchylenie standardowe i histogram aktywacji dla każdej warstwy zdolnej do trenowania (jak już wiesz, wywołania zwrotne służą do modyfikowania sposobu działania pętli treningowej; więcej szczegółów na ten temat poznasz w rozdziale 16.):

```
from fastai.callback.hook import *
```

Chcemy szybko zakończyć proces trenowania, a to oznacza użycie wysokiego współczynnika uczenia. Zobaczmy, co uzyskamy przy wartości 0,06:

```
def fit(epochs=1):
    learn = Learner(dls, simple_cnn(), loss_func=F.cross_entropy,
                    metrics=accuracy, cbs=ActivationStats(with_hist=True))
    learn.fit(epochs, 0.06)
    return learn
learn = fit()
```

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	2,307071	2,305865	0,113500	00:16

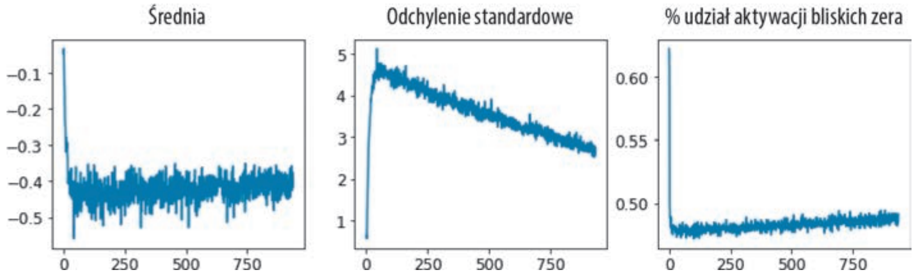
Trenowanie nie powiodło się! Dowiedzmy się, jaki był tego powód.

Jedną z przydatnych cech wywołań zwrotnych przekazywanych do obiektu `Learner` jest to, że są od razu dostępne. Ich nazwy są takie same jak nazwy odpowiednich klas (z wyjątkiem tego, że zawierają wyłącznie małe litery, a także znaki podkreślenia jako separatory wyrazów). Tak więc wywołanie zwrotne `ActivationStats` jest dostępne za pośrednictwem funkcji `activation_stats`. Na pewno pamiętasz funkcję `learn.recorder`. Zgadnij, jak została zaimplementowana. Zgadza się, jest to po prostu wywołanie zwrotne o nazwie `Recorder`!

Kup ksi k Pole ksi k

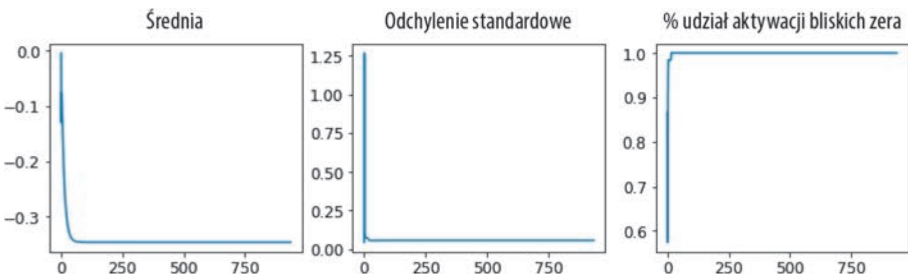
Wywołanie zwrotne `ActivationStats` zawiera przydatne narzędzia służące do rysowania wartości aktywacji w trakcie trenowania. Funkcja `plot_layer_stats(`*`idx`*`)` wykreśla średnią i odchylenie standardowe aktywacji warstwy o numerze *`idx`*, a także procentowy udział aktywacji bliskich zera. Oto wykres pierwszej warstwy:

```
learn.activation_stats.plot_layer_stats(0)
```



Ogólnie rzecz biorąc, model podczas trenowania powinien charakteryzować się spójną, a przynajmniej gładką średnią i takim samym odchyleniem standardowym aktywacji warstw. Aktywacje bliskie zera są szczególnie problematyczne, oznacza to bowiem, że w modelu istnieją obliczenia, które w ogóle nic nie robią (ponieważ mnożenie przez zero daje zero). Gdy w warstwie pojawia się kilka zer, przenoszą się one do następnej itd., co powoduje powstawanie coraz większej liczby zer. Oto przedostatnia warstwa sieci:

```
learn.activation_stats.plot_layer_stats(-2)
```



Zgodnie z oczekiwaniami problemy nasilają się na samym końcu sieci, gdyż po każdej warstwie rośnie poziom niestabilności i liczba zerowych aktywacji. Sprawdźmy, co moglibyśmy zrobić, aby trenowanie było bardziej stabilne.

Zwiększenie wielkości paczki

Jednym ze sposobów uczynienia procesu trenowania bardziej stabilnym jest zwiększenie wielkości paczki. Większe paczki zawierają dokładniejsze gradienty, które są obliczane na podstawie większej ilości danych. Z drugiej strony zwiększona wielkość paczki oznacza mniej przetworzonych paczek w epoce, czyli mniej okazji dla modelu do aktualizacji wag. Zobaczmy, czy pomoże zwiększenie wielkości paczki do 512:

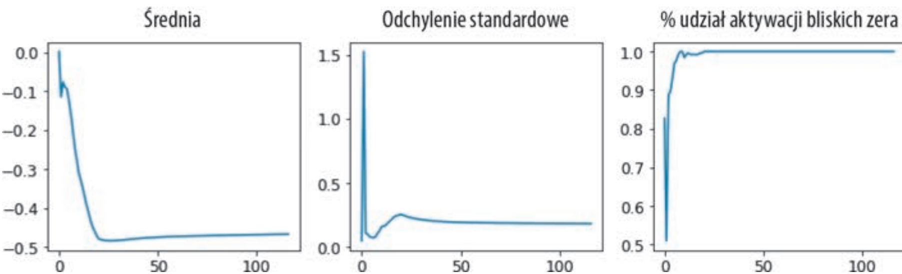
```
dls = get_dls(512)
learn = fit()
```


Kup ksi k Pole ksi k

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	2,309385	2,302744	0,113500	00:08

Sprawdźmy, jak wygląda warstwa przedostatnia:

```
learn.activation_stats.plot_layer_stats(-2)
```



Ponownie większość aktywacji jest zbliżona do zera. Zobaczmy, co jeszcze możemy zrobić, aby poprawić stabilność trenowania.

Trenowanie jednocykliczne

Początkowe wagi nie są dobrze dopasowane do zadania, które próbujemy rozwiązać. Dlatego rozpoczęcie trenowania z wysokim współczynnikiem uczenia jest niebezpieczne: bardzo łatwo możemy sprawić, że trening natychmiast podąży w złym kierunku. Prawdopodobnie nie chcemy też zakończyć trenowania z wysokim współczynnikiem uczenia, więc nie pomijamy minimum. Przez resztę czasu chcielibyśmy jednak używać wysokiego współczynnika uczenia, ponieważ w ten sposób szybciej wykonamy zadanie. Dlatego też powinniśmy zmienić wartość współczynnika uczenia z niskiej na wysoką, a następnie ponownie wrócić do niskiej.

Leslie Smith (tak, to ten sam facet, który wynalazł wyszukiwarkę współczynnika uczenia!) rozwinął ten pomysł w artykule *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates* („Superkonwergencja: bardzo szybkie trenowanie sieci neuronowych przy użyciu wysokich współczynników uczenia" — *https://oreil.ly/EB8NU*). Zaprojektował podzielony na dwie fazy harmonogram dostosowywania współczynnika uczenia. W pierwszej fazie współczynnik rośnie od wartości minimalnej do maksymalnej (*rozgrzewanie*), a w drugiej spada z powrotem do wartości minimalnej (*wyżarzanie*). Smith nazwał to połączenie dwóch faz **trenowaniem jednocyklicznym** (ang. *1cycle training*).

Trenowanie jednocykliczne pozwala na wykorzystanie znacznie wyższych współczynników uczenia niż w przypadku innych metod, co daje dwie korzyści:

- Trenowanie z wyższymi współczynnikami uczenia powoduje, że czas poświęcony na trening skraca się — ten fenomen Smith nazwał superkonwergencją.
- Trenowanie z wyższymi współczynnikami uczenia sprawia, że model jest mniej narażony na nadmierne dopasowanie, ponieważ pomijamy ostre lokalne minima, a kończymy na gładszej (a zatem mającej lepsze możliwości uogólniania) części wykresu funkcji straty.

Kup ksi k Pole ksi k

Drugi punkt jest szczególnie interesujący — opiera się na obserwacji, że w przypadku dobrze uogólniającego modelu strata nie zmieni się zbytnio, jeśli dane wejściowe zmienią się tylko nieznacznie. Jeśli model jest przez dłuższy czas trenowany z wysokim współczynnikiem uczenia i może przy tym zwrócić niezły wynik funkcji straty, musiał zapewne znaleźć obszar, który również dobrze się uogólnia, ponieważ paczki są często zmieniane (jest to w zasadzie definicja wysokiego współczynnika uczenia). Problem polega na tym, że zmiana współczynnika uczenia na wysoki częściej spowoduje pojawienie się kiepskiej wartości straty niż poprawę wyniku. Powinniśmy więc nie przechodzić od razu do wysokiej wartości współczynnika uczenia. Zaczynamy od niskiej, dzięki czemu strata jest poprawna. Następnie pozwalamy optymalizatorowi na stopniowe znajdowanie coraz gładszych obszarów dla parametrów, przechodząc jednocześnie do wyższych wartości współczynnika.

Po znalezieniu gładkiego obszaru dla parametrów będziemy chcieli wyszukać najlepszy jego fragment, co oznacza, że powinniśmy ponownie zmniejszyć wartość współczynnika uczenia. Dlatego też trenowanie jednocykliczne ma fazę rozgrzewania, w której stopniowo następuje zwiększanie wartości tego współczynnika, a następnie fazę wychładzania, w której jego wartość się zmniejsza. Wielu badaczy odkryło, że w praktyce takie podejście prowadzi do dokładniejszych modeli i szybszego trenowania. Dlatego jest ono używane domyślnie w funkcji `fine_tune` z biblioteki *fastai*.

W rozdziale 16. poznasz metodę *momentum* wykorzystywaną przez stochastyczny spadek wzdłuż gradientu. Krótko mówiąc, momentum to technika, dzięki której optymalizator kieruje się nie tylko w stronę gradientów, ale także w kierunku poprzednich kroków. Leslie Smith przedstawił ideę *cyklicznego momentum* w artykule *A Disciplined Approach to Neural Network Hyper-Parameters: Part 1* („Metodyczne podejście do hiperparametrów sieci neuronowych. Część 1." — *https://oreil.ly/oL7GT*). Badacz sugeruje, że wartość współczynnika momentum zmienia się odwrotnie do wartości współczynnika uczenia. Gdy wykorzystujemy wysokie współczynniki uczenia, używamy mniejszego współczynnika momentum. I przeciwnie, większa wartość współczynnika momentum pojawia się ponownie w fazie wyżarzania.

Trenowanie jednocykliczne możemy zastosować przez wywołanie funkcji `fit_one_cycle` z biblioteki *fastai*:

```
def fit(epochs=1, lr=0.06):
    learn = Learner(dls, simple_cnn(), loss_func=F.cross_entropy,
                    metrics=accuracy, cbs=ActivationStats(with_hist=True))
    learn.fit_one_cycle(epochs, lr)
    return learn
learn = fit()
```

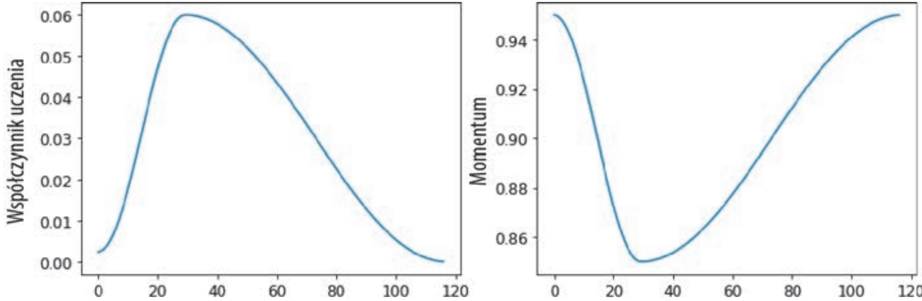
Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	0,210838	0,084827	0,974300	00:08

W końcu jakiś postęp! Osiągnęliśmy całkiem rozsądną dokładność.

W trakcie trenowania możemy wyświetlić współczynnik uczenia i momentum przez wywołanie funkcji `plot_sched` obiektu `learn.recorder`. Obiekt `learn.recorder` (jak sama nazwa wskazuje) rejestruje wszystko, co dzieje się podczas treningu, w tym straty, wskaźniki i hiperparametry, takie jak współczynnik uczenia i momentum:

```
learn.recorder.plot_sched()
```


Kup ksi k Pole ksi k



Smith w pierwotnym artykule o trenowaniu jednocyklicznym opisał rozgrzewanie i wyżarzanie działające zgodnie z funkcją liniową. W bibliotece *fastai* rozszerzyliśmy tę opcję — uzupełniliśmy ją o możliwość wyżarzania według funkcji cosinus. Funkcja `fit_one_cycle` wykorzystuje następujące parametry, które możesz zmodyfikować:

`lr_max`
: Najwyższa wartość współczynnika uczenia, która będzie mogła zostać zastosowana (może to również być lista współczynników uczenia dla każdej grupy warstw lub obiekt `slice` języka Python zawierający współczynniki uczenia dla pierwszej i ostatniej grupy warstw).

`div`
: Dzielnik dla `lr_max`, pozwalający uzyskać początkowy współczynnik uczenia.

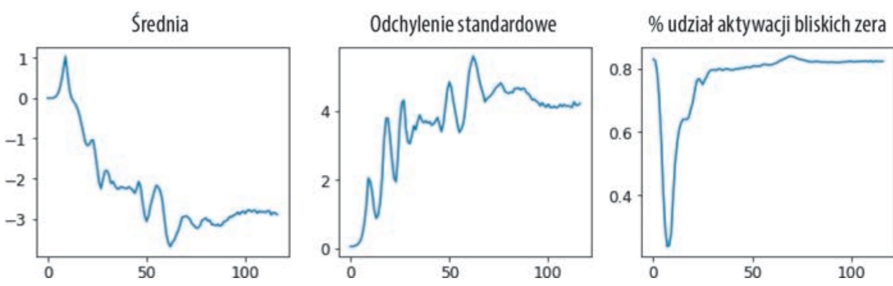
`div_final`
: Dzielnik dla `lr_max`, pozwalający uzyskać końcowy współczynnik uczenia.

`pct_start`
: Odsetek paczek używanych w fazie rozgrzewania.

`moms`
: Krotka (*`mom1`*`,` *`mom2`*`,` *`mom3`*), gdzie *`mom1`* jest początkową wartością momentum, *`mom2`* jest minimalną wartością momentum, a *`mom3`* jest końcową wartością momentum.

Przyjrzyjmy się ponownie statystykom warstw:

```
learn.activation_stats.plot_layer_stats(-2)
```

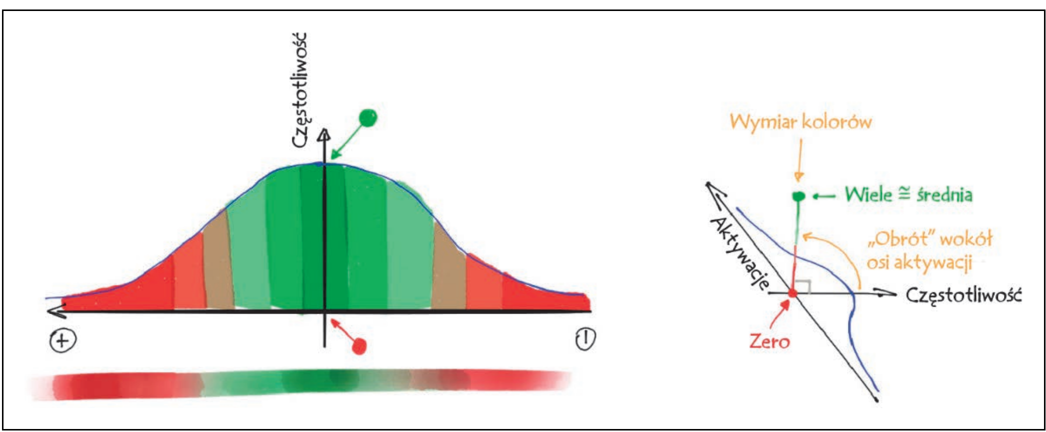


Kup książkę Poleć książkę

Odsetek wag niezerowych znacznie się poprawił, chociaż wciąż jest dość wysoki. Aby poznać jeszcze więcej szczegółów procesu trenowania, użyjmy funkcji `color_dim`, przekazując jej indeks warstw:

```
learn.activation_stats.color_dim(-2)
```

Funkcja `color_dim` została opracowana przez portal fast.ai we współpracy ze studentem Stefano Giomo. Giomo, który nazwał to rozwiązanie *wymiarem kolorów*, udostępnił szczegółowe informacje (*https://oreil.ly/bPXGw*) dotyczące projektowania i działania metody. Podstawowym pomysłem jest stworzenie histogramu aktywacji warstwy, który w przypadku poprawnego trenowania powinien zostać wyświetlony w postaci gładkiej funkcji, takiej jak rozkład normalny (rysunek 13.14).



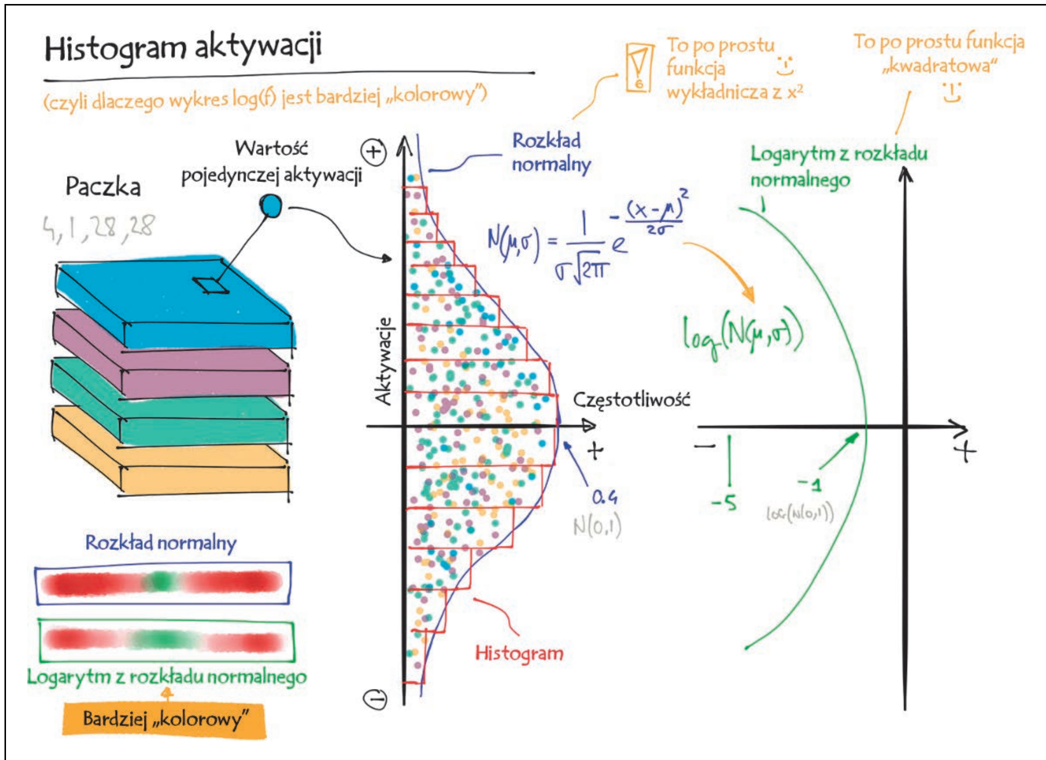
Rysunek 13.14. Histogram w wymiarze kolorów (dzięki uprzejmości Stefano Giomo)

Aby utworzyć wykres za pomocą funkcji `color_dim`, używamy histogramu po lewej stronie i zamieniamy go na reprezentację wykorzystującą kolory, pokazaną na dole. Następnie obracamy go, jak zaprezentowano po prawej stronie rysunku. Okazuje się, że rozkład staje się wyraźniejszy, jeśli wyznaczymy logarytm wartości histogramu. Giomo opisuje:

> Ostateczny wykres dla warstwy jest tworzony przez nałożenie na siebie histogramów aktywacji z każdej paczki wzdłuż osi poziomej. Zatem każdy pionowy wycinek przedstawia histogram aktywacji dla pojedynczej paczki. Intensywność koloru odpowiada wysokości histogramu, innymi słowy — liczbie aktywacji w każdym jego przedziale.

Na rysunku 13.15 pokazano schematycznie całe rozwiązanie.

Kup ksi k   Pole ksi k



Rysunek 13.15. Schemat działania wymiaru kolorów (dzięki uprzejmości Stefano Giomo)

Na rysunku wyjaśniono, dlaczego wykres log(*f*) jest bardziej kolorowy niż wykres samej funkcji *f*, gdy jest ona zgodna z rozkładem normalnym. Chodzi o to, że wyznaczenie logarytmu zmienia krzywą Gaussa w kwadratową, która nie jest tak wąska.

Mając to na uwadze, spójrzmy jeszcze raz na wynik dla warstwy przedostatniej:

```
learn.activation_stats.color_dim(-2)
```

Rysunek prezentuje klasyczny obraz „złego trenowania". Na początku prawie wszystkie aktywacje są równe zero — widać to po lewej stronie rysunku, gdzie dominuje kolor ciemnoniebieski. Jasnożółty wąski pas na dole symbolizuje prawie zerowe aktywacje. Następnie, dla kilku pierwszych paczek, liczba niezerowych aktywacji rośnie wykładniczo. Wykres dąży w złym kierunku i gwałtownie się załamuje! Widzimy, że powraca kolor ciemnoniebieski, a dolna część rysunku ponownie staje się jasnożółta.

Kup książkę Poleć książkę

Wygląda na to, że trenowanie rozpoczyna się od nowa. W dalszej kolejności widzimy, że aktywacje ponownie rosną i znów się załamują. Po kilkukrotnym powtórzeniu cyklu w końcu widzimy, że aktywacje rozprzestrzeniają się w całym zakresie.

O wiele lepiej byłoby, gdyby trenowanie mogło od początku przebiegać płynnie. Cykle wykładniczego wzrostu, a następnie załamania mogą generować wiele niemal zerowych aktywacji, co skutkuje powolnym trenowaniem i słabymi wynikami końcowymi. Jednym z rozwiązań tego problemu jest zastosowanie normalizacji wsadowej.

Normalizacja wsadowa

Aby przyspieszyć proces i poprawić słabe wyniki końcowe, musimy poradzić sobie z początkowym dużym odsetkiem prawie zerowych aktywacji, a następnie postarać się utrzymać poprawny rozkład aktywacji w trakcie trenowania.

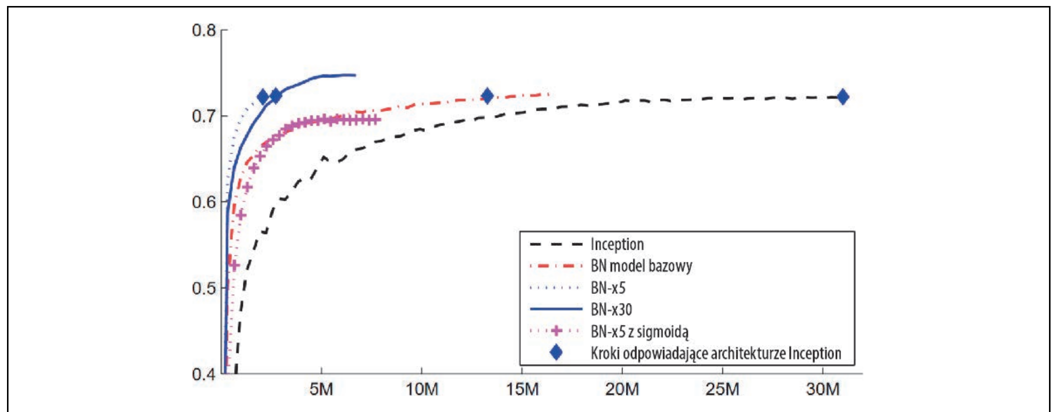
Sergey Ioffe i Christian Szegedy przedstawili odpowiednie rozwiązanie w artykule *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* („Normalizacja wsadowa: przyspieszenie trenowania sieci głębokich przez zmniejszenie wewnętrznego przesunięcia kowariantnego” — *https://oreil.ly/MTZJL*) z 2015 roku. Oto opis problemu, z którym się zetknęliśmy:

> Trenowanie głębokich sieci neuronowych jest skomplikowane, ponieważ rozkład danych wejściowych dla każdej warstwy zmienia się podczas trenowania, gdy zmieniają się parametry warstw poprzednich. Spowalnia to cały proces i wymaga niższych współczynników uczenia oraz starannej inicjalizacji parametrów. Ten problem nazywamy wewnętrznym przesunięciem kowariantnym, a rozwiązujemy go przez normalizację danych wejściowych warstwy.

Oto opis rozwiązania:

> Należy uzupełnić architekturę modelu o normalizację i wykonywać ją dla każdej minipaczki treningowej. Normalizacja wsadowa pozwala używać znacznie wyższych współczynników uczenia. Nie wymaga również nadmiernej ostrożności przy inicjalizacji parametrów.

Artykuł wywołał duże emocje, ponieważ zawierał wykres zaprezentowany na rysunku 13.16. Ten wykres wyraźnie pokazał, że normalizacja wsadowa może wytrenować model dokładniejszy od takiego, jaki można było zaprojektować przy ówczesnym stanie wiedzy (architektura *Inception*), i około pięciu razy szybszy.



Rysunek 13.16. Wpływ użycia normalizacji wsadowej (dzięki uprzejmości Sergeya Ioffe'a i Christiana Szegedyego)

Kup ksi k Pole ksi k

Normalizacja wsadowa (często nazywana *batchnorm*) działa na zasadzie uśrednienia wartości średnich i odchyleń standardowych dla wszystkich aktywacji warstwy i wykorzystania uzyskanych wyników do normalizacji. Może to jednak powodować problemy, ponieważ sieć w celu wykonania dokładnych prognoz może wymagać, by niektóre aktywacje były naprawdę wysokie. Dodano więc dwa parametry, które mają możliwości uczenia (co oznacza, że zostaną zaktualizowane w kroku SGD), zwykle nazywane `gamma` i `beta`. Gdy normalizacja aktywacji w celu uzyskania nowego wektora aktywacji `y` zostaje zakończona, warstwa z normalizacją wsadową zwraca wynik równy `gamma * y + beta`.

Dlatego też aktywacje mogą mieć dowolną wartość średniej lub wariancji, niezależnie od średniej i odchylenia standardowego wyników z poprzedniej warstwy. Statystyki te są uczone oddzielnie, dzięki czemu proces trenowania jest łatwiejszy. Sposób działania modelu jest odmienny w zależności od tego, czy mamy do czynienia z trenowaniem czy walidacją. W fazie trenowania używamy średniej i odchylenia standardowego paczki, aby znormalizować dane, podczas gdy w trakcie walidacji wykorzystujemy średnią z bieżących statystyk wyznaczonych w fazie trenowania.

Uzupełnijmy funkcję `conv` o warstwę z normalizacją wsadową:

```
def conv(ni, nf, ks=3, act=True):
    layers = [nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)]
    layers.append(nn.BatchNorm2d(nf))
    if act: layers.append(nn.ReLU())
    return nn.Sequential(*layers)
```

Następnie dopasujmy model:

```
learn = fit()
```

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	0,130036	0,055021	0,986400	00:10

Osiągnęliśmy naprawdę wyjątkowy wynik! Wywołajmy funkcję `color_dim`:

```
learn.activation_stats.color_dim(-4)
```

Właśnie to chcieliśmy zobaczyć: płynny wzrost aktywacji bez żadnych „sytuacji awaryjnych". Normalizacja wsadowa spełniła pokładane w niej nadzieje! W rzeczywistości metoda ta odniosła taki sukces, że jest (lub coś bardzo podobnego do niej) stosowana w prawie wszystkich nowoczesnych sieciach neuronowych.

Kup ksi k   Pole ksi k

Interesującą kwestią jest to, że modele zawierające warstwy z normalizacją wsadową potrafią lepiej uogólniać niż te, które nie zawierają takich warstw. Chociaż nie spotkaliśmy jeszcze dokładnej analizy tego fenomenu, zdaniem większości badaczy wynika on stąd, że normalizacja wsadowa zwiększa przypadkowość w procesie trenowania. Każda minipaczka różni się od innych średnią i odchyleniem standardowym. Aktywacje zostaną więc za każdym razem znormalizowane przy użyciu innych wartości. Aby model mógł wykonywać dokładne prognozy, będzie musiał się nauczyć, jak stać się odpornym na te zmiany. Zwiększenie przypadkowości w procesie trenowania często jest pomocne.

Ponieważ model działa naprawdę dobrze, wytrenujmy go jeszcze przez kilka epok i zobaczmy, co się stanie. Zwiększmy też współczynnik uczenia, ponieważ w streszczeniu artykułu napisano, że powinniśmy być w stanie „przeprowadzać proces trenowania ze znacznie wyższym współczynnikiem uczenia”:

```
learn = fit(5, lr=0.1)
```

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	0,191731	0,121738	0,960900	00:11
1	0,083739	0,055808	0,981800	00:10
2	0,053161	0,044485	0,987100	00:10
3	0,034433	0,030233	0,990200	00:10
4	0,017646	0,025407	0,991200	00:10

```
learn = fit(5, lr=0.1)
```

Epoka	Strata zbioru treningowego	Strata zbioru walidacyjnego	Dokładność	Czas
0	0,183244	0,084025	0,975800	00:13
1	0,080774	0,067060	0,978800	00:12
2	0,050215	0,062595	0,981300	00:12
3	0,030020	0,030315	0,990700	00:12
4	0,015131	0,025148	0,992100	00:12

W tym momencie można powiedzieć, że potrafimy już rozpoznawać cyfry! Czas przejść do czegoś trudniejszego!

Podsumowanie

Dowiedziałeś się, że konwolucja jest po prostu mnożeniem macierzy, mającym dwa ograniczenia związane z macierzą wag: pewne elementy są zawsze równe zero, a niektóre są powiązane ze sobą (zawsze muszą mieć tę samą wartość). W rozdziale 1. zaprezentowaliśmy osiem wymagań pochodzących z książki *Parallel Distributed Processing*, wydanej w 1986 roku. Jedno z nich dotyczyło „wzorca połączeń między jednostkami”. Właśnie taką rolę odgrywają wspomniane przed chwilą ograniczenia — wymuszają określony wzorzec połączeń.

Kup ksi k Pole ksi k

Pozwalają one również na wykorzystywanie w modelu znacznie mniejszej liczby parametrów bez utraty przez niego zdolności rozpoznawania złożonych cech wizualnych. Oznacza to, że możemy szybciej trenować głębsze modele przy mniejszym prawdopodobieństwie powstania nadmiernego dopasowania. Chociaż uniwersalne twierdzenie aproksymacyjne dowodzi, że w pełni połączona sieć używająca jednej ukrytej warstwy teoretycznie *powinna* rozwiązać każde możliwe zadanie, wiemy, że w *praktyce* możemy wytrenować znacznie lepsze modele, jeśli zdefiniujemy odpowiednie architektury.

Konwolucje są zdecydowanie najpowszechniejszym wzorcem połączeń używanym w sieciach neuronowych (wraz z regularnymi warstwami liniowymi, zwanymi *w pełni połączonymi*). Prawdopodobnie powstanie jednak o wiele więcej takich wzorców.

Dowiedziałeś się również, jak można interpretować aktywacje warstw w sieci, aby sprawdzić, czy trenowanie przebiega poprawnie. Ponadto zobaczyłeś, w jaki sposób normalizacja wsadowa pomaga ustabilizować proces trenowania i sprawia, że staje się płynniejszy. W następnym rozdziale wykorzystamy obie warstwy, które poznałeś, do zbudowania najpopularniejszej architektury komputerowej: sieci ResNet.

Pytania

1. Co to jest cecha?
2. Podaj macierz jądra splotu dla detektora górnych krawędzi.
3. Podaj operację matematyczną stosowaną przez jądro 3×3 na pojedynczym pikselu obrazu.
4. Jaka jest wartość jądra splotu zastosowanego do macierzy zer 3×3?
5. Co to jest dopełnienie?
6. Co to jest krok?
7. Utwórz zagnieżdżoną listę składaną i wykorzystaj ją w dowolnie wybranym przez siebie zadaniu.
8. Jakie są kształty parametrów `input` i `weight` w funkcji konwolucji dwuwymiarowej z biblioteki PyTorch?
9. Co to jest kanał?
10. Jaka jest zależność między konwolucją a mnożeniem macierzy?
11. Co to jest konwolucyjna sieć neuronowa?
12. Jakie korzyści można osiągnąć ze zmodyfikowania części sieci neuronowej?
13. Co to jest funkcja `Flatten()`? W jakim miejscu musi zostać wywołana w przypadku sieci CNN wykorzystującej zbiór MNIST? Uzasadnij odpowiedź.
14. Co oznacza skrót NCHW?
15. Dlaczego trzecia warstwa sieci CNN działającej ze zbiorem MNIST wykorzystuje wzór `7*7*(1168-16)`?

Kup ksi k   Pole ksi k

16. Co to jest pole receptywne?
17. Jaka będzie wielkość pola receptywnego aktywacji po dwóch operacjach konwolucji z krokiem 2? Uzasadnij odpowiedź.
18. Uruchom plik *conv-example.xlsx* i poeksperymentuj ze śledzeniem poprzedników.
19. Zapoznaj się z twitterową listą najnowszych „polubień” Jeremy’ego lub Sylvaina i sprawdź, czy zawiera ona interesujące Cię zasoby lub pomysły.
20. W jaki sposób obraz kolorowy może zostać przedstawiony jako tensor?
21. Jak działa konwolucja z danymi zawierającymi kolory?
22. Jakiej metody moglibyśmy użyć, aby sprawdzić takie dane w obiekcie `DataLoaders`?
23. Dlaczego podwajamy liczbę filtrów po każdej operacji konwolucji z krokiem 2?
24. Dlaczego w przypadku pierwszej konwolucji obsługującej bazę MNIST używamy większego jądra splotu (z wykorzystaniem parametru `simple_cnn`)?
25. Jakie informacje związane z warstwą zapamiętuje wywołanie zwrotne `ActivationStats`?
26. W jaki sposób po przeprowadzeniu trenowania możemy uzyskać dostęp do wywołania zwrotnego obiektu `Learner`?
27. Jakie są trzy rodzaje statystyk kreślonych przez funkcję `plot_layer_stats`? Co przedstawia oś X?
28. Dlaczego aktywacje bliskie zera sprawiają problemy?
29. Jakie są wady i zalety trenowania wykorzystującego paczki o większej wielkości?
30. Dlaczego na początku trenowania powinniśmy unikać wysokiego współczynnika uczenia?
31. Co to jest trenowanie jednocykliczne?
32. Jakie są zalety trenowania z wysokim współczynnikiem uczenia?
33. Dlaczego pod koniec trenowania należy zastosować niski współczynnik uczenia?
34. Co to jest momentum cykliczne?
35. Jakie wywołanie zwrotne rejestruje podczas trenowania wartości hiperparametrów (i inne informacje)?
36. Co przedstawia jedna kolumna pikseli na wykresie `color_dim`?
37. Jak wygląda „złe trenowanie” zaprezentowane za pomocą funkcji `color_dim`? Uzasadnij odpowiedź.
38. Jakie parametry z możliwością trenowania zawiera warstwa normalizacji wsadowej?
39. Jakie statystyki są używane przez normalizację wsadową podczas trenowania? Co się dzieje podczas walidacji?
40. Dlaczego modele z warstwami normalizacji wsadowej lepiej uogólniają?

Kup ksi k Pole ksi k

Dalsze badania

1. Jakie cechy, poza detektorami krawędzi, były wykorzystywane w widzeniu komputerowym (zwłaszcza przed upowszechnieniem się uczenia głębokiego)?
2. W bibliotece PyTorch są dostępne również inne warstwy normalizacji. Przetestuj je i sprawdź, które działają najlepiej z Twoimi modelami. Dowiedz się, dlaczego zaprojektowano takie warstwy i czym różnią się od normalizacji wsadowej.
3. Spróbuj w metodzie `conv` umieścić funkcję aktywacji po warstwie normalizacji wsadowej. Czy dostrzeżesz jakąś różnicę w działaniu modelu? Postaraj się zdobyć wiedzę dotyczącą zalecanej kolejności wykonywania operacji.

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

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

Uczenie głębokie?
Dobrze zrozum, dobrze zastosuj!

Uczenie głębokie zmienia oblicze wielu branż. Ta rewolucja już się zaczęła, jednak potencjał AI i sieci neuronowych jest znacznie większy. Korzystamy więc dziś z osiągnięć komputerowej analizy obrazu i języka naturalnego, wspierania badań naukowych czy budowania skutecznych strategii biznesowych — wchodzimy do świata, który do niedawna był dostępny głównie dla naukowców. W konsekwencji trudno o źródła wiedzy, które równocześnie byłyby przystępne dla zwykłych programistów i miały wysoką wartość merytoryczną. Problem polega na tym, że bez dogłębnego zrozumienia działania algorytmów uczenia głębokiego trudno tworzyć dobre aplikacje.

Oto praktyczny i przystępny przewodnik po koncepcjach uczenia głębokiego, napisany tak, aby ułatwić zrozumienie najnowszych technik w tej dziedzinie bez znajomości wyższej matematyki. Książka daje znakomite podstawy uczenia głębokiego, a następnie stopniowo wprowadza zagadnienia sposobu działania modeli, ich budowy i trenowania. Pokazano w niej również praktyczne techniki przekształcania modeli w działające aplikacje. Znalazło się tu mnóstwo wskazówek ułatwiających poprawianie dokładności, szybkości i niezawodności modeli. Nie zabrakło też informacji o najlepszych sposobach wdrażania od podstaw algorytmów uczenia głębokiego i stosowaniu ich w najnowocześniejszych rozwiązaniach.

W książce między innymi:

- gruntownie i przystępnie omówione podstawy uczenia głębokiego
- najnowsze techniki uczenia głębokiego i ich praktyczne zastosowanie
- działanie modeli oraz zasady ich treningu
- praktyczne tworzenie aplikacji korzystających z uczenia głębokiego
- wdrażanie algorytmów uczenia głębokiego
- etyczne implikacje AI

Jeremy Howard jest przedsiębiorcą, ekspertem, programistą i naukowcem. Wykłada na Uniwersytecie w San Francisco. Inwestował w wiele start-upów, był ich mentorem i doradcą.

Sylvain Gugger jest inżynierem badawczym w Hugging Face. Wcześniej nauczał informatyki i matematyki w ramach programu CPGE. Autor kilku cenionych podręczników.

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7509-3
9 788328 375093

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 129,00 zł