François Chollet
J.J. Allaire

DEEP LEARNING

PRACA Z JĘZYKIEM R I BIBLIOTEKĄ KERAS

Tytuł oryginału: Deep Learning with R

Tłumaczenie: Konrad Matuk

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-4780-9

Original edition copyright © 2018 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2019 by HELION SA.
All rights reserved.

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 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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/delerk.zip

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Przedmowa 9
Podziękowania 11
O książce 13
O autorach 17

CZĘŚĆ I. PODSTAWY UCZENIA GŁĘBOKIEGO 19

Rozdział 1. Czym jest uczenie głębokie? 21

1.1. Sztuczna inteligencja, uczenie maszynowe i uczenie głębokie 22
1.1.1. Sztuczna inteligencja 22
1.1.2. Uczenie maszynowe 22
1.1.3. Formy danych umożliwiające uczenie 24
1.1.4. „Głębia” uczenia głębokiego 26
1.1.5. Działanie uczenia głębokiego przedstawione na trzech rysunkach 27
1.1.6. Co dotychczas osiągnięto za pomocą uczenia głębokiego? 29
1.1.7. Nie wierz w tymczasową popularność 30
1.1.8. Nadzieje na powstanie sztucznej inteligencji 31
1.2. Zanim pojawiło się uczenie głębokie: krótka historia uczenia maszynowego 32
1.2.1. Modelowanie probabilistyczne 32
1.2.2. Wczesne sieci neuronowe 33
1.2.3. Metody jądrowe 33
1.2.4. Drzewa decyzyjne, lasy losowe i gradientowe wzmacnianie maszyn 35
1.2.5. Powrót do sieci neuronowych 35
1.2.6. Co wyróżnia uczenie głębokie? 36
1.2.7. Współczesne uczenie maszynowe 37
1.3. Dlaczego uczenie głębokie? Dlaczego teraz? 38
1.3.1. Sprzęt 38
1.3.2. Dane 39
1.3.3. Algorytmy 40
1.3.4. Nowa fala inwestycji 40
1.3.5. Demokratyzacja uczenia głębokiego 41
1.3.6. Co dalej? 41

Rozdział 2. Matematyczne podstawy sieci neuronowych 43

2.1. Pierwszy przykład sieci neuronowej 44
2.2. Reprezentacja danych sieci neuronowych 47
2.2.1. Skalary (tensory zerowymiarowe) 48
2.2.2. Wektory (tensory jednowymiarowe) 48
2.2.3. Macierze (tensory dwuwymiarowe) 48
2.2.4. Trójwymiarowe tensory i tensory o większej liczbie wymiarów 49

Kup ksi k Pole ksi k

2.2.5. *Główne atrybuty* 49
2.2.6. *Obsługa tensorów R* 50
2.2.7. *Wsad danych* 50
2.2.8. *Prawdziwe przykłady tensorów danych* 51
2.2.9. *Dane wektorowe* 51
2.2.10. *Dane szeregu czasowego i dane sekwencyjne* 52
2.2.11. *Dane w postaci obrazów* 52
2.2.12. *Materiały wideo* 53
2.3. Koła zębate sieci neuronowych: operacje na tensorach 53
2.3.1. *Operacje wykonywane element po elemencie* 54
2.3.2. *Operacje na tensorach o różnych wymiarach* 55
2.3.3. *Iloczyn tensorowy* 55
2.3.4. *Zmiana kształtu tensora* 57
2.3.5. *Geometryczna interpretacja operacji tensorowych* 58
2.3.6. *Interpretacja geometryczna uczenia głębokiego* 59
2.4. Silnik sieci neuronowych: optymalizacja gradientowa 60
2.4.1. *Czym jest pochodna?* 61
2.4.2. *Pochodna operacji tensorowej: gradient* 62
2.4.3. *Stochastyczny spadek wzdłuż gradientu* 63
2.4.4. *Łączenie pochodnych: algorytm propagacji wstecznej* 66
2.5. Ponowna analiza pierwszego przykładu 67
2.6. Podsumowanie rozdziału 68

Rozdział 3. Rozpoczynamy korzystanie z sieci neuronowych 71

3.1. Anatomia sieci neuronowej 72
3.1.1. *Warstwy: podstawowe bloki konstrukcyjne uczenia głębokiego* 72
3.1.2. *Modele: sieci warstw* 73
3.1.3. *Funkcja straty i optymalizatory: najważniejsze elementy konfiguracji procesu uczenia* 74
3.2. Wprowadzenie do pakietu Keras 75
3.2.1. *Keras, TensorFlow, Theano i CNTK* 76
3.2.2. *Instalowanie pakietu Keras* 77
3.2.3. *Praca z pakietem Keras: krótkie wprowadzenie* 77
3.3. Przygotowanie stacji roboczej do uczenia głębokiego 79
3.3.1. *Dwie opcje uruchamiania pakietu Keras* 79
3.3.2. *Wady i zalety uruchamiania uczenia głębokiego w chmurze* 80
3.3.3. *Jaki procesor graficzny najlepiej nadaje się do uczenia głębokiego?* 80
3.4. Przykład klasyfikacji binarnej: klasyfikacja recenzji filmów 81
3.4.1. *Zbiór danych IMDB* 81
3.4.2. *Przygotowywanie danych* 82
3.4.3. *Budowa sieci neuronowej* 83
3.4.4. *Walidacja modelu* 87
3.4.5. *Używanie wytrenowanej sieci do generowania przewidywań dotyczących nowych danych* 90
3.4.6. *Dalsze eksperymenty* 90
3.4.7. *Wnioski* 91
3.5. Przykład klasyfikacji wieloklasowej: klasyfikacja krótkich artykułów prasowych 91
3.5.1. *Zbiór danych Agencji Reutera* 91
3.5.2. *Przygotowywanie danych* 93

Kup ksi k Pole ksi k

3.5.3. Budowanie sieci 93
3.5.4. Walidacja modelu 94
3.5.5. Generowanie przewidywań dotyczących nowych danych 96
3.5.6. Inne sposoby obsługi etykiet i funkcji straty 97
3.5.7. Dlaczego warto tworzyć odpowiednio duże warstwy pośrednie? 97
3.5.8. Dalsze eksperymenty 98
3.5.9. Wnioski 98
3.6. Przykład regresji: przewidywanie cen mieszkań 99
3.6.1. Zbiór cen mieszkań w Bostonie 99
3.6.2. Przygotowywanie danych 100
3.6.3. Budowanie sieci 100
3.6.4. K-składowa walidacja krzyżowa 101
3.6.5. Wnioski 105
3.7. Podsumowanie rozdziału 105

Rozdział 4. Podstawy uczenia maszynowego 107

4.1. Cztery rodzaje uczenia maszynowego 108
4.1.1. Uczenie nadzorowane 108
4.1.2. Uczenie nienadzorowane 108
4.1.3. Uczenie częściowo nadzorowane 109
4.1.4. Uczenie przez wzmacnianie 109
4.2. Ocena modeli uczenia maszynowego 109
4.2.1. Zbiory treningowe, walidacyjne i testowe 111
4.2.2. Rzeczy, o których warto pamiętać 114
4.3. Wstępna obróbka danych, przetwarzanie cech i uczenie cech 114
4.3.1. Przygotowywanie danych do przetwarzania przez sieci neuronowe 115
4.3.2. Przetwarzanie cech 116
4.4. Nadmierne dopasowanie i zbyt słabe dopasowanie 118
4.4.1. Redukcja rozmiaru sieci 119
4.4.2. Dodawanie regularyzacji wag 121
4.4.3. Porzucanie — technika dropout 123
4.5. Uniwersalny przepływ roboczy uczenia maszynowego 125
4.5.1. Definiowanie problemu i przygotowywanie zbioru danych 125
4.5.2. Wybór miary sukcesu 126
4.5.3. Określanie techniki oceny wydajności modelu 127
4.5.4. Przygotowywanie danych 127
4.5.5. Tworzenie modeli lepszych od linii bazowej 128
4.5.6. Skalowanie w górę: tworzenie modelu, który ulega nadmiernemu dopasowaniu 129
4.5.7. Regularyzacja modelu i dostrajanie jego hiperparametrów 129
4.6. Podsumowanie rozdziału 130

CZĘŚĆ II. UCZENIE GŁĘBOKIE W PRAKTYCE 131

Rozdział 5. Uczenie głębokie i przetwarzanie obrazu 133

5.1. Wprowadzenie do konwolucyjnych sieci neuronowych 134
5.1.1. Działanie sieci konwolucyjnej 136
5.1.2. Operacja max-pooling 141

Kup ksi k Pole ksi k

5.2. Trenowanie konwolucyjnej sieci neuronowej na małym zbiorze danych 143
5.2.1. Stosowanie uczenia głębokiego w problemach małych zbiorów danych 144
5.2.2. Pobieranie danych 144
5.2.3. Budowa sieci neuronowej 147
5.2.4. Wstępna obróbka danych 148
5.2.5. Stosowanie techniki augmentacji danych 151
5.3. Korzystanie z wcześniej wytrenowanej konwolucyjnej sieci neuronowej 155
5.3.1. Ekstrakcja cech 155
5.3.2. Dostrajanie 163
5.3.3. Wnioski 167
5.4. Wizualizacja efektów uczenia konwolucyjnych sieci neuronowych 168
5.4.1. Wizualizacja pośrednich aktywacji 169
5.4.2. Wizualizacja filtrów konwolucyjnych sieci neuronowych 175
5.4.3. Wizualizacja map ciepła aktywacji klas 181
5.5. Podsumowanie rozdziału 185

Rozdział 6. Uczenie głębokie w przetwarzaniu tekstu i sekwencji 187

6.1. Praca z danymi tekstowymi 188
6.1.1. Kodowanie słów i znaków metodą gorącej jedynki 189
6.1.2. Osadzanie słów 192
6.1.3. Łączenie wszystkich technik: od surowego tekstu do osadzenia słów 197
6.1.4. Wnioski 203
6.2. Rekurencyjne sieci neuronowe 203
6.2.1. Warstwa rekurencyjna w pakiecie Keras 206
6.2.2. Warstwy LSTM i GRU 209
6.2.3. Przykład warstwy LSTM zaimplementowanej w pakiecie Keras 212
6.2.4. Wnioski 213
6.3. Zaawansowane zastosowania rekurencyjnych sieci neuronowych 214
6.3.1. Problem prognozowania temperatury 214
6.3.2. Przygotowywanie danych 217
6.3.3. Punkt odniesienia w postaci zdrowego rozsądku 220
6.3.4. Podstawowe rozwiązanie problemu przy użyciu techniki uczenia maszynowego 221
6.3.5. Punkt odniesienia w postaci pierwszego modelu rekurencyjnego 223
6.3.6. Stosowanie rekurencyjnego porzucania w celu zmniejszenia nadmiernego dopasowania 225
6.3.7. Tworzenie stosów warstw rekurencyjnych 226
6.3.8. Korzystanie z dwukierunkowych rekurencyjnych sieci neuronowych 228
6.3.9. Kolejne rozwiązania 232
6.3.10. Wnioski 233
6.4. Konwolucyjne sieci neuronowe i przetwarzanie sekwencji 234
6.4.1. Przetwarzanie sekwencji za pomocą jednowymiarowej sieci konwolucyjnej 234
6.4.2. Jednowymiarowe łączenie danych sekwencyjnych 235
6.4.3. Implementacja jednowymiarowej sieci konwolucyjnej 235
6.4.4. Łączenie sieci konwolucyjnych i rekurencyjnych w celu przetworzenia długich sekwencji 237
6.4.5. Wnioski 241
6.5. Podsumowanie rozdziału 242

Kup książkę Poleć książkę

Rozdział 7. Najlepsze zaawansowane praktyki uczenia głębokiego *245*

7.1. Funkcjonalny interfejs programistyczny pakietu Keras: wykraczanie poza model sekwencyjny 246

7.1.1. Wprowadzenie do funkcjonalnego interfejsu API 247

7.1.2. Modele z wieloma wejściami 249

7.1.3. Modele z wieloma wyjściami 251

7.1.4. Skierowane acykliczne grafy warstw 254

7.1.5. Udostępnianie wag warstwy 258

7.1.6. Modele pełniące funkcję warstw 259

7.1.7. Wnioski 260

7.2. Monitorowanie modeli uczenia głębokiego przy użyciu wywołań zwrotnych pakietu Keras i narzędzia TensorBoard 260

7.2.1. Używanie wywołań zwrotnych w celu sterowania procesem trenowania modelu 260

7.2.2. Wprowadzenie do TensorBoard — platformy wizualizacji danych pakietu TensorFlow 264

7.2.3. Wnioski 268

7.3. Korzystanie z pełni możliwości modeli 268

7.3.1. Konstrukcja zaawansowanych architektur 269

7.3.2. Optymalizacja hiperparametru 272

7.3.3. Składanie modeli 274

7.3.4. Wnioski 276

7.4. Podsumowanie rozdziału 276

Rozdział 8. Stosowanie uczenia głębokiego w celu generowania danych *279*

8.1. Generowanie tekstu za pomocą sieci LSTM 281

8.1.1. Krótka historia generatywnych sieci rekurencyjnych 281

8.1.2. Generowanie danych sekwencyjnych 282

8.1.3. Dlaczego strategia próbkowania jest ważna? 282

8.1.4. Implementacja algorytmu LSTM generującego tekst na poziomie liter 285

8.1.5. Wnioski 289

8.2. DeepDream 290

8.2.1. Implementacja algorytmu DeepDream w pakiecie Keras 291

8.2.2. Wnioski 296

8.3. Neuronowy transfer stylu 297

8.3.1. Strata treści 298

8.3.2. Strata stylu 298

8.3.3. Implementacja neuronowego transferu stylu przy użyciu pakietu Keras 299

8.3.4. Wnioski 304

8.4. Generowanie obrazów przy użyciu wariacyjnych autoenkoderów 306

8.4.1. Próbkowanie z niejawnej przestrzeni obrazów 306

8.4.2. Wektory koncepcyjne używane podczas edycji obrazu 307

8.4.3. Wariacyjne autoenkodery 308

8.4.4. Wnioski 314

8.5. Wprowadzenie do generatywnych sieci z przeciwnikiem 315

8.5.1. Schematyczna implementacja sieci GAN 316

8.5.2. Zbiór przydatnych rozwiązań 317

8.5.3. Generator 318

Kup ksi k Pole ksi k

8.5.4. Dyskryminator 319
8.5.5. Sieć z przeciwnikiem 320
8.5.6. Trenowanie sieci DCGAN 320
8.5.7. Wnioski 322
8.6. Podsumowanie rozdziału 323

Rozdział 9. Wnioski 325

9.1. Przypomnienie najważniejszych koncepcji 326
9.1.1. Sztuczna inteligencja 326
9.1.2. Co sprawia, że uczenie głębokie to wyjątkowa dziedzina uczenia maszynowego? 326
9.1.3. Jak należy traktować uczenie głębokie? 327
9.1.4. Najważniejsze technologie 328
9.1.5. Uniwersalny przepływ roboczy uczenia maszynowego 329
9.1.6. Najważniejsze architektury sieci 330
9.1.7. Przestrzeń możliwości 334
9.2. Ograniczenia uczenia głębokiego 336
9.2.1. Ryzyko antropomorfizacji modeli uczenia maszynowego 337
9.2.2. Lokalne uogólnianie a ekstremalne uogólnianie 339
9.2.3. Wnioski 340
9.3. Przyszłość uczenia głębokiego 341
9.3.1. Modele jako programy 342
9.3.2. Wykraczanie poza algorytm propagacji wstecznej i warstwy różniczkowalne 343
9.3.3. Zautomatyzowane uczenie maszynowe 344
9.3.4. Nieustanne uczenie się i wielokrotne używanie modułowych procedur składowych 345
9.3.5. Przewidywania dotyczące dalekiej przyszłości 346
9.4. Bycie na bieżąco z nowościami związanymi z szybko rozwijającą się dziedziną 348
9.4.1. Zdobywaj wiedzę praktyczną, pracując z prawdziwymi problemami przedstawianymi w serwisie Kaggle 348
9.4.2. Czytaj o nowych rozwiązaniach w serwisie arXiv 348
9.4.3. Eksploruj ekosystem związany z pakietem Keras 349
9.5. Ostatnie słowa 349

DODATKI 351

Dodatek A. Instalowanie pakietu Keras i innych bibliotek niezbędnych do jego działania w systemie Ubuntu 353

Dodatek B. Uruchamianie kodu w środowisku RStudio Server przy użyciu zdalnej instancji procesora graficznego EC2 359

Skorowidz 365

Kup ksi k Pole ksi k

8 Stosowanie uczenia głębokiego w celu generowania danych

W tym rozdziale opisałem:

- Generowanie tekstu za pomocą sieci LSTM.
- Implementację narzędzia DeepDream.
- Wykonywanie transferu neuronowego.
- Wariacyjne autoenkodery.
- Generatywne sieci z przeciwnikiem.

Potencjał udawania przez sztuczną inteligencję ludzkiego procesu myślowego wykracza poza pasywne zadania, takie jak rozpoznawanie obiektów, i zadania reakcyjne, takie jak prowadzenie samochodu. Sięga on również do zadań kreatywnych. Kilka lat temu nie wierzono mi, że w niedalekiej przyszłości większość naszej kultury będzie tworzona ze znaczną pomocą sztucznej inteligencji. Nie wierzyły mi nawet osoby od dłuższego czasu zajmujące się uczeniem maszynowym. Prognozę tę sformułowałem w 2014 r. Trzy lata później liczba niedowiarków zaczęła coraz szybciej maleć. Latem 2015 r. zobaczyliśmy efekty pracy algorytmu DeepDream opracowanego przez firmę Google, który bawił nas psychodelicznym obrazem psich oczu i paranoidalnych artefaktów. W 2016 r. pojawiła się aplikacja Prisma zamieniająca zdjęcia w obrazy o różnych stylach. Latem 2016 r. powstał eksperymentalny film krótkometrażowy *Sunspring*, stworzony na podstawie

Kup ksi k Pole ksi k

scenariusza napisanego w pełni (łącznie z dialogami) przez algorytm LSTM. Coraz częściej sieci neuronowe stosuje się do komponowania muzyki.

Oczywiście dzieła tworzone obecnie przez sztuczną inteligencję nie są najwyższych lotów. Sztuczna inteligencja nie potrafi jeszcze zbliżyć się do poziomu ludzi pracujących jako scenarzyści, malarze i kompozytorzy. Zastępowanie ludzi nigdy nie było celem sztucznej inteligencji — nie chcemy zastąpić własnej inteligencji czymś innym, chcemy wnieść do naszego życia i naszej pracy *więcej* inteligentnych rozwiązań. W wielu zadaniach, a szczególnie w zadaniach kreatywnych, sztuczna inteligencja będzie używana w charakterze narzędzia *wspierającego* — będzie raczej nas wspierać, niż zastępować.

Tworzenie dzieła przez artystę w dużej mierze sprowadza się do prostego rozpoznawania wzorców i umiejętności technicznych. Te części procesu twórczego wiele osób uważa za mniej atrakcyjne. Tu właśnie wkracza sztuczna inteligencja. Nasz język i nasza sztuka charakteryzują się statystyczną strukturą. Algorytmy uczenia głębokiego mogą uczyć się takich właśnie struktur. Modele uczenia maszynowego mogę uczyć się **ukrytej przestrzeni** obrazów, muzyki i opowiadań. Następnie mogą generować *próbki* z tych przestrzeni w celu stworzenia nowego dzieła sztuki o charakterze podobnym do tego, co model rozpoznał w treningowym zbiorze danych. Oczywiście takie próbkowanie samo w sobie nie jest niczym kreatywnym. To tylko operacja matematyczna: algorytm nie ma pojęcia o ludzkim życiu, ludzkich emocjach i naszym doświadczaniu świata. Algorytm uczy się z doświadczeń, które bardzo różnią się od naszych. Znaczenie temu, co zostanie wygenerowane przez model, może nadać tylko interpretacja przez człowieka. Dane wygenerowane przez algorytm w rękach uzdolnionego artysty mogą zostać odpowiednio zmodyfikowane i nabrać piękna. Próbkowanie przestrzeni może stać się pędzlem wspomagającym artystę, modyfikować naszą kreatywność i poszerzać wyobraźnię. Algorytmy mogą sprawić, że tworzeniem sztuki zacznie zajmować się szersza grupa osób, ponieważ sztuczna inteligencja eliminuje potrzebę posiadania doświadczenia i umiejętności technicznych — oddziela ekspresję artystyczną od warsztatu.

Iannis Xenakis — wizjoner i pionier muzyki elektronicznej i algorytmicznej, wyraził tę samą myśl w latach 60. w kontekście stosowania automatycznych technologii w komponowaniu muzyki[1].

> *Kompozytor zwolniony z obowiązku wykonywania skrupulatnych obliczeń może poświęcić się pracy nad ogólnymi problemami nowej formy muzycznej i przyglądać się uważnie wszelkim szczegółom tej formy, modyfikując wartości danych wejściowych. Może np. sprawdzić wszystkie kombinacje instrumentalne: od solistów, przez małe orkiestry, do dużych orkiestr. Dzięki pomocy komputerów kompozytor zaczyna działać jak pilot: naciska przyciski, wprowadza współrzędne i nadzoruje trajektorię lotu kosmicznej kapsuły dźwięku przez muzyczne konstelacje i galaktyki, które do niedawna mógł sobie tylko wyobrazić.*

W tym rozdziale opiszę różne potencjalne zastosowania uczenia głębokiego w celach twórczych. Przedstawię zagadnienia związane z sekwencyjnym generowaniem danych (techniki przydatne podczas tworzenia tekstu lub muzyki), algorytmem DeepDream, tworzeniem obrazów przy użyciu wariacyjnych autoenkoderów i generatywnych sieci

---

[1] Iannis Xenakis, *Musiques formelles: nouveaux principes formels de composition musicale*, specjalne wydanie „La Revue musicale", numery 253 – 254, 1963.

Kup ksi k Pole ksi k

z przeciwnikiem. Podczas lektury tego rozdziału dowiesz się, jak sprawić, aby Twój komputer wygenerował nowe treści, i zaczniesz dostrzegać fantastyczne możliwości wynikające z łączenia techniki ze sztuką.

8.1. Generowanie tekstu za pomocą sieci LSTM

W tym podrozdziale opiszę stosowanie rekurencyjnych sieci neuronowych do generowania danych sekwencyjnych. Zrobię to na przykładzie generowania tekstu, ale to samo rozwiązanie może zostać użyte w celu utworzenia dowolnych danych o charakterze sekwencyjnym, takich jak sekwencje dźwięków tworzących muzykę i ruchy pędzlem tworzące obraz (dane tego typu można uzyskać, nagrywając tabletem ruchy dłoni artysty).

Możliwości generowania danych sekwencyjnych nie ograniczają się do tworzenia sztuki. Techniki tego typu stosuje się w syntezowaniu mowy i generowaniu wypowiedzi czatbotów. Udostępniona w 2016 r. przez firmę Google funkcja inteligentnej odpowiedzi, która może automatycznie wygenerować listę szybkich odpowiedzi na e-maile lub wiadomości tekstowe, jest oparta na podobnej technice.

8.1.1. Krótka historia generatywnych sieci rekurencyjnych

Pod koniec 2014 r. mało kto miał do czynienia z sieciami LSTM. Dotyczyło to nawet osób zajmujących się zawodowo uczeniem maszynowym. Skuteczne generowanie danych sekwencyjnych przy użyciu rekurencyjnych sieci neuronowych zaczęło być popularne dopiero w 2016 r. Pomimo tego techniki te mają dość długą historię — za jej początek można uznać opracowanie algorytmu LSTM w 1997 r.[2]. Początkowo algorytm ten stosowano do generowania tekstu znak po znaku.

W 2002 r. Douglas Eck, pracując wówczas w szwajcarskim laboratorium Schmidhubera, zastosował po raz pierwszy algorytm LSTM w celu wygenerowania muzyki. Udało mu się uzyskać obiecujące rezultaty. Eck jest obecnie badaczem w Google Brain, gdzie w 2016 r. założył zespół badawczy Magenta zajmujący się stosowaniem nowoczesnych technik uczenia głębokiego w celu tworzenia chwytliwej muzyki. Jak widać, wdrożenie pomysłu może czasami trwać nawet 15 lat.

Pod koniec lat dwutysięcznych i na początku drugiej dekady XXI w. Alex Graves stał się pionierem stosowania rekurencyjnych sieci neuronowych w celu generowania sekwencyjnych danych. Za punkt zwrotny rozwoju tej techniki uważa się napisanie przez niego pracy na temat stosowania różnych gęstych rekurencyjnych sieci w celu wygenerowania pisma odręcznego przy użyciu danych szeregu czasowego określających położenie długopisu[3]. Właśnie to zastosowanie sieci neuronowych było dla mnie urzeczywistnieniem idei **śniącej maszyny** i stanowiło duże źródło inspiracji, gdy rozpoczynałem pracę nad pakietem Keras. Graves miał podobne spostrzeżenia na ten temat. Świadczy o tym treść komentarza ukrytego w pliku LaTeX, załadowanego na serwer

[2] Sepp Hochreiter i Jürgen Schmidhuber, *Long Short-Term Memory*, „Neural Computation” 9, nr 8, 1997.

[3] Alex Graves, *Generating Sequences With Recurrent Neural Networks*, arXiv, 2013, *https://arxiv.org/abs/1308.0850*.

Kup książkę Poleć książkę

arXiv jako wstępna wersja artykułu: „Generowanie danych sekwencyjnych jest czymś, co maksymalnie zbliża komputery do śnienia". Po upływie kilku lat niektóre z wówczas nowatorskich rozwiązań traktujemy jako coś zwykłego, ale w tamtych czasach trudno było przyglądać się temu, co pokazywał Graves, i nie być oszołomionym ilością nowych możliwości.

Od tego czasu rekurencyjne sieci neuronowe zaczęły być skutecznie stosowane w celu generowania muzyki, obrazów, treści dialogów, syntezy mowy i projektowania molekuł. Użyto ich nawet do wygenerowania scenariusza filmu, który został zrealizowany z udziałem prawdziwej obsady.

8.1.2. Generowanie danych sekwencyjnych

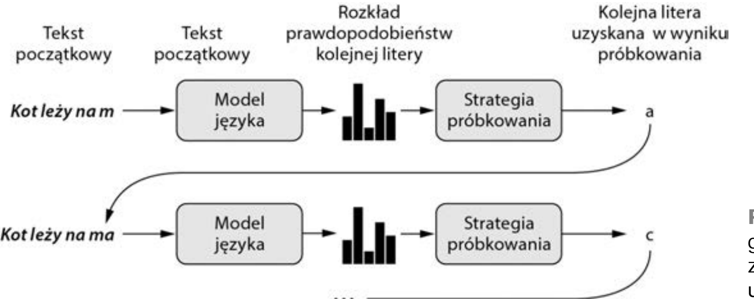
Uniwersalnym sposobem generowania danych sekwencyjnych przy użyciu uczenia głębokiego jest trenowanie sieci (zwykle sieci rekurencyjnej lub konwolucyjnej) w celu przewidywania kolejnego znaku lub kilku kolejnych znaków wchodzących w skład sekwencji na podstawie wcześniejszych znaków pełniących funkcję danych wejściowych. Przykładowo, fraza wejściowa „kot leży na maci" sprawi, że wytrenowana do przewidywania sieć wygeneruje literę „e" jako kolejną literę tej frazy. Pracując z danymi tekstowymi, możemy operować na tokenach mających formę słów lub znaków. Sieć mogącą modelować prawdopodobieństwo kolejnego tokena na podstawie poprzednich tokenów określamy mianem **modelu języka**. Model języka rozpoznaje **ukrytą przestrzeń** języka — jego strukturę statystyczną.

Po wytrenowaniu modelu języka możemy go używać w celu tworzenia **próbek** (generowania nowych sekwencji): do modelu kierowany jest początkowy łańcuch znaków (nazywa się go **danymi warunkującymi**), a model ma wygenerować kolejną literę lub kolejne słowo (możliwe jest nawet jednoczesne generowanie kilku tokenów). Wygenerowane dane wyjściowe są dodawane do danych wejściowych, a cały proces jest powtarzany wielokrotnie (patrz rysunek 8.1). Pętla ta pozwala na generowanie sekwencji o dowolnej długości. Charakter tych sekwencji odzwierciedla strukturę danych, na których model został wytrenowany — sekwencje wyglądają **prawie** tak, jakby zostały napisane przez człowieka. W tej sekcji zaprezentuję przykład, w którym warstwa LSTM na podstawie łańcuchów N znaków wyciągniętych z korpusu tekstu ma przewidywać znak $N + 1$. Na wyjściu modelu będzie znajdować się warstwa softmax określająca prawdopodobieństwa wszystkich liter na umieszczenie w charakterze kolejnego znaku. Warstwa LSTM, z której będę korzystać, określana jest mianem **modelu języka na poziomie liter**.

8.1.3. Dlaczego strategia próbkowania jest ważna?

Podczas generowania tekstu bardzo ważny jest sposób wyboru kolejnego znaku. Naiwną techniką wyboru kolejnego znaku jest tzw. **chciwe próbkowanie** (ang. *greedy sampling*) — zawsze wybierany jest znak o najwyższym prawdopodobieństwie. W praktyce zastosowanie tej techniki prowadzi do uzyskania powtarzalnych, przewidywalnych łańcuchów, które nie wyglądają naturalnie. Lepszym rozwiązaniem, dokonującym mniej oczywistych wyborów, jest proces próbkowania korzystający z mechanizmu losowania

Kup ksi k Pole ksi k



Rysunek 8.1. Proces generowania tekstu znak po znaku przy użyciu modelu języka

wybierającego kolejne znaki na podstawie rozkładu prawdopodobieństwa. Określamy je mianem **próbkowania stochastycznego** (przymiotnik **stochastyczny** w tym kontekście oznacza to samo co przymiotnik **losowy**). Po zastosowaniu tej techniki, jeżeli litera „e" charakteryzowała się prawdopodobieństwem wybrania jako kolejny znak na poziomie równym 0,3, litera ta będzie wybierana w 30% sytuacji. Zwróć uwagę na to, że próbkowanie chciwe może być również rzutowane na rozkład prawdopodobieństwa. Wówczas jeden znak będzie charakteryzował się prawdopodobieństwem równym 1, a pozostałe znaki będą miały zerowe wartości prawdopodobieństwa.

Probabilistyczne próbkowanie wyjściowe z warstwy softmax modelu jest dobrym rozwiązaniem, ponieważ pozwala na pojawienie się czasem nawet mniej prawdopodobnych znaków, co umożliwia uzyskanie bardziej interesujących zdań i wykazanie pewnej kreatywności poprzez tworzenie nowych realistycznie brzmiących słów, które nie wystąpiły w treningowym zbiorze danych. Strategia ta ma jedną wadę: nie oferuje żadnego sposobu *kontrolowania tego, jak bardzo losowy* jest proces próbkowania.

Dlaczego mielibyśmy chcieć zwiększać lub zmniejszać losowość? Przyjrzyjmy się ekstremalnemu przykładowi próbkowania o charakterze czysto losowym (kolejny znak jest wybierany z jednolitego rozkładu prawdopodobieństw — każda litera charakteryzuje się taką samą wartością prawdopodobieństwa). To najbardziej losowy scenariusz — rozkład prawdopodobieństwa charakteryzuje się maksymalną entropią. Oczywiście nie doprowadzi on do uzyskania niczego ciekawego. Z drugiej strony inny ekstremalny przypadek — próbkowanie chciwe — również nie przyczynia się do wygenerowania niczego ciekawego (nie ma tutaj żadnej losowości, a rozkład prawdopodobieństwa charakteryzuje się minimalną entropią). Próbkowanie z „realnego" rozkładu prawdopodobieństwa (rozkład generowany na wyjściu modelu przez funkcję `softmax`) stanowi rozwiązanie pośrednie, leży dokładnie na środku między dwoma zaprezentowanymi ekstremami. Oczywiście istnieje wiele innych punktów pośrednich (o wyższej lub niższej entropii), którym warto się przyjrzeć. Zmniejszenie entropii sprawi, że wygenerowane struktury będą charakteryzowały się bardziej przewidywalną strukturą, a więc potencjalnie mogą wyglądać bardziej realistycznie. Zmniejszanie entropii sprawi, że uzyskamy bardziej zaskakujące i kreatywne sekwencje. Gdy wykonuje się operację próbkowania z generatywnych modeli, zawsze warto sprawdzić wpływ różnej ilości czynnika losowego na proces generowania. Ostatecznie to my (ludzie) oceniamy to, jak bardzo interesujące są wygenerowane dane, a więc jest to bardzo subiektywne i nie można z góry ocenić optymalnego poziomu entropii.

W celu kontrolowania tego, jak bardzo losowy jest charakter procesu losowania, wprowadzimy parametr określany mianem **temperatury softmax**. Parametr ten charakteryzuje entropię rozkładu prawdopodobieństwa używanego podczas próbkowania — określa to, jak bardzo zaskakujące lub przewidywalne będą wybory kolejnych liter. Na podstawie wartości temperatury (`temperature`) poprzez zmiany wag obliczany jest nowy rozkład prawdopodobieństwa (przetwarzamy rozkład poprzedni wygenerowany przez warstwę wyjściową softmax modelu). Proces ten przebiega w następujący sposób:

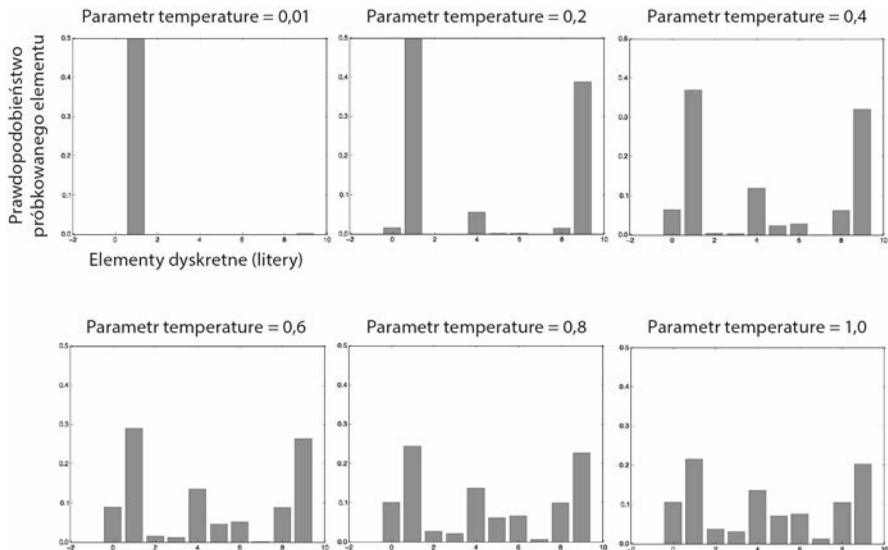
Listing 8.1. Zmiany wag rozkładu prawdopodobieństwa w celu uzyskania innej temperatury softmax

Rozkład original_distribution jest jednowymiarowym wektorem zawierającym wartości prawdopodobieństwa, których suma jest równa 1. Współczynnik temperature określa entropię rozkładu wyjściowego.

```
reweight_distribution <- function(original_distribution,
                                  temperature = 0.5) {
  distribution <- log(original_distribution) / temperature
  distribution <- exp(distribution)
  distribution / sum(distribution)
)
```

Zwracana jest zmodyfikowana wersja rozkładu. Suma wartości poszczególnych prawdopodobieństw nie musi być równa 1, a więc w celu uzyskania prawidłowych wartości należy przeprowadzić operację dzielenia wartości przez ich sumę.

Większe wartości parametru `temperature` prowadzą do uzyskania rozkładu próbkowania o wyższej entropii, a więc generowane będą bardziej zaskakujące dane. Mniejsze wartości tego parametru zmniejszają losowość i pozwalają na uzyskanie bardziej przewidywalnych danych (patrz rysunek 8.2).



Rysunek 8.2. Ten sam rozkład prawdopodobieństwa z różnymi wagami; zmniejszenie parametru temperature powoduje większą przewidywalność; zwiększenie parametru temperature zwiększa losowość

Kup ksi k Pole ksi k

8.1.4. Implementacja algorytmu LSTM generującego tekst na poziomie liter

Czas skorzystać z pakietu Keras i zastosować teorię w praktyce. Na początek będziemy potrzebować dużo danych tekstowych do trenowania modelu języka. Możemy skorzystać z dowolnego wystarczająco rozbudowanego zestawu plików tekstowych — Wikipedii, *Władcy pierścieni* itd. W zaprezentowanym przykładzie posłużymy się wybranymi dziełami Friedricha Nietzschego — niemieckiego filozofa żyjącego w XIX w. — przetłumaczonymi na język angielski. W związku z tym wytrenujemy model odwzorowujący specyficzny styl pisania Nietzschego. Ponadto model ten będzie generował teksty tylko na wybrane tematy — nie będzie to ogólny model języka angielskiego.

PRZYGOTOWANIE DANYCH

Zacznijmy od pobrania korpusu i zapisania go przy użyciu tylko małych liter.

Listing 8.2. Pobieranie i parsowanie pliku tekstowego

```
library(keras)
library(stringr)

path <- get_file(
  "nietzsche.txt",
  origin = "https://s3.amazonaws.com/text-datasets/nietzsche.txt"
)
text <- tolower(readChar(path, file.info(path)$size))
cat("Długość korpusu:", nchar(text), "\n")
```

Teraz dokonamy wyodrębnienia częściowo zachodzących na siebie sekwencji o długości `maxlen`, zakodujemy je techniką kodowania z gorącą jedynką, a następnie umieścimy je w trójwymiarowej tablicy o kształcie `x` (`sequences`, `maxlen`, `unique_characters`). Jednocześnie przygotujemy tablicę `y` zawierającą „wartości docelowe", które w tym przypadku są po prostu literami umieszczanymi po każdej z wyodrębnionych sekwencji. Wartości te zostaną zapisane przy użyciu techniki kodowania z gorącą jedynką.

Listing 8.3. Zapis sekwencji liter za pomocą wektorów

```
maxlen <- 60

step <- 3

text_indexes <- seq(1, nchar(text) - maxlen, by = step)
sentences <- str_sub(text, text_indexes, text_indexes + maxlen - 1)
next_chars <- str_sub(text, text_indexes + maxlen, text_indexes + maxlen)

cat("Liczba sekwencji: ", length(sentences), "\n")

chars <- unique(sort(strsplit(text, "")[[1]]))
cat("Liczba unikatowych znaków:", length(chars), "\n")
char_indices <- 1:length(chars)
names(char_indices) <- chars
```

Wyodrębniamy sekwencje składające się z 60 znaków. (maxlen)

Nowa sekwencja jest próbkowana co 3 znaki. (step)

Zmienna, w której zapisywane będą wyodrębnione sekwencje. (sentences)

Zmienna, w której zapisywane będą kolejne znaki (cele). (next_chars)

Lista unikatowych znaków wchodzących w skład korpusu. (chars)

Słownik przypisujący unikatowe znaki do ich indeksów. (char_indices)

Kup książkę Poleć książkę

```
cat("Tworzenie wektorów...\n")
x <- array(0L, dim = c(length(sentences), maxlen, length(chars)))
y <- array(0L, dim = c(length(sentences), length(chars)))
for (i in 1:length(sentences)) {
  sentence <- strsplit(sentences[[i]], "")[[1]]
  for (t in 1:length(sentence)) {
    char <- sentence[[t]]
    x[i, t, char_indices[[char]]] <- 1
  }
  next_char <- next_chars[[i]]
  y[i, char_indices[[next_char]]] <- 1
}
```

Znaki są zapisywane w formie tablic binarnych przy użyciu kodowania z gorącą jedynką.

BUDOWANIE SIECI

Sieć składa się z pojedynczej warstwy `layer_lstm` i klasyfikatora `dense` z funkcją aktywacji `softmax`. Pamiętajmy o tym, że generowanie danych sekwencyjnych nie musi być przeprowadzane przy użyciu rekurencyjnych sieci neuronowych. Ostatnio coraz częściej stosuje się w tym celu jednowymiarowe sieci konwolucyjne.

Listing 8.4. Model jednowarstwowej sieci LSTM przewidujący kolejny znak

```
model <- keras_model_sequential() %>%
  layer_lstm(units = 128, input_shape = c(maxlen, length(chars))) %>%
  layer_dense(units = length(chars), activation = "softmax")
```

Wartości docelowe (znaki) są zakodowane przy użyciu techniki gorącej jedynki, a więc funkcją straty trenowanego modelu będzie `categorical_crossentropy`.

Listing 8.5. Konfiguracja kompilacji modelu

```
optimizer <- optimizer_rmsprop(lr = 0.01)

model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer
)
```

TRENOWANIE MODELU JĘZYKA I PRÓBKOWANIE Z NIEGO

Dysponując wytrenowanym modelem i kawałkiem początkowego tekstu, możemy wygenerować nowy tekst. W tym celu należy powtarzać następujące operacje:

1. Użyj modelu w celu wygenerowania rozkładu prawdopodobieństwa następnego znaku kontynuującego obecny tekst.
2. Zmodyfikuj rozkład, korzystając z określonej wartości parametru `temperature`.
3. Przeprowadź operację losowego próbkowania następnego znaku na podstawie zmodyfikowanego rozkładu.
4. Dodaj nowy znak na końcu obecnego tekstu.

Oto kod używany do zmiany wag rozkładu prawdopodobieństwa wygenerowanego przez model. Kod ten tworzy **funkcję próbkującą**, która również określa indeks znaku.

Kup ksi k Pole ksi k

Listing 8.6. Funkcja próbkująca kolejny znak na podstawie przewidywań modelu

```
sample_next_char <- function(preds, temperature = 1.0) {
  preds <- as.numeric(preds)
  preds <- log(preds) / temperature
  exp_preds <- exp(preds)
  preds <- exp_preds / sum(exp_preds)
  which.max(t(rmultinom(1, 1, preds)))
}
```

Na koniec poniższa pętla wykonuje operację trenowania modelu i generowania tekstu. Wygenerujemy teksty przy różnych wartościach parametru `temperature` (wartości te będą zmieniane przy rozpoczęciu kolejnych epok procesu trenowania). Pozwoli to nam zobaczyć, jak zmienia się tekst wraz z udoskonalaniem modelu, a także to, jak parametr `temperature` wpływa na strategię próbkowania.

Listing 8.7. Pętla generująca tekst

```
for (epoch in 1:60) {     <- Model będzie trenowany przez 60 epok.

  cat("epoch", epoch, "\n")

  model %>% fit(x, y, batch_size = 128, epochs = 1)     <- Dopasowywanie modelu.

  start_index <- sample(1:(nchar(text) - maxlen - 1), 1)                  | Losowanie tekstu
  seed_text <- str_sub(text, start_index, start_index + maxlen - 1)       | początkowego.

  cat("--- Generowanie przy użyciu tekstu początkowego:", seed_text, "\n\n")

  for (temperature in c(0.2, 0.5, 1.0, 1.2)) {     <- Sprawdzanie różnych parametrów
                                                      temperature procesu próbkowania.
    cat("------ Wartość parametru temperature:", temperature, "\n")
    cat(seed_text, "\n")

    generated_text <- seed_text
                                  Generowanie 400 znaków (proces rozpoczyna się
    for (i in 1:400) {     <-     od wylosowanego tekstu początkowego).

      sampled <- array(0, dim = c(1, maxlen, length(chars)))   |
      generated_chars <- strsplit(generated_text, "")[[1]]     |
      for (t in 1:length(generated_chars)) {                   | Jedna iteracja trenowania
        char <- generated_chars[[t]]                           | modelu na dostępnych danych
        sampled[1, t, char_indices[[char]]] <- 1               | treningowych.
      }                                                        |

      preds <- model %>% predict(sampled, verbose = 0)             |
      next_index <- sample_next_char(preds[1,], temperature)       | Próbkowanie kolejnego znaku.
      next_char <- chars[[next_index]]                             |

      generated_text <- paste0(generated_text, next_char)
      generated_text <- substring(generated_text, 2)

      cat(next_char)
    }
    cat("\n\n")
  }
}
```

Kup książkę Poleć książkę

Oto tekst wygenerowany przy użyciu wylosowanej frazy *new faculty, and the jubilation reached its climax when kant*. Został on uzyskany po 20 epokach trenowania algorytmu — na długo przed osiągnięciem końca procesu trenowania — przy parametrze `temperature` przyjmującym wartość równą 0,2:

```
new faculty, and the jubilation reached its climax when kant and such a man
in the same time the spirit of the surely and the such the such
as a man is the sunligh and subject the present to the superiority of the
special pain the most man and strange the subjection of the
special conscience the special and nature and such men the subjection of the
special men, the most surely the subjection of the special
intellect of the subjection of the same things and
```

Oto wynik przy parametrze `temperature` przyjmującym wartość równą 0,5:

```
new faculty, and the jubilation reached its climax when kant in the eterned
and such man as it's also become himself the condition of the
experience of off the basis the superiory and the special morty of the
strength, in the langus, as which the same time life and "even who
discless the mankind, with a subject and fact all you have to be the stand
and lave no comes a troveration of the man and surely the
conscience the superiority, and when one must be w
```

A to wynik przy parametrze `temperature` przyjmującym wartość 1,0:

```
new faculty, and the jubilation reached its climax when kant, as a
periliting of manner to all definites and transpects it it so
hicable and ont him artiar resull
too such as if ever the proping to makes as cnecience. to been juden,
all every could coldiciousnike hother aw passife, the plies like
which might thiod was account, indifferent germin, that everythery
certain destrution, intellect into the deteriorablen origin of moralian,
and a lessority o
```

Po 60 epokach proces trenowania można uznać za praktycznie zakończony — tekst generowany przez model zaczyna wyglądać na coraz bardziej składny. Oto tekst wygenerowany przy parametrze `temperature` przyjmującym wartość 0,2:

```
cheerfulness, friendliness and kindness of a heart are the sense of the
spirit is a man with the sense of the sense of the world of the
self-end and self-concerning the subjection of the strengthorixes--the
subjection of the subjection of the subjection of the
self-concerning the feelings in the superiority in the subjection of the
subjection of the spirit isn't to be a man of the sense of the
subjection and said to the strength of the sense of the
```

A to tekst wygenerowany przy parametrze `temperature` przyjmującym wartość 0,5:

```
cheerfulness, friendliness and kindness of a heart are the part of the soul
who have been the art of the philosophers, and which the one
won't say, which is it the higher the and with religion of the frences.
the life of the spirit among the most continuess of the
strengther of the sense the conscience of men of precisely before enough
presumption, and can mankind, and something the conceptions, the
subjection of the sense and suffering and the
```

Kup ksi k Pole ksi k

Przy parametrze temperature przyjmującym wartość 1,0 wygenerowany został następujący tekst:

```
cheerfulness, friendliness and kindness of a heart are spiritual by the
ciuture for the
entalled is, he astraged, or errors to our you idstood--and it needs,
to think by spars to whole the amvives of the newoatly, prefectly
raals! it was
name, for example but voludd atu-especity"--or rank onee, or even all
"solett increessic of the world and
implussional tragedy experience, transf, or insiderar,--must hast
if desires of the strubction is be stronges
```

Jak widać, niska wartość parametru temperature prowadzi do uzyskania tekstu, który charakteryzuje się dużą przewidywalnością i powtarzalnością, ale jego lokalna struktura jest bardzo realistyczna — wszystkie wygenerowane słowa (**słowo** jest lokalnym wzorcem składającym się ze znaków) występują w języku angielskim. Przy wyższych wartościach parametru temperature wygenerowany tekst staje się bardziej interesujący, zaskakujący, a nawet kreatywny — algorytm czasami wymyśla nawet nowe słowa, które brzmią tak, jakby były naprawdę istniejącymi słowami (są to np. *eterned* i *troveration*), ale lokalna struktura tekstu zaczyna się załamywać i większość słów wygląda tak, jakby była prawie losowym zbiorem znaków. Bez wątpienia najciekawsze efekty w przypadku tego generowania tekstu uzyskuje się przy parametrze temperature równym 0,5. Zawsze warto eksperymentować z różnymi strategiami próbkowania! Dobra równowaga między wytrenowaną strukturą a losowością sprawi, że wygenerowany tekst będzie interesujący.

Trenując model dłużej, tworząc większy model i stosując większy zbiór danych, można generować próbki, które wyglądają o wiele składniej i bardziej realistycznie. Oczywiście nie należy oczekiwać od modelu wygenerowania tekstu, który będzie miał jakiś większy sens — mechanizm generujący tekst tylko próbkuje litery z modelu statystycznego określającego ich kolejność. Język jest kanałem komunikacji, a rozmowy dotyczące różnych tematów charakteryzują się inną strukturą statystyczną. Tezę tę można udowodnić, odpowiadając sobie na pytanie: co, jeżeli język ludzki zostałby skompresowany tak, jak kompresowana jest większość cyfrowej komunikacji między komputerami? Wówczas język przenosiłby tyle samo informacji, ale nie charakteryzowałby się żadną ukrytą strukturą statystyczną, co uniemożliwiłoby wytrenowanie modelu języka w sposób, w jaki zrobiliśmy to przed chwilą.

8.1.5. Wnioski

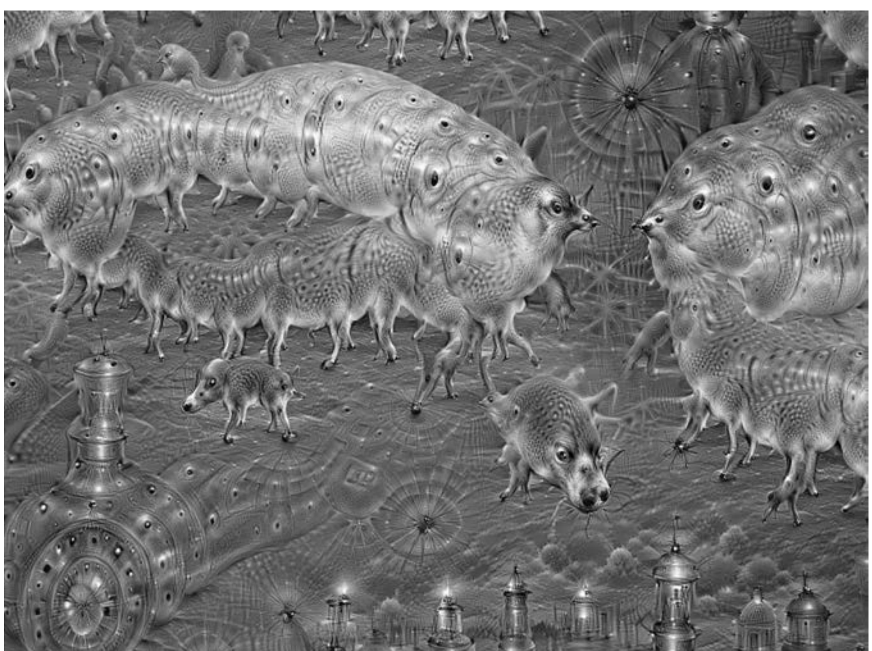
- Dyskretna sekwencja danych może zostać wygenerowana poprzez trenowanie modelu pod kątem przewidywania kolejnych elementów tekstu na podstawie wcześniejszego ciągu znaków.
- Model trenowany na zbiorze danych tekstowych określany jest mianem **modelu języka**. Może on być oparty na słowach lub literach.
- Próbkowanie zbioru elementów tekstu wymaga kompromisu między bezkrytycznym przyjmowaniem przewidywań modelu a losowością.

Kup ksi k Pole ksi k

- Można to zrobić przy użyciu parametru `temperature` funkcji `softmax`. Wybór właściwej wartości tego parametru powinien zostać dokonany na drodze eksperymentów.

8.2. DeepDream

DeepDream to technika artystycznego modyfikowania obrazów, która korzysta z reprezentacji wyuczonych przez konwolucyjne sieci neuronowe. Została ona upubliczniona przez firmę Google w 2015 r. w postaci implementacji napisanej z wykorzystaniem biblioteki uczenia głębokiego o nazwie Caffe (miało to miejsce kilka miesięcy przed opublikowaniem pierwszej wersji pakietu TensorFlow)[4]. Rozwiązanie to bardzo szybko wywołało sensację z powodu generowania obrazów przypominających narkotyczne wizje (patrz przykład przedstawiony na rysunku 8.3) — grafiki te były pełne różnych artefaktów, psich oczu, piór ptaków itd., co wynikało z tego, że sieć konwolucyjna DeepDream była trenowana na zbiorze ImageNet, w którym zdjęć psów i ptaków jest więcej od zdjęć innych zwierząt i przedmiotów.



Rysunek 8.3. Przykładowy obraz wygenerowany przez algorytm DeepDream

Algorytm DeepDream jest bardzo podobny do techniki filtrowania wizualizacji przy użyciu konwolucyjnej sieci neuronowej przedstawionej w rozdziale 5. Rozwiązanie to składało się z sieci konwolucyjnej działającej odwrotnie — dane wejściowe były prze-

[4] Alexander Mordvintsev, Christopher Olah i Mike Tyka, *DeepDream: A Code Example for Visualizing Neural Networks*, Google Research Blog, 1 lipca 2015 r., *http://mng.bz/xXlM*.

Kup ksi k Pole ksi k

twarzane przez algorytm wzrostu gradientowego w celu maksymalizacji aktywacji wybranego filtra górnej warstwy konwolucyjnej sieci neuronowej. Twórcy algorytmu DeepDream skorzystali z tego samego rozwiązania, ale opracowany przez nich algorytm charakteryzuje się pewnymi różnicami:

- W algorytmie DeepDream próbuje się maksymalizować aktywację całej warstwy, a nie wybranego filtra, co powoduje jednoczesne mieszanie się wizualizacji wielu cech.
- Generowanie obrazu nie rozpoczyna się od pustego, nieco zaszumianego obrazu wejściowego — na wejściu przetwarzany jest gotowy obraz, co powoduje nanoszenie efektów na utworzony wcześniej obraz — elementy tego obrazu są zniekształcane w sposób artystyczny.
- Obrazy wejściowe są przetwarzane przy różnych skalach określanych mianem **oktaw**, co ma na celu poprawienie jakości wizualizacji.

Spróbujmy samodzielnie wygenerować jakieś ciekawe obrazy.

8.2.1. Implementacja algorytmu DeepDream w pakiecie Keras

Zaczniemy od konwolucyjnej sieci neuronowej wytrenowanej na zbiorze obrazów ImageNet. Pakiet Keras zawiera wiele takich sieci. Są to między innymi: VGG16, VGG19, Xception i ResNet50. Algorytm DeepDream może zostać zaimplementowany przy użyciu każdej z tych sieci, ale wybór sieci będzie miał oczywiście wpływ na generowane wizualizacje. Wynika to z tego, że różne architektury sieci konwolucyjnych uczą się różnych cech. W oryginalnym algorytmie DeepDream zastosowano model Inception. Wykorzystanie tego modelu pozwala na wygenerowanie ładnie wyglądających grafik, a więc skorzystamy z modelu Inception V3 dołączonego do pakietu Keras.

Listing 8.8. Ładowanie wytrenowanego modelu Inception V3

```
library(keras)

K.set_learning_phase(0)

model <- application_inception_v3(
  weights = "imagenet",
  include_top = FALSE,
)
```

K.set_learning_phase(0): **Nie będziemy trenować modelu. Polecenie to wyłącza wszystkie operacje używane tylko podczas trenowania.**

application_inception_v3(...): **Sieć Inception V3 jest budowana bez swojej konwolucyjnej bazy. Model zostanie załadowany z wagami wytrenowanymi na zbiorze ImageNet.**

Następnie musimy zająć się obliczaniem **straty** — wartości, którą będziemy starali się maksymalizować w procesie wzrostu gradientu. W rozdziale 5. podczas filtrowania wizualizacji staraliśmy się maksymalizować wartość określonego filtra wybranej warstwy sieci. Tym razem będziemy jednocześnie maksymalizować aktywacje wszystkich filtrów wielu warstw, a konkretnie rzecz biorąc, będziemy maksymalizować sumę normy L2 aktywacji zbioru warstw wysokiego poziomu. Wybór warstw (a także dokładanie się poszczególnych warstw do finalnej wartości straty) ma największy wpływ na generowane wizualizacje. W związku z tym chcemy, aby parametry te można było z łatwością modyfikować. Niższe warstwy odpowiadają za wzorce geometryczne,

Kup książkę Poleć książkę

a wyższe warstwy odpowiadają za elementy obrazu pozwalające na rozpoznawanie klas zbioru ImageNet (np. ptaków lub psów). Zaczniemy od niezbyt optymalnej konfiguracji czterech warstw, ale z pewnością warto wypróbować później działanie wielu innych konfiguracji.

Listing 8.9. Konfiguracja algorytmu DeepDream

```
layer_contributions <- list(
  mixed2 = 0.2,
  mixed3 = 3,
  mixed4 = 2,
  mixed5 = 1.5
)
```

Lista z nazwami przypisująca nazwy warstw do współczynników wpływu aktywacji warstw na wartość straty, którą chcemy maksymalizować. Zauważ, że nazwy warstw są wprowadzone na stałe w wbudowanej aplikacji Inception V3. Listę nazw wszystkich warstw modelu można wyświetlić za pomocą polecenia summary(model).

Teraz czas zdefiniować tensor zawierający wartość straty: ważoną sumę normy L2 aktywacji warstw z listingu 8.9.

Listing 8.10. Definiowanie mechanizmu maksymalizującego wartość straty

```
layer_dict <- model$layers
names(layer_dict) <- lapply(layer_dict, function(layer) layer$name)

loss <- k_variable(0)
for (layer_name in names(layer_contributions)) {
  coeff <- layer_contributions[[layer_name]]
  activation <- layer_dict[[layer_name]]$output
  scaling <- k_prod(k_cast(k_shape(activation), "float32"))
  loss <- loss + (coeff * k_sum(k_square(activation)) / scaling)
}
```

Tworzy listę przypisującą nazwy warstw do instancji warstw.

Strata będzie definiowana przez dodanie wartości charakteryzujących wpływ poszczególnych warstw na stratę.

Przechwytuje wyjście warstwy.

Dodaje normę L2 cech warstwy do straty.

Teraz możemy uruchomić proces wzrostu gradientu.

Listing 8.11. Proces wzrostu gradientu

```
dream <- model$input

grads <- k_gradients(loss, dream)[[1]]

grads <- grads / k_maximum(k_mean(k_abs(grads)), 1e-7)

outputs <- list(loss, grads)
fetch_loss_and_grads <- k_function(list(dream), outputs)

eval_loss_and_grads <- function(x) {
  outs <- fetch_loss_and_grads(list(x))
  loss_value <- outs[[1]]
  grad_values <- outs[[2]]
  list(loss_value, grad_values)
}
gradient_ascent <- function(x, iterations, step, max_loss = NULL) {
  for (i in 1:iterations) {
    c(loss_value, grad_values) %<-% eval_loss_and_grads(x)
    if (!is.null(max_loss) && loss_value > max_loss)
```

W tym tensorze znajduje się wygenerowany obraz (wizja).

Oblicza gradienty wizji na podstawie wartości straty.

Normalizuje gradienty (to ważny zabieg).

Konfiguruje funkcję Keras służącą do uzyskiwania wartości straty i gradientów na podstawie obrazu wejściowego.

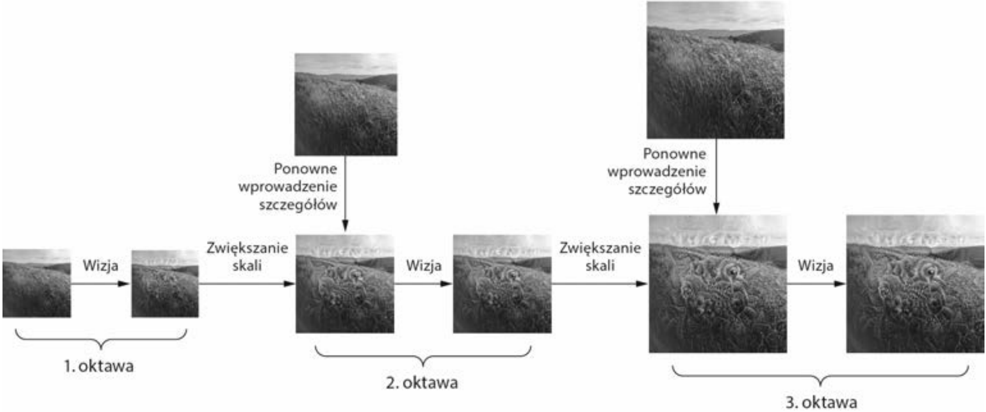
Funkcja wykonująca iteracje procesu wzrostu gradientu.

Kup książkę Poleć książkę

```
    break
    cat("...Wartość straty", i, ":", loss_value, "\n")
    x <- x + (step * grad_values)
  }
  x
}
```

Funkcja wykonująca iteracje procesu wzrostu gradientu.

Na koniec możemy zająć się właściwym algorytmem DeepDream. Na początku definiowana jest lista **skal** (nazywanych również **oktawami**), które są używane podczas przetwarzania obrazów. Każda kolejna skala jest większa od poprzedniej o współczynnik równy 1,4 (jest o 40% większa) — zaczynamy od przetwarzania małego obrazu, a następnie zwiększamy jego skalę (patrz rysunek 8.4).



Rysunek 8.4. Działanie algorytmu DeepDream: następujące po sobie operacje skalowania przestrzennego (oktawy) i dodawania szczegółów

Po każdej kolejnej operacji skalowania (od najmniejszej do największej) uruchamiany jest algorytm wzrostu gradientu w celu maksymalizacji zdefiniowanej wcześniej straty przy danej skali. Po każdym zakończeniu pracy tego algorytmu skala obrazu jest zwiększana o 40%.

W celu uniknięcia utraty dużej ilości szczegółów obrazu po każdej operacji skalowania (w wyniku tych operacji otrzymywany jest coraz bardziej rozmyty i rozpikselowany obraz) możemy wykonać prosty zabieg polegający na ponownym dodaniu utraconych szczegółów do obrazu. Jest to możliwe do wykonania, ponieważ wiemy, jak powinien wyglądać oryginalny obraz w większej rozdzielczości. Dysponując obrazem S o małym rozmiarze i obrazem L o większym rozmiarze, możemy przekształcić obraz L do rozmiaru obrazu S i określić różnice między tymi obrazami — różnica ta będzie wskazywać utracone szczegóły.

Listing 8.12. Określanie wzrostu gradientu przy kolejnych operacjach skalowania obrazu

```
step <- 0.01
num_octave <- 3
```

Rozmiar kroku algorytmu wzrostu gradientu.

Liczba operacji skalowania, przy których należy uruchomić algorytm wzrostu gradientu.

Modyfikacja tych parametrów pozwala na uzyskanie innych efektów wizualnych.

```
octave_scale <- 1.4
iterations <- 20
max_loss <- 10.
base_image_path <- '…'

img <- preprocess_image(base_image_path)


original_shape <- dim(img)[-1]
successive_shapes <- list(original_shape)
for (i in 1:num_octave) {
  shape <- as.integer(original_shape / (octave_scale ^ i))
  successive_shapes[[length(successive_shapes) + 1]] <- shape
}

successive_shapes <- rev(successive_shapes)

original_img <- img
shrunk_original_img <- resize_img(img, successive_shapes[[1]])

 for (shape in successive_shapes) {
  cat("Zmiana kształtu obrazu", shape, "\n")
  img <- resize_img(img, shape)
  img <- gradient_ascent(img,
                         iterations = iterations,
                         step = step,
                         max_loss = max_loss)

  upscaled_shrunk_original_img <- resize_img(shrunk_original_img, shape)
  same_size_original <- resize_img(original_img, shape)
  lost_detail <- same_size_original - upscaled_shrunk_original_img


  img <- img + lost_detail
  shrunk_original_img <- resize_img(original_img, shape)
  save_img(img, fname = sprintf("dream/at_scale_%s.png",
                                paste(shape, collapse = "x")))
}
```

Różnica między rozmiarami kolejnych wersji obrazu.

Liczba kroków wzrostu wykonywanych przy każdej operacji skalowania.

Modyfikacja tych parametrów pozwala na uzyskanie innych efektów wizualnych.

Jeżeli strata przekroczy wartość równą 10, to proces wzrostu gradientu zostanie przerwany w celu zapobiegnięcia powstawania brzydkich artefaktów.

Tu należy umieścić ścieżkę obrazu, który chcemy przetwarzać.

Ładowanie obrazu do tablicy (funkcję tę zdefiniowano w listingu 8.13).

Przygotowywanie listy kształtów definiujących skalowania, przy których uruchomiony zostanie algorytm wzrostu gradientu.

Odwracanie listy kształtów tak, aby znalazły się w kolejności rosnącej.

Zmiana rozmiaru tablicy obrazu w celu zmniejszenia jego skali.

Zwiększanie skali generowanego obrazu.

Po wygenerowaniu obrazu uruchamiany jest algorytm wzrostu gradientu.

Przeskalowywanie mniejszej wersji oryginalnego obrazu (obraz ten będzie rozpikselowany).

Tworzenie wersji oryginalnego obrazu o tym rozmiarze, mającej wysoką jakość.

Przywracanie szczegółów.

Różnica między tymi obrazami określa szczegóły utracone podczas zwiększania skali obrazu.

W kodzie algorytmu zwiększania gradientu zastosowano poniższe funkcje pomocnicze.

Listing 8.13. Funkcje pomocnicze

```
resize_img <- function(img, size) {
  image_array_resize(img, size[[1]], size[[2]])
}

save_img <- function(img, fname) {
  img <- deprocess_image(img)
  image_array_save(img, fname)
```

```
preprocess_image <- function(image_path) {
  image_load(image_path) %>%
    image_to_array() %>%
    array_reshape(dim = c(1, dim(.))) %>%
    inception_v3_preprocess_input()
}

deprocess_image <- function(img) {
  img <- array_reshape(img, dim = c(dim(img)[[2]], dim(img)[[3]], 3))
  img <- img / 2
  img <- img + 0.5
  img <- img * 255

  dims <- dim(img)
  img <- pmax(0, pmin(img, 255))
  dim(img) <- dims
  img
}
```

Funkcja narzędziowa używana podczas otwierania obrazu, zmiany jego rozdzielczości i zapisywania go w formie tensora, który może zostać przetworzony przez sieć Inception V3.

Funkcja narzędziowa konwertująca tensor do postaci właściwego obrazu.

Cofa obróbkę wstępną dokonaną przez funkcję imagenet_preprocess_input.

UWAGA Oryginalna wersja sieci Inception V3 była trenowana pod kątem rozpoznawania elementów obrazów o rozdzielczości 299×299, a w naszym procesie zmniejszamy rozdzielczość obrazu o określony współczynnik, tak więc zaprezentowana implementacja algorytmu DeepDream daje najlepsze efekty podczas pracy z obrazami o rozdzielczości znajdującej się w zakresie od 300×300 do 400×400. Pomimo tego możesz spróbować użyć tego kodu do przetworzenia obrazu o dowolnej rozdzielczości i dowolnych proporcjach.

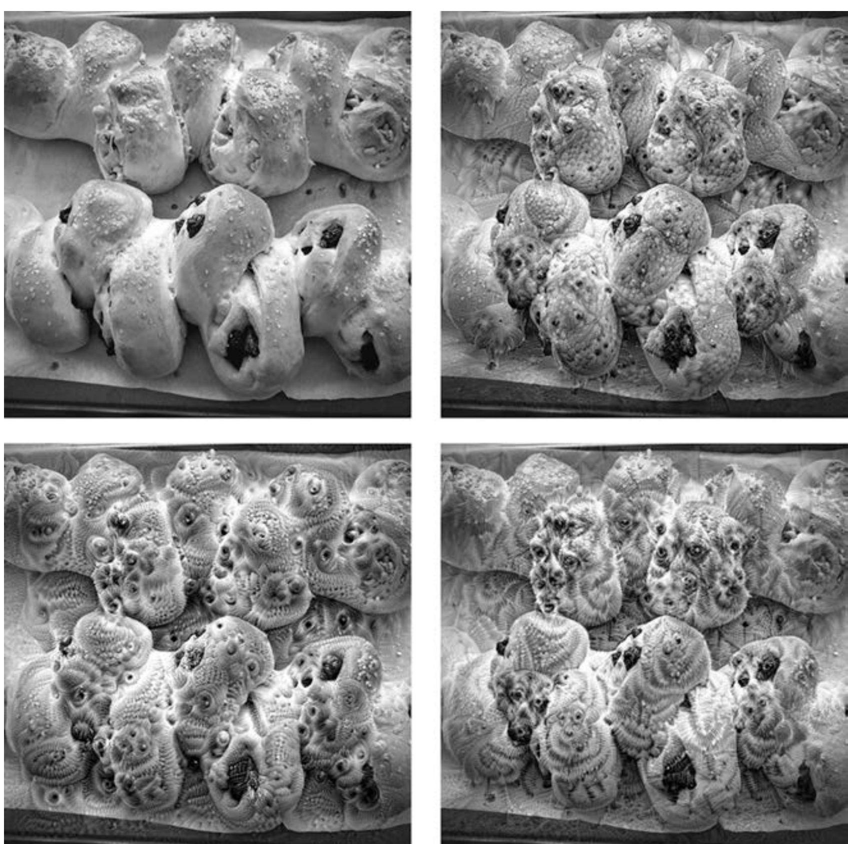
Na rysunku 8.5 pokazano efekt przetwarzania zdjęcia wykonanego na wzgórzach pomiędzy zatoką San Francisco a kampusem firmy Google.

Rysunek 8.5. Przetwarzanie przykładowego obrazu za pomocą algorytmu DeepDream

Polecam samodzielną zabawę tym algorytmem poprzez zmianę warstw, które są używane podczas obliczania straty. Niższe warstwy sieci zawierają bardziej lokalne, mniej abstrakcyjne reprezentacje danych i prowadzą do generowania wzorów, które bardziej przypominają kształty geometryczne. Wyższe warstwy sieci natomiast prowadzą do

Kup książkę Poleć książkę

generowania bardziej rozpoznawalnych wzorców wizualnych przypominających najpopularniejsze obiekty zbioru ImageNet, takie jak oczy psa czy ptasie pióra. W celu szybkiego sprawdzenia efektów różnych kombinacji warstw możesz skorzystać z mechanizmu losowego generowania parametrów słownika `layer_contributions`. Na rysunku 8.6 przedstawiono efekty przetwarzania obrazu domowych wypieków uzyskane z wykorzystaniem różnych konfiguracji warstw sieci neuronowej.



Rysunek 8.6. Przetwarzanie przykładowego obrazu przez różne konfiguracje algorytmu DeepDream

8.2.2. Wnioski

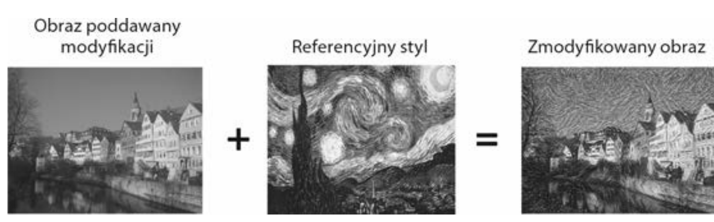
- Algorytm DeepDream składa się z sieci konwolucyjnej działającej na odwrót. Sieć ta generuje obrazy wejściowe na podstawie wyuczonych przez nią reprezentacji danych.
- Uzyskane obrazy są ciekawe i przypominają wizje generowane przez ośrodek wzroku człowieka, który zażył środki psychodeliczne.
- Możliwości tego procesu nie ograniczają się do przetwarzania grafiki i korzystania z sieci konwolucyjnych. Można go użyć także w celu zniekształcania np. mowy lub muzyki.

Kup ksi k Pole ksi k

8.3. Neuronowy transfer stylu

Kolejnym ważnym rozwiązaniem modyfikującym obrazy, opartym na technologii uczenia głębokiego, jest **neuronowy transfer stylu** opracowany latem 2015 r. przez zespół kierowany przez Leona Gatysego[5]. Algorytm neuronowego transferu stylu od tego czasu był wielokrotnie usprawniany i modyfikowany. Zastosowano go w wielu aplikacjach pozwalających na edycję zdjęć przy użyciu smartfona. Dla uproszczenia skupimy się na oryginalnej wersji tego algorytmu.

Neuronowy transfer stylu polega na zastosowaniu stylu obrazu referencyjnego w celu przetworzenia innego obrazu z zachowaniem jego zawartości. Przykład tego procesu pokazano na rysunku 8.7.



Rysunek 8.7. Przykład transferu stylu

Pojęcie **stylu** odnosi się do tekstur, kolorów i sposobu przedstawiania rzeczy widocznych na obrazie. **Treścią** określamy wysokopoziomową makrostrukturę obrazu. Niebieskie i żółte linie narysowane pędzlem widoczne na rysunku 8.7 (obraz *Gwiaździsta noc*, namalowany przez Vincenta van Gogha) charakteryzują styl, a budynki widoczne na zdjęciu Tübingen są treścią.

Idea transferu stylu powiązana z generowaniem tekstur była znana w środowisku osób zajmujących się przetwarzaniem obrazu na długo przed pojawieniem się w 2015 r. neuronowego transferu stylu, ale transfer stylu oparty na technikach uczenia głębokiego okazał się dawać o wiele lepsze rezultaty od tych, które uzyskiwano z zastosowaniem klasycznych technik przetwarzania obrazu. Ta nowatorska technika zyskała wiele kreatywnych zastosowań.

Implementacja transferu stylu jest oparta na tych samych rozwiązaniach, co wszystkie algorytmy uczenia głębokiego — definiujemy w niej funkcję straty i staramy się ją zminimalizować. Celem algorytmu jest zachowanie treści oryginalnego obrazu przy jednoczesnym przyjęciu stylu obrazu referencyjnego. Gdybyśmy mogli matematycznie zdefiniować treść (`content`) i styl (`style`), to wówczas funkcja straty (`loss`) miałaby następującą postać:

```
loss <- distance(style(reference_image) - style(generated_image)) +
        distance(content(original_image) - content(generated_image))
```

`Distance` (odległość) jest funkcją normy takiej jak norma L2, `content` jest funkcją przyjmującą obraz i generującą reprezentację jego treści, a `style` — funkcją przyjmującą obraz i obliczającą reprezentację jego stylu. Minimalizacja straty sprawia, że wartość zwracana przez funkcję `style(generated_image)` zbliża się do wartości zwracanej przez

[5] Leon A. Gatys, Alexander S. Ecker i Matthias Bethge, *A Neural Algorithm of Artistic Style*, arXiv, 2015, *https://arxiv.org/abs/1508.06576*.

`style(reference_image)`, a `content(generated_image)` zbliża się do `content(original_image)`, co prowadzi do zdefiniowanego wcześniej transferu stylu.

Głównym spostrzeżeniem Gatysa i jego zespołu było to, że głębokie konwolucyjne sieci neuronowe umożliwiają matematyczne zdefiniowanie funkcji `style` i `content`. Sprawdźmy, jak do tego dochodzi.

8.3.1. Strata treści

Przypominam, że aktywacje wcześniejszych warstw sieci zawierają **lokalne** informacje o obrazie, a aktywacje wyższych warstw zawierają coraz bardziej **globalne** i **abstrakcyjne** informacje. W związku z tym można przyjąć, że aktywacje różnych warstw konwolucyjnej sieci zawierają rozkład treści obrazu przeprowadzony według różnych przestrzennych skal, a więc treść obrazu, która jest bardziej globalna i abstrakcyjna, powinna być opisywana przez reprezentacje górnych warstw sieci konwolucyjnej.

Dobrym kandydatem na funkcję straty treści jest norma L2 pomiędzy aktywacjami górnej warstwy uprzednio wytrenowanej sieci neuronowej, obliczona przy użyciu przetwarzanego obrazu i aktywacji tej samej warstwy określonych z zastosowaniem wygenerowanego obrazu. Rozwiązanie takie gwarantuje to, że z punktu widzenia górnej warstwy wygenerowany obraz będzie wyglądał podobnie do oryginalnego obrazu, oczywiście przy założeniu, że górne warstwy konwolucyjnej sieci neuronowej naprawdę „widzą" treść obrazów wejściowych. Wówczas rozwiązanie takie pozwoli na zachowanie treści obrazu.

8.3.2. Strata stylu

Mechanizm obliczający stratę treści korzysta tylko z jednej górnej warstwy, a mechanizm obliczający stratę stylu według Gatysa korzysta z wielu warstw sieci konwolucyjnej — próbujemy wziąć pod uwagę styl referencyjnego obrazu, który jest rozsiany po wszystkich przestrzennych skalach sieci konwolucyjnej. Gates, określając stratę stylu, korzysta z **macierzy Grama** składającej się z aktywacji warstw — iloczynu skalarnego map cech danej warstwy. Iloczyn skalarny może być rozumiany jako reprezentacja mapy korelacji między cechami warstwy. Korelacje cech określają parametry statystyczne wzorców poszczególnych skal przestrzennych, co empirycznie odpowiada wyglądowi tekstur skal.

Mechanizm obliczający stratę stylu próbuje zachować podobne do siebie wewnętrzne korelacje wewnątrz aktywacji różnych warstw między stylem obrazu referencyjnego a stylem obrazu wygenerowanego. Rozwiązanie to sprawia, że tekstury znalezione w różnych przestrzennych skalach obrazu referencyjnego mają podobny styl do tych, które występują w wygenerowanym obrazie.

Ogólnie rzecz biorąc, możemy skorzystać z wytrenowanej wcześniej konwolucyjnej sieci neuronowej w celu zdefiniowania funkcji straty, która:

- Zachowa treść poprzez utrzymanie podobnych wysokopoziomowych warstw aktywacji treści przetwarzanego obrazu i wygenerowanego obrazu. Sieć konwolucyjna powinna „postrzegać" oba te obrazy tak, jakby przedstawiały to samo.

- Zachowa styl, utrzymując podobne **korelacje** aktywacji warstw niskiego poziomu, a także warstw wysokiego poziomu. Korelacje cech odwzorowują **tekstury** — wygenerowany obraz i obraz referencyjny powinny charakteryzować się takimi samymi teksturami na różnych przestrzennych skalach.

Przyjrzyjmy się implementacji w Keras oryginalnego algorytmu neuronowego transferu stylu opracowanego w 2015 r. Rozwiązanie to jest pod wieloma względami podobne do implementacji algorytmu DeepDream zaprezentowanej w poprzednim podrozdziale.

8.3.3. Implementacja neuronowego transferu stylu przy użyciu pakietu Keras

Neuronowy transfer stylu może zostać zaimplementowany przy użyciu dowolnej wytrenowanej wcześniej konwolucyjnej sieci neuronowej. Skorzystamy z sieci VGG19 — tej samej, z której korzystał również Gates. VGG19 jest prostą wersją sieci VGG16 opisanej w rozdziale 5. Sieć ta zawiera trzy dodatkowe warstwy konwolucyjne.

Oto czynności, które musisz wykonać:

1. Skonfiguruj sieć obliczającą jednocześnie aktywacje warstw VGG19 obrazu referencyjnego, obrazu przetwarzanego i obrazu wyjściowego.
2. Skorzystaj z obliczonych aktywacji warstw wszystkich trzech obrazów w celu zdefiniowania opisanej wcześniej funkcji straty, którą będziesz minimalizować w celu osiągnięcia transferu stylu.
3. Skonfiguruj algorytm spadku gradientowego w celu zminimalizowania funkcji straty.

Zacznijmy od zdefiniowania ścieżek obrazu referencyjnego i obrazu, który ma zostać przetworzony. Transfer przebiega łatwiej w przypadku obrazów o zbliżonych rozmiarach, a więc później zmienimy rozmiary obrazów tak, aby wszystkie miały wysokość 400 pikseli.

Listing 8.14. Początkowe definiowanie zmiennych

```
library(keras)

target_image_path <- 'img/portrait.jpg'   # Ścieżka obrazu, który ma zostać zmodyfikowany.

style_reference_image_path <- "img/transfer_style_reference.jpg"   # Ścieżka obrazu referencyjnego.

img <- image_load(target_image_path)   # Wymiary wygenerowanego obrazu.
width <- img$size[[1]]
height <- img$size[[2]]
img_nrows <- 400
img_ncols <- as.integer(width * img_nrows / height)
```

Potrzebujemy jeszcze funkcji pomocniczych służących do ładowania, przetwarzania wstępnego i przetwarzania końcowego obrazów wejściowych i wyjściowych sieci konwolucyjnej VGG19.

Listing 8.15. Funkcje pomocnicze

```
preprocess_image <- function(path) {
  img <- image_load(path, target_size = c(img_nrows, img_ncols)) %>%
    image_to_array() %>%
    array_reshape(c(1, dim(.)))
  imagenet_preprocess_input(img)
}

eprocess_image <- function(x) {
  x <- x[1,,,]
  x[,,1] <- x[,,1] + 103.939
  x[,,2] <- x[,,2] + 116.779
  x[,,3] <- x[,,3] + 123.68
  x <- x[,,c(3,2,1)]
  x[x > 255] <- 255
  x[x < 0] <- 0
  x[] <- as.integer(x)/255
  x
}
```

Wyśrodkowywanie w punkcie zerowym poprzez usunięcie średniej wartości pikseli zbioru ImageNet. Proces ten odwraca transformację przeprowadzoną przez funkcję imagenet_preprocess_input.

Konwersja z systemu BGR na system RGB. Proces ten jest częścią odwracania działania modułu vgg19.preprocess_input.

Czas skonfigurować sieć VGG19. Na wejściu przyjmuje ona wsad składający się z trzech obrazów: obrazu referencyjnego, obrazu przetwarzanego i obrazu zastępczego, który zostanie wypełniony wygenerowaną grafiką. Obraz zastępczy jest symbolicznym tensorem, którego wartości są ustalane z zewnątrz przy użyciu tablic. Obraz referencyjny i obraz przetwarzany mają charakter statyczny, a więc definiuje się je przy użyciu polecenia `k_constant`. Dane obrazu zastępczego będą z czasem wypełniane danymi obrazu generowanego.

Listing 8.16. Ładowanie wytrenowanej wcześniej sieci VGG19 i kierowanie do niej trzech obrazów

```
target_image <- k_constant(preprocess_image(target_image_path))
style_reference_image <- k_constant(
  preprocess_image(style_reference_image_path)
)

combination_image <- k_placeholder(c(1, img_nrows, img_ncols, 3))

 input_tensor <- k_concatenate(list(target_image, style_reference_image,
                                  combination_image), axis = 1)

 model <- application_vgg19(input_tensor = input_tensor,
                            weights = "imagenet",
                            include_top = FALSE)

cat("Model został załadowany.\n")
```

Obiekt zastępczy, który zostanie wypełniony danymi wygenerowanego obrazu.

Wsad składający się z trzech obrazów.

Budowanie sieci VGG19 z danymi wejściowymi w postaci wsadu składającego się z trzech obrazów. Model zostanie załadowany z wagami wytrenowanymi wcześniej na podstawie zbioru ImageNet.

Czas zdefiniować stratę treści, dzięki której górna warstwa sieci konwolucyjnej VGG19 będzie postrzegać w podobny sposób obraz przetwarzany i obraz generowany.

Kup ksi k Pole ksi k

Listing 8.17. Strata treści

```
content_loss <- function(base, combination) {
  k_sum(k_square(combination - base))
}
```

Teraz możemy zdefiniować stratę stylu. Podczas jej obliczania korzystamy z funkcji pomocniczej tworzącej macierz Grama — mapę korelacji cech początkowej macierzy.

Listing 8.18. Strata stylu

```
gram_matrix <- function(x) {
  features <- k_batch_flatten(k_permute_dimensions(x, c(3, 1, 2)))
  gram <- k_dot(features, k_transpose(features))
  gram
}

style_loss <- function(style, combination){
  S <- gram_matrix(style)
  C <- gram_matrix(combination)
  channels <- 3
  size <- img_nrows*img_ncols
  k_sum(k_square(S - C)) / (4 * channels^2  * size^2)
}
```

Do tych dwóch elementów straty należy dodać trzeci — **całkowitą stratę wariacji**. Parametr ten odwołuje się do pikseli wygenerowanego obrazu i ułatwia uzyskanie przestrzennej ciągłości tego obrazu, zapobiegając jego całkowitemu rozpikselowaniu. Można go traktować jako stratę regularyzacji.

Listing 8.19. Całkowita strata wariacji

```
total_variation_loss <- function(x) {
  y_ij  <- x[,1:(img_nrows - 1L), 1:(img_ncols - 1L),]
  y_i1j <- x[,2:(img_nrows), 1:(img_ncols - 1L),]
  y_ij1 <- x[,1:(img_nrows - 1L), 2:(img_ncols),]
  a <- k_square(y_ij - y_i1j)
  b <- k_square(y_ij - y_ij1)
  k_sum(k_pow(a + b, 1.25))
}
```

Minimalizowana strata będzie średnią ważoną tych trzech strat. W celu obliczenia straty treści musimy odwołać się tylko do jednej górnej warstwy modelu (warstwy `block5_conv2`), ale podczas obliczania strat stylu musimy korzystać z listy warstw, na której znajdują się warstwy niskiego oraz wysokiego poziomu. Na koniec tego procesu należy pamiętać o dodaniu całkowitej straty wariacji.

Najprawdopodobniej podczas samodzielnej pracy odczujesz chęć dostrojenia współczynnika `content_weight` pod kątem wybranych obrazów (referencyjnego i przetwarzanego). Współczynnik ten określa wpływ straty treści na całkowitą wartość straty. Wyższa wartość `content_weight` sprawi, że w wygenerowanym obrazie łatwiej będzie dostrzec zawartość przetwarzanego obrazu.

Kup ksi k Pole ksi k

Listing 8.20. Definiowanie minimalizowanej, ostatecznej wartości straty

```
outputs_dict <- lapply(model$layers, `[[`, "output")
names(outputs_dict) <- lapply(model$layers, `[[`, "name")

content_layer <- 'block5_conv2'
style_layers = c("block1_conv1", "block2_conv1",
                 "block3_conv1", "block4_conv1",
                 "block5_conv1")
total_variation_weight <- 1e-4
style_weight <- 1.
content_weight <- 0.025

loss <- K.variable(0.)

layer_features <- outputs_dict[[content_layer]]
target_image_features <- layer_features[1,,,]
combination_features <- layer_features[3,,,]
loss <- loss + content_weight * content_loss(target_image_features,
                                             combination_features)

for (layer_name in style_layers){
  layer_features <- outputs_dict[[layer_name]]
  style_reference_features <- layer_features[2,,,]
  combination_features <- layer_features[3,,,]
  sl <- style_loss(style_reference_features, combination_features)
  loss <- loss + ((style_weight / length(style_layers)) * sl)
}

loss <- loss +
  (total_variation_weight * total_variation_loss(combination_image))
```

Lista przypisująca nazwy warstw do tensorów aktywacji.

Warstwa używana podczas obliczania straty treści.

Warstwy używane podczas obliczania straty stylu.

Wagi używane podczas obliczania średniej ważonej (całkowitej wartości straty).

Strata jest definiowana przez dodawanie wszystkich komponentów do tej zmiennej skalarnej.

Dodawanie straty treści.

Dla każdej warstwy dodawany jest komponent straty stylu.

Dodawanie całkowitej straty wariacji.

Na koniec musimy skonfigurować algorytm spadku gradientowego. W pracy Gatysa optymalizacja jest przeprowadzana przy użyciu algorytmu L-BFGS. Skorzystamy z tego samego rozwiązania. To jedna z ważniejszych różnic między tą implementacją neuronowego transferu stylu a implementacją techniki DeepDream opisaną w poprzednim podrozdziale. Implementacja algorytmu L-BFGS jest w postaci funkcji optim(), ale charakteryzuje się dwoma ograniczeniami:

- Wymaga przekazania wartości funkcji straty i wartości gradientów w formie dwóch oddzielnych funkcji.
- Można jej używać tylko do przetwarzania płaskich wektorów, a my mamy do przetworzenia trójwymiarową tablicę reprezentującą obraz.

Obliczanie wartości funkcji straty i wartości gradientów w sposób niezależny byłoby niewydajne, ponieważ wiązałoby się z koniecznością wykonywania wielu zbędnych obliczeń — taki proces przebiegałby praktycznie dwukrotnie wolniej od procesu jednoczesnego obliczania tych wartości. W celu obejścia tego ograniczenia skorzystamy z klasy R6 o nazwie `Evaluator`, która jednocześnie oblicza wartość straty i wartości gradientów, a następnie przy pierwszym wywołaniu zwraca wartość straty, a przy drugim — wartości gradientów.

Listing 8.21. Konfiguracja algorytmu spadku gradientowego

```
grads <- k_gradients(loss, combination_image)[[1]]

fetch_loss_and_grads <-
      k_function(list(combination_image), list(loss, grads))

 eval_loss_and_grads <- function(image) {
  image <- array_reshape(image, c(1, img_nrows, img_ncols, 3))
  outs <- fetch_loss_and_grads(list(image))
  list(
    loss_value = outs[[1]],
    grad_values = array_reshape(outs[[2]], dim = length(outs[[2]]))
  )
}

library(R6)
Evaluator <- R6Class("Evaluator",
  public = list(

    loss_value = NULL,
    grad_values = NULL,

    initialize = function() {
      self$loss_value <- NULL
      self$grad_values <- NULL
    },

    loss = function(x){
      loss_and_grad <- eval_loss_and_grads(x)
      self$loss_value <- loss_and_grad$loss_value
      self$grad_values <- loss_and_grad$grad_values
      self$loss_value
    },

    grads = function(x){
      grad_values <- self$grad_values
      self$loss_value <- NULL
      self$grad_values <- NULL
      grad_values
    }
  )
)

evaluator <- Evaluator$new()
```

Ustalanie wartości gradientów wygenerowanego obrazu.

Funkcja przechwytująca bieżące wartości straty i gradientów.

Klasa obudowująca funkcję fetch_loss_and_grads w sposób umożliwiający uzyskanie wartości strat i gradientów za pomocą wywołań dwóch oddzielnych metod, co jest wymagane przez wybrany optymalizator, którego będziemy używać.

Na koniec uruchamiamy proces wzrostu gradientowego. Korzystamy z algorytmu L-BFGS. W każdej iteracji algorytmu zapisujemy aktualną wersję generowanego obrazu (pojedyncza iteracja jest reprezentacją 20 kroków algorytmu wzrostu gradientu).

Listing 8.22. Pętla transferu stylu

```
iterations <- 20

dms <- c(1, img_nrows, img_ncols, 3)

x <- preprocess_image(target_image_path)
x <- array_reshape(x, dim = length(x))
```

Stan początkowy: obraz docelowy.

Spłaszczamy obraz, ponieważ optymalizator może przetwarzać tylko płaskie wektory.

```
for (i in 1:iterations) {

opt <- optim(
    array_reshape(x, dim = length(x)),
    fn = evaluator$loss,
    gr = evaluator$grads,
    method = "L-BFGS-B",
    control = list(maxit = 15)
  )

  cat("Strata:", opt$value, "\n")

  image <- x <- opt$par
  image <- array_reshape(image, dms)

  im <- deprocess_image(image)
  plot(as.raster(im))
}
```

Optymalizacja L-BFGS przetwarza piksele wygenerowanego obrazu w celu zminimalizowania straty. Zwróć uwagę na konieczność przekazania funkcji obliczającej stratę i funkcji obliczającej gradienty w formie dwóch oddzielnych argumentów.

Na rysunku 8.8 pokazano efekt pracy algorytmu. Pamiętaj o tym, że technika ta przeprowadza transformację polegającą na zmianie tekstur lub ich przeniesieniu. Najlepiej sprawdza się z obrazami referencyjnymi wypełnionymi wyraźnymi teksturami i obrazami źródłowymi, które nie wymagają dużej ilości szczegółów do bycia rozpoznawalnymi. Zwykle nie da się w ten sposób wykonać operacji zmiany stylu portretu. Algorytm ten jest bliższy klasycznym technikom przetwarzania sygnału niż sztucznej inteligencji, a więc nie należy od niego oczekiwać cudów!

Działanie tego algorytmu jest dość powolne, ale transformacja ta jest na tyle prosta, że można jej dokonać również za pomocą małej szybkiej jednokierunkowej sieci konwolucyjnej (wymogiem jest dysponowanie odpowiednim zbiorem danych treningowych). Szybki transfer stylu może być osiągnięty poprzez wygenerowanie wejściowych i wyjściowych obrazów treningowych utrzymanych w jednym stylu za pomocą techniki zaprezentowanej w tym podrozdziale. Po wykonaniu tej operacji należy wytrenować prostą sieć konwolucyjną pod kątem wykonywania transformacji zgodnej z określonym stylem. Teraz zmiana stylu obrazu będzie przeprowadzana praktycznie natychmiastowo w wyniku jednej iteracji algorytmu małej konwolucyjnej sieci neuronowej.

8.3.4. Wnioski

- Transfer stylu polega na utworzeniu nowego obrazu, na którym zachowana zostanie treść przetwarzanego obrazu, ale zostanie ona przedstawiona w stylu obrazu referencyjnego.
- Treść może zostać rozpoznana przez aktywacje wysokopoziomowych warstw konwolucyjnej sieci neuronowej.
- Styl może zostać rozpoznany przez wewnętrzne korelacje aktywacji różnych warstw konwolucyjnej sieci neuronowej.
- Uczenie głębokie pozwala na utworzenie mechanizmu transferu stylu w formie procesu optymalizacji korzystającego z funkcji straty zdefiniowanej przy użyciu wytrenowanej wcześniej konwolucyjnej sieci neuronowej.
- Tę prostą ideę można rozbudowywać i modyfikować.

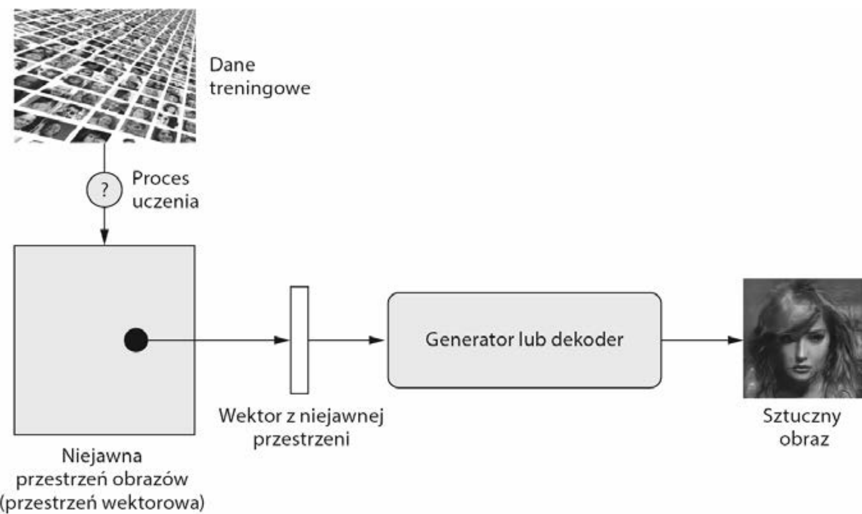
Rysunek 8.8. Przykładowe efekty pracy algorytmu

8.4. Generowanie obrazów przy użyciu wariacyjnych autoenkoderów

Próbkowanie z niejawnej przestrzeni obrazów w celu utworzenia zupełnie nowych obraz lub edycji istniejących obrazów jest obecnie najpopularniejszym kreatywnym zastosowaniem sztucznej inteligencji. W tym oraz w kolejnym podrozdziale tej książki opiszę wybrane wysokopoziomowe techniki generowania obrazów oraz przedstawię szczegóły implementacji dwóch technik tego typu: **wariacyjnych autoenkoderów** (ang. *variational autoencoder*, *VAE*) i **generatywnych sieci z przeciwnikiem** (ang. *generative adversarial network*, *GAN*). Zastosowania tych technik nie ograniczają się do generowania obrazów. Można ich używać również do generowania dźwięku, muzyki, a nawet tekstu, ale w praktyce najciekawsze rezultaty dają one podczas generowania obrazów, a więc skupimy się na tym zastosowaniu.

8.4.1. Próbkowanie z niejawnej przestrzeni obrazów

Głównym elementem procesu generowania obrazu jest tworzenie niskowymiarowej **niejawnej przestrzeni** reprezentacji (oczywiście jest to przestrzeń wektorowa), w której każdy punkt może zostać przypisany do realistycznie wyglądającego obrazu. Moduł zdolny do wykonania takiego mapowania, przyjmujący dowolny punkt w charakterze danych wejściowych i generujący na jego podstawie obraz (siatkę pikseli), określamy mianem **generatora** (w przypadku sieci GAN) lub **dekodera** (w przypadku koderów VAE). Po utworzeniu niejawnej przestrzeni można przeprowadzać operację próbkowania z niej punktów w sposób celowy lub losowy oraz mapowania ich w przestrzeni obrazu, co pozwala na generowanie nowych obrazów (patrz rysunki 8.9 i 8.10).



Rysunek 8.9. Trenowanie niejawnej wektorowej przestrzeni obrazów i używanie jej do próbkowania nowych obrazów

Rysunek 8.10. Ciągła przestrzeń twarzy wygenerowana przez Toma White'a przy użyciu koderów VAE

Sieci GAN i kodery VAE to dwie różne strategie trenowania niejawnych przestrzeni reprezentacji obrazów. Każda z nich ma swoje charakterystyki. Kodery VAE doskonale nadają się do trenowania niejawnych przestrzeni, które mają wyraźne struktury — przestrzeni, w których kierunki są osiami o określonym znaczeniu. Sieci GAN generują obrazy, które mogą wyglądać bardzo realistycznie, ale niejawna przestrzeń, z której są one próbkowane, może nie posiadać tak wyraźnej struktury i ciągłości.

8.4.2. Wektory koncepcyjne używane podczas edycji obrazu

O ideę **wektorów koncepcyjnych** otarliśmy się w rozdziale 6. przy okazji opisywania tworzenia osadzeń słów: w niejawnej przestrzeni reprezentacji lub przestrzeni osadzeń kierunki mogą służyć do kodowania osi zmiany oryginalnych danych. W niejawnej przestrzeni obrazów twarzy może istnieć np. **wektor uśmiechu** s — wektor taki, że jeżeli punkt z przestrzeni reprezentuje pewien obraz twarzy, to punkt $z + s$ reprezentuje tę samą twarz, która się uśmiecha. Po zidentyfikowaniu takiego wektora możemy edytować obrazy, podążając w określonym kierunku niejawnej przestrzeni. Wektory koncepcyjne mogą odwoływać się do dowolnego wymiaru wariacji przestrzeni obrazu — w przypadku obrazów twarzy można odkryć wektory dodające okulary przeciwsłoneczne, usuwające okulary z twarzy, zamieniające twarz mężczyzny w twarz kobiety itd. Na rysunku 8.11 przedstawiono przykład wektora uśmiechu opracowanego przy użyciu koderów VAE wytrenowanych na zbiorze danych zdjęć twarzy znanych osób (zbiorze CelebA) przez Toma White'a z nowozelandzkiej uczelni Victoria University School of Design.

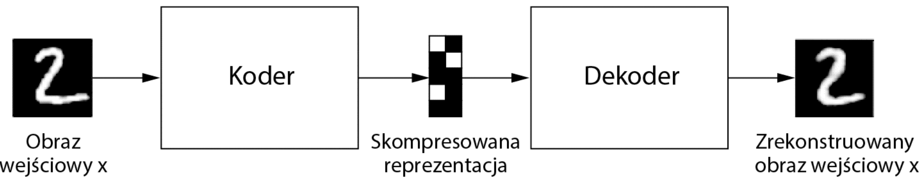
Kup ksi k Pole ksi k

Rysunek 8.11. Wektor uśmiechu

8.4.3. Wariacyjne autoenkodery

Wariacyjne autoenkodery zostały opracowane jednocześnie przez Kingmę i Wellinga w grudniu 2013 r.[6] i Rezende, Mohameda i Wierstrę w styczniu 2014 r.[7]. Są to generatywne modele sprawdzające się szczególnie w zadaniach edycji obrazu przy użyciu wektorów koncepcyjnych. Wariacyjne autoenkodery to nowoczesne podejście do autoenkoderów — sieci, która ma na celu zakodowanie danych wejściowych w postaci niskopoziomowej niejawnej przestrzeni, a następnie ich dekodowanie — jest to połączenie uczenia głębokiego z wnioskowaniem bayesowskim.

Klasyczny autoenkoder obrazu przetwarza obraz wejściowy, mapuje go na niejawną przestrzeń wektorową z zastosowaniem modułu kodera, a następnie dekoduje go do postaci wejściowej o takich samych wymiarach jak obraz oryginalny przy użyciu modułu dekodera (patrz rysunek 8.12). Autoenkoder jest trenowany tak, aby mógł dokonać rekonstrukcji obrazów wejściowych. Nakładając różne ograniczenia na wyjście kodera, można zmusić autoenkoder do uczenia się interesujących nas niejawnych reprezentacji danych. Zwykle kod ma być niskopoziomowy i rzadki (powinien zawierać głównie zera). W takim przypadku koder działa tak, jakby kompresował dane wejściowe do postaci formy składającej się z kilku bitów informacji.



Rysunek 8.12. Autoenkoder przypisujący obraz wejściowy x do skompresowanej reprezentacji, a następnie dekodujący go do zrekonstruowanej postaci obrazu x

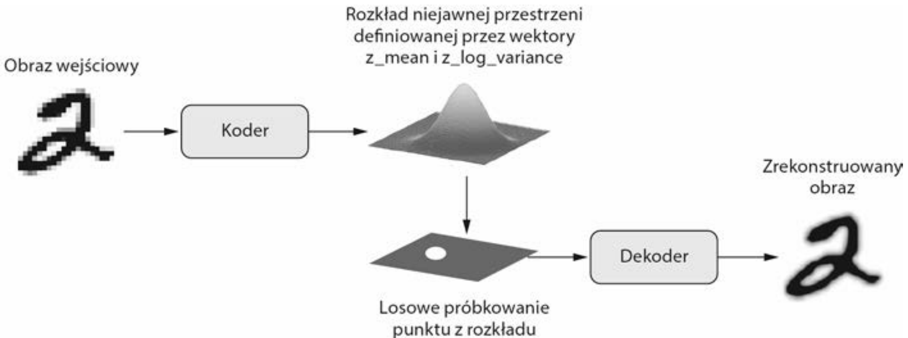
W praktyce takie klasyczne autoenkodery nie pozwalają na uzyskanie szczególnie przydatnej przestrzeni o ładnej strukturze. Nie są one również zbyt wydajne w roli mechanizmów kompresji danych. W związku z tym wyszły z użycia. Technika kodowania VAE

[6] Diederik P. Kingma i Max Welling, *Auto-Encoding Variational Bayes*, arXiv, 2013, *https://arxiv.org/abs/1312.6114*.

[7] Danilo Jimenez Rezende, Shakir Mohamed i Daan Wierstra, *Stochastic Backpropagation and Approximate Inference in Deep Generative Models*, arXiv, 2014, *https://arxiv.org/abs/1401.4082*.

usprawniła działanie autoenkoderów, dodając do nich nieco magii statystyki, dzięki której są one w stanie uczyć się ciągłych przestrzeni charakteryzujących się określoną strukturą. W ten sposób powstało solidne narzędzie służące do generowania obrazów.

Koder VAE zamiast kompresować obraz wejściowy do określonej formy kodu niejawnej przestrzeni, zamienia obraz w parametry rozkładu statystycznego: średnią i wariancję. Oznacza to założenie, że obraz wejściowy został wygenerowany przez proces statystyczny i podczas kodowania i dekodowania należy wziąć pod uwagę losowość tego procesu. Koder VAE korzysta następnie z parametrów średniej i wariancji w celu losowego próbkowania jednego elementu rozkładu i zdekodowania go z powrotem do oryginalnej postaci (patrz rysunek 8.13). Stochastyczność tego procesu poprawia jego siłę i zmusza niejawną przestrzeń do zapisywania wszędzie reprezentacji mających znaczenie: każdy punkt próbkowany w tej przestrzeni jest dekodowany do postaci poprawnego obiektu wejściowego.



Rysunek 8.13. Koder VAE mapuje obraz na dwa wektory: z_mean i z_log_variance, które definiują rozkład prawdopodobieństwa w niejawnej przestrzeni używanej do próbkowania dekodowanego punktu

Z technicznego punktu widzenia koder VAE działa w następujący sposób:

1. Moduł kodera zamienia próbki wejściowe `input_img` na dwa parametry niejawnej przestrzeni reprezentacji: `z_mean` i `z_log_variance`.
2. Punkt `z` jest losowo próbkowany z niejawnego rozkładu normalnego, co ma doprowadzić do wygenerowania obrazu wejściowego za pomocą działania `z = z_mean + exp(z_log_variance) * epsilon`, gdzie `epsilon` jest losowym tensorem o małych wartościach.
3. Moduł dekodera mapuje ten punkt w niejawnej przestrzeni z powrotem na formę oryginalnego obrazu.

Współczynnik `epsilon` przyjmuje wartość losową, a więc proces zapewnia to, że każdy punkt zbliżony w niejawnej przestrzeni do punktu, w którym zakodowano obraz `input_img(z-mean)`, po rozkodowaniu będzie wyglądał podobnie do obrazu `input_img` (niejawna przestrzeń będzie miała charakter sensowny i ciągły). Po zdekodowaniu dowolnych dwóch zbliżonych do siebie punktów niejawnej przestrzeni uzyskamy bardzo podobne do siebie obrazy. Ciągłość w połączeniu z małą liczbą wymiarów niejawnej

przestrzeni sprawia, że każdy kierunek tej przestrzeni tworzy oś zmienności danych o konkretnym znaczeniu — niejawna przestrzeń ma charakter wysoce ustrukturyzowany i łatwo poruszać się w niej przy użyciu wektorów koncepcji.

Parametry kodera VAE są trenowane przy użyciu dwóch funkcji straty: **straty rekonstrukcji**, która wymusza dopasowanie dekodowanych próbek do obrazów wejściowych, i **straty regularyzacji**, która wspiera tworzenie niejawnych przestrzeni o poprawnej formie, a także umożliwia zmniejszenie nadmiernego dopasowania do treningowego zbioru danych.

Przeanalizujmy szybko implementację kodera VAE w pakiecie Keras. Schematycznie wygląda ona tak:

```
c(z_mean, z_log_variance) %<% encoder(input_img)   # Kodowanie obrazu wejściowego za pomocą parametrów średniej i wariancji.

z <- z_mean + exp(z_log_variance) * epsilon   # Wyciąganie punktu z przestrzeni przy użyciu małej losowej wartości epsilon.

reconstructed_img <- decoder(z)   # Dekodowanie obrazu.

model <- keras_model(input_img, reconstructed_img)   # Tworzenie instancji modelu autoenkodera mapującego obraz wejściowy na jego rekonstrukcję.
```

Model jest następnie trenowany z wykorzystaniem dwóch funkcji straty: straty rekonstrukcji i straty regularyzacji.

Poniższy kod pokazuje sieć kodera mapującą obrazy na parametry rozkładu prawdopodobieństwa umieszczone w przestrzeni o niejawnym charakterze. W praktyce jest to prosta sieć konwolucyjna przypisująca obraz wejściowy `x` do dwóch wektorów: `z_mean` i `z_log_var`.

Listing 8.23. Sieć kodera VAE

```
library(keras)

img_shape <- c(28, 28, 1)
batch_size <- 16
latent_dim <- 2L   # Liczba wymiarów niejawnej przestrzeni: pracujemy z przestrzenią dwuwymiarową.

input_img <- layer_input(shape = img_shape)

x <- input_img %>%
  layer_conv_2d(filters = 32, kernel_size = 3, padding = "same",
                activation = "relu") %>%
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu", strides = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu") %>%
  layer_conv_2d(filters = 64, kernel_size = 3, padding = "same",
                activation = "relu")

shape_before_flattening <- k_int_shape(x)

x <- x %>%
  layer_flatten() %>%
```

Kup książkę Poleć książkę

```
  layer_dense(units = 32, activation = "relu")

z_mean <- x %>%
  layer_dense(units = latent_dim)
z_log_var <- x %>%
  layer_dense(units = latent_dim)
```

Obraz wejściowy jest końcowo zapisywany w formie tych dwóch parametrów.

Oto kod pozwalający na użycie parametrów rozkładu statystycznego `z_mean` i `z_log_var`, które z założenia mają pozwalać na utworzenie obrazu `input_img` w celu wygenerowania punktu `z` niejawnej przestrzeni. Część kodu R korzystającego z zaplecza pakietu Keras obudowujemy warstwą `layer_lambda`. W Keras wszystko musi być warstwą, a więc kod, który nie należy do wbudowanej warstwy, powinien mieć formę warstwy `layer_lambda` lub innej samodzielnie zdefiniowanej warstwy.

Listing 8.24. Funkcja próbkowania niejawnej przestrzeni

```
sampling <- function(args) {
  c(z_mean, z_log_var) %<-% args
  epsilon <- k_random_normal(shape = list(k_shape(z_mean)[1], latent_dim),
                             mean = 0, stddev = 1)
  z_mean + k_exp(z_log_var) * epsilon
}

z <- list(z_mean, z_log_var) %>%
  layer_lambda(sampling)
```

Poniższy fragment kodu przedstawia implementację dekodera. Wektor `z` jest modyfikowany tak, aby uzyskał wymiary obrazu, a następnie używanych jest kilka warstw konwolucyjnych w celu wygenerowania ostatecznej postaci obrazu wyjściowego mającego takie same wymiary jak oryginalny obraz `input_img`.

Listing 8.25. Sieć dekodera VAE mapująca punkty niejawnej przestrzeni na obrazy

```
decoder_input <- layer_input(k_int_shape(z)[-1])

x <- decoder_input %>%
  layer_dense(units = prod(as.integer(shape_before_flattening[-1])),
              activation = "relu") %>%
  layer_reshape(target_shape = shape_before_flattening[-1]) %>%
  layer_conv_2d_transpose(filters = 32, kernel_size = 3, padding = "same",
                          activation = "relu", strides = c(2, 2)) %>%
  layer_conv_2d(filters = 1, kernel_size = 3, padding = "same",
                activation = "sigmoid")

decoder <- keras_model(decoder_input, x)

z_decoded <- decoder(z)
```

Wejście wektora z.

Zwiększanie rozdzielczości obiektu wejściowego.

Zmiana kształtu wektora w celu uzyskania map cech o takim samym kształcie jak kształt mapy cech przed ostatnią warstwą flatten modułu kodującego.

Uzyskanie mapy cech o takim samym rozmiarze jak nieprzetworzone dane wejściowe

Warstwy używane w celu odkodowania wektora z do formy mapy cech o takim samym rozmiarze jak oryginalny obraz wejściowy.

Tworzenie instancji modelu dekodera zamieniającego obiekt decoder_input na zdekodowany obraz.

Przyjmuje wektor z i zwraca jego zdekodowaną postać.

Dualizm funkcji straty kodera VAE nie wpasowuje się w tradycyjne ramy funkcji próbującej w formie `loss(input, target)`. W związku z tym zdefiniujemy dodatkową warstwę, która będzie wewnętrznie korzystać z metody `add_loss` w celu wygenerowania wartości straty.

Listing 8.26. Definicja warstwy używanej do obliczania wartości straty kodera VAE

```
library(R6)

CustomVariationalLayer <- R6Class("CustomVariationalLayer",
  inherit = KerasLayer,

  public = list(

    vae_loss = function(x, z_decoded) {
      x <- k_flatten(x)
      z_decoded <- k_flatten(z_decoded)
      xent_loss <- metric_binary_crossentropy(x, z_decoded)
      kl_loss <- -5e-4 * k_mean(
        1 + z_log_var - k_square(z_mean) - k_exp(z_log_var),
        axis = -1L
      )
      k_mean(xent_loss + kl_loss)
    },

    call = function(inputs, mask = NULL) {    <- Własne warstwy implementuje się, pisząc metodę wywołującą.
      x <- inputs[[1]]
      z_decoded <- inputs[[2]]
      loss <- self$vae_loss(x, z_decoded)
      self$add_loss(loss, inputs = inputs)
      x    <- Nie korzystamy z tych zwracanych danych, ale warstwa musi coś zwracać.
    }
  )
)

layer_variational <- function(object) {
  create_layer(CustomVariationalLayer, object, list())
}
# Obudowuje klasę R6, nadając jej postać standardowej funkcji warstwy Keras.

y <- list(input_img, z_decoded) %>%
  layer_variational()
# Wywołujemy własną warstwę na obiekcie wejściowym i odkodowanym obiekcie wyjściowym w celu wygenerowania ostatecznego obiektu generowanego przez model.
```

Teraz możemy utworzyć instancję modelu i ją wytrenować. Wartością straty zajęliśmy się w utworzonej ręcznie warstwie, a więc nie musimy określać zewnętrznej funkcji straty w czasie kompilacji (`loss=NULL`), co z kolei oznacza, że nie będziemy przekazywać docelowych danych w czasie trenowania (do funkcji `fit()` modelu przekazujemy tylko argument `x_train`).

Listing 8.27. Trenowanie kodera VAE

```
vae <- keras_model(input_img, y)

vae %>% compile(
  optimizer = "rmsprop",
```

```
  loss = NULL
)
mnist <- dataset_mnist()
c(c(x_train, y_train), c(x_test, y_test)) %<-% mnist

x_train <- x_train / 255
x_train <- array_reshape(x_train, dim =c(dim(x_train), 1))

x_test <- x_test / 255
x_test <- array_reshape(x_test, dim =c(dim(x_test), 1))

vae %>% fit(
  x = x_train, y = NULL,
  epochs = 10,
  batch_size = batch_size,
  validation_data = list(x_test, NULL)
)
```

Po wytrenowaniu modelu na zbiorze MNIST możemy użyć sieci `decoder` w celu wygenerowania obrazów na podstawie dowolnej niejawnej przestrzeni wektorów.

Listing 8.28. Próbkowanie siatki punktów z niejawnej dwuwymiarowej przestrzeni i dekodowanie ich w celu wygenerowania obrazów

```
n <- 15
digit_size <- 28

grid_x <- qnorm(seq(0.05, 0.95, length.out = n))
grid_y <- qnorm(seq(0.05, 0.95, length.out = n))


op <- par(mfrow = c(n, n), mar = c(0,0,0,0), bg = "black")
for (i in 1:length(grid_x)) {
  yi <- grid_x[[i]]
  for (j in 1:length(grid_y)) {
    xi <- grid_y[[j]]
    z_sample <- matrix(c(xi, yi), nrow = 1, ncol = 2)
    z_sample <- t(replicate(batch_size, z_sample, simplify = "matrix"))
    x_decoded <- decoder %>% predict(z_sample, batch_size = batch_size)
    digit <- array_reshape(x_decoded[1,,,], dim = c(digit_size, digit_size))
    plot(as.raster(digit))
  }
}
par(op)
```

Wyświetlamy siatkę 15×15 cyfr (łącznie 255 cyfr).

Transformacja liniowych współrzędnych przy użyciu funkcji qnorm w celu wygenerowania wartości niejawnej zmiennej z (pracujemy z przestrzenią Gaussa).

Tworzenie obrazów na podstawie wsadu.

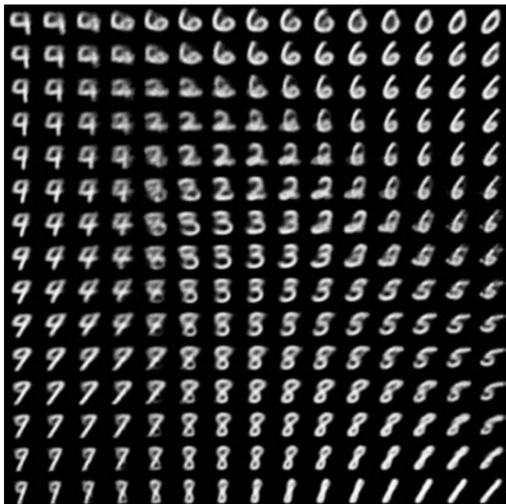
Zwielokrotnianie obiektu z w celu uzyskania pełnego wsadu.

Zmienia kształt pierwszej cyfry wsadu z 28×28×1 na 28×28.

Wygenerowana siatka cyfr (patrz rysunek 8.14) pokazuje całkowitą ciągłość rozkładu różnych klas cyfr — na kolejnych obrazach widać, że cyfry stopniowo przechodzą w siebie. Kierunki w tej przestrzeni mają znaczenie: podążając w jednym z wybranych kierunków, otrzymujemy obraz przypominający bardziej cyfrę 4, a podrażając w innym otrzymujemy obraz przypominający bardziej cyfrę 1 itd.

W kolejnym podrozdziale opiszę następne narzędzie służące do generowania sztucznych obrazów: generatywne sieci z przeciwnikiem (sieci GAN).

Kup ksi k Pole ksi k



Rysunek 8.14. Siatka cyfr rozkodowana na podstawie niejawnej przestrzeni

8.4.4. Wnioski

- Generowanie obrazów za pomocą technik uczenia głębokiego przebiega poprzez uczenie modeli niejawnych przestrzeni odzwierciedlających statystyczne informacje o zbiorze danych zawierającym obrazy. Próbkując i rozkodowując punkty z tej przestrzeni, możemy wygenerować nowe obrazy. Można to robić za pomocą dwóch głównych technik: koderów VAE i sieci GAN.
- Kodery VAE tworzą wysoce ustrukturyzowaną, ciągłą przestrzeń reprezentacji. W związku z tym doskonale nadają się do edycji obrazów: zamieniania twarzy, dodawania lub usuwania uśmiechu itd. Ponadto sprawdzają się również podczas generowania animacji opartych na niejawnej przestrzeni — animacji pokazujących stopniowe przeobrażanie się jednego obrazu w inny.
- Sieci GAN umożliwiają generowanie realistycznie wyglądających obrazów, ale nie korzystają z ciągłej przestrzeni o solidnej strukturze.

Większość mechanizmów generowania obrazów, które widziałem, są oparte na koderach VAE, ale sieci GAN są dość popularne wśród naukowców, a przynajmniej było tak w latach 2016 – 2017. W kolejnym podrozdziale opiszę ich działanie i implementację.

WSKAZÓWKA Jeżeli chcesz pobawić się jeszcze mechanizmem generowania obrazów, to polecam Ci korzystanie ze zbioru danych Largescale Celeb Faces Attributes (CelebA). Jest on dostępny za darmo i zawiera ponad 200 000 portretowych zdjęć gwiazd, które doskonale nadają się do eksperymentowania z wektorami koncepcyjnymi. Pracuje się z nim o wiele lepiej niż ze zbiorem MNIST.

Kup ksi k Pole ksi k

8.5. Wprowadzenie do generatywnych sieci z przeciwnikiem

Generatywne sieci z przeciwnikiem (sieci GAN) zostały opracowane w 2014 r. przez zespół kierowany przez Iana Goodfellowa[8] jako alternatywa dla koderów VAE. Sieci te pozwalają na generowanie dość realistycznie wyglądających obrazów poprzez wymuszenie tego, żeby pod względem parametrów statystycznych sztucznie wygenerowane obrazy były praktycznie nie do odróżnienia od prawdziwych zdjęć.

Intuicyjnie działanie sieci GAN można porównać do czynności podejmowanych przez fałszerza próbującego podrobić obraz Picassa. Na początku fałszerz nie posiada najlepszych umiejętności. Miesza niektóre ze swoich podróbek z oryginalnymi obrazami Picassa i pokazuje je wszystkie osobie zajmującej się sprzedażą dzieł sztuki, która ocenia autentyczność każdego z obrazów i udziela fałszerzowi informacji zwrotnej na temat tego, co sprawia, że dany obraz wygląda tak, jakby namalował go Picasso. Fałszerz wraca do swojej pracowni i przygotowuje kolejne falsyfikaty. Wraz z upływem czasu będzie on coraz lepiej imitował styl Picassa, a sprzedawca będzie coraz lepiej dostrzegał podróbki. Na koniec obie osoby będą dysponowały doskonałymi podróbkami obrazów Picassa.

Tak właśnie działają sieci GAN: składają się z sieci fałszerza i sieci eksperta. Każda z nich jest trenowana tak, aby działała lepiej od drugiej. Sieć GAN składa się z dwóch części:

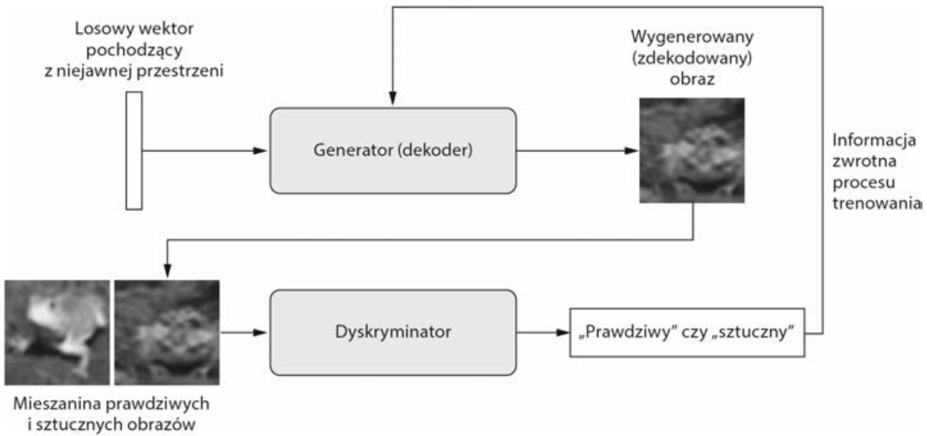
- **sieci generatora** — sieci przyjmującej na wejściu losowy wektor (losowy punkt niejawnej przestrzeni) i dekodującej go w celu wygenerowania syntetycznego obrazu;
- **sieci dyskryminatora** (**sieci przeciwnika**) — sieci przyjmującej na wejściu obraz (prawdziwy lub sztucznie wygenerowany) i określającej to, czy obraz pochodzi z treningowego zbioru danych, czy został wygenerowany przez sieć generatora.

Sieć generatora jest trenowana tak, aby mogła oszukać sieć dyskryminatora, a więc wraz z postępem procesu trenowania będzie generować coraz bardziej realistyczne obrazy — obrazy te będą praktycznie nie do odróżnienia od obrazów rzeczywistych z punktu widzenia dyskryminatora (patrz rysunek 8.15), a sieć dyskryminatora będzie w tym czasie stale dostosowywać się do coraz lepszych efektów pracy generatora, zwiększając poziom realizmu generowanych obrazów. Po zakończeniu trenowania generator jest w stanie zamienić dowolny punkt swojej przestrzeni wejściowej w realistyczny obraz. W przeciwieństwie do koderów VAE ta niejawna przestrzeń nie jest ciągła i nie musi być tak łatwa w interpretacji.

Warto zwrócić uwagę na to, że sieć GAN jest pierwszym rozwiązaniem przedstawionym w tej książce, w którym minimum optymalizacji nie jest czymś ustalonym. W standardowej sytuacji algorytm spadku gradientowego śledzi funkcję straty, której kolejne wartości nie wpływają w dużym stopniu na otrzymywane wyniki, a w przypadku sieci GAN każda zmiana wartości straty powoduje niewielką modyfikację wszystkiego. To dynamiczny system, w którym proces optymalizacji nie szuka minimum, ale równowagi

[8] Ian J. Goodfellow i inni, *Generative Adversarial Networks*, arXiv, 2014, *https://arxiv.org/abs/1406.2661*.

Kup ksi k Pole ksi k



Rysunek 8.15. Generator zamienia losowe wektory niejawnej przestrzeni w obrazy, a dyskryminator stara się odróżnić prawdziwe obrazy od tych, które zostały wygenerowane; generator jest trenowany w celu oszukania dyskryminatora

między dwoma siłami. W związku z tym sieci GAN są bardzo trudne do trenowania. Uzyskanie działającej sieci GAN wymaga dokładnego dostrajania architektury modelu i parametrów jego trenowania. Na rysunku 8.16 przedstawiono efekty działania sieci GAN.

Rysunek 8.16. Obrazy wyciągnięte z niejawnej przestrzeni przez sieć GAN; zostały wygenerowane przez Mike'a Tykę przy użyciu wieloetapowej sieci GAN trenowanej na zbiorze danych ze zdjęciami twarzy (*http://www.miketyka.com/*)

8.5.1. Schematyczna implementacja sieci GAN

W tej sekcji wyjaśnię implementację najprostszej formy sieci GAN w pakiecie Keras. Sieci GAN są zaawansowane, a więc zagłębianie się w techniczne szczegóły wykraczałoby poza zakres tematyczny tej książki. Zaprezentuję implementację **głębokiej konwolucyjnej sieci GAN** (DCGAN) — sieci GAN, w której generator i dyskryminator są głębokimi sieciami konwolucyjnymi. W praktyce zastosuję warstwę `layer_conv_2d_transpose` w celu zwiększenia próbkowania w generatorze.

Sieć GAN będzie trenowana na zbiorze CIFAR10 składającym się z 50 000 kolorowych obrazów o rozdzielczości 32×32, podzielonych na 10 równych klas (w każdej z klas znajduje się 5000 obrazów). Dla ułatwienia będziemy korzystać tylko z obrazów należących do klasy „żaba".

Kup książkę Poleć książkę

Schematycznie działanie tej sieci GAN można przedstawić w następujący sposób:

1. Sieć generatora (`generator`) mapuje wektory o kształcie `(latent_dim,)` na obrazy o kształcie `(32, 32, 3)`.
2. Sieć dyskryminatora (`discriminator`) mapuje obrazy o kształcie `(32, 32, 3)` na binarną wartość określającą prawdopodobieństwo tego, że obraz jest prawdziwy.
3. Sieć `gan` tworzy łańcuch składający się z generatora i dyskryminatora: `gan(x) <- discriminator(generator(x))`. Sieć `gan` mapuje wektory niejawnej przestrzeni na oceny realizmu wystawiane przez dyskryminator.
4. Trenujemy dyskryminator przy użyciu przykładów prawdziwych i sztucznych obrazów oznaczonych etykietami, tak jakbyśmy trenowali zwykły model klasyfikacji obrazów.
5. W celu wytrenowania generatora korzystamy z gradientów wag generatora w odniesieniu do straty modelu `gan`. W związku z tym każdy krok trenowania ma modyfikować wagi generatora tak, aby zwiększyć prawdopodobieństwo zaklasyfikowania wygenerowanych obrazów jako prawdziwych. Innymi słowy, trenujemy generator tak, aby był w stanie oszukać dyskryminator.

8.5.2. Zbiór przydatnych rozwiązań

Proces trenowania sieci GAN i dostrajania ich implementacji jest bardzo trudny. W związku z tym warto poznać rozwiązania ułatwiające pracę z tymi sieciami. Rozwiązania te mają naturę heurystyczną i nie są teoretycznymi wskazówkami — podobnie jak z większością zagadnień uczenia głębokiego, mamy do czynienia bardziej z alchemią niż fizyką. Mają one pomóc w zrozumieniu bieżącego problemu. Są sprawdzone w działaniu, ale nie nadają się do każdego kontekstu.

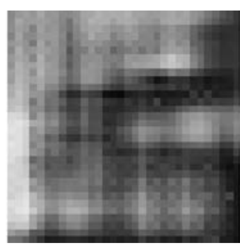
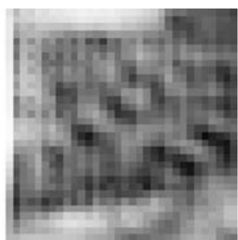
Oto kilka rozwiązań zastosowanych w zaprezentowanej w tym podrozdziale implementacji generatora i dyskryminatora GAN. Nie jest to lista wszystkich możliwych rozwiązań pomocniczych. Jeżeli zainteresował Cię ten temat, to zajrzyj do książek poświęconych sieciom GAN.

- Ostatnią warstwą aktywacji modelu jest `tanh`, a nie warstwa `sigmoid` spotykana w większości innych modeli.
- Próbkowanie punktów z niejawnej przestrzeni dokonujemy przy użyciu **rozkładu normalnego** (rozkładu Gaussa), a nie rozkładu jednorodnego.
- Stochastyczność przyczynia się do uzyskania bardziej solidnego modelu. Trenowanie sieci GAN przyczynia się do powstania dynamicznej równowagi, a więc proces ten może utknąć w wielu punktach. Wprowadzanie losowości do procesu trenowania pomaga temu zapobiec. Losowość wprowadzamy na dwa sposoby: korzystając z mechanizmu odrzucania zaimplementowanego w dyskryminatorze i poprzez dodanie losowego szumu do etykiet przetwarzanych przez dyskryminator.
- Rzadkie gradienty mogą przeszkodzić w trenowaniu sieci GAN. W uczeniu głębokim rzadki charakter danych jest często czymś wręcz pożądanym, ale nie dotyczy to sieci GAN. Do powstawania rzadkich gradientów mogą przyczynić się dwie rzeczy: operacje `maxpooling` i aktywacje `ReLU`. Zamiast operacji `maxpooling` polecam

Kup ksi k Pole ksi k

stosowanie krokowych konwolucji w celu zmniejszenia objętości próbek, a zamiast aktywacji `ReLU` polecam stosowanie warstwy `layer_activation_leaky_relu`. Działa ona podobnie do warstwy `ReLU`, ale nie posiada tak dużych ograniczeń — pozwala na pojawianie się niewielkich ujemnych wartości aktywacji.

- W wygenerowanych obrazach często pojawiają się artefakty wyglądające jak szachownica (patrz rysunek 8.17). Powstają one w wyniku nierównego pokrycia przestrzeni pikseli w generatorze. W celu rozwiązania tego problemu, za każdym razem, gdy będziemy korzystać z kroku `layer_conv_2d_transpose` lub `layer_conv_2d`, zastosujemy rozmiar jądra podzielny przez rozmiar kroku (dotyczy to generatora i dyskryminatora).

Rysunek 8.17. Artefakty przypominające swym wyglądem szachownice; są wywołane przez brak korelacji między rozmiarem kroku a rozmiarem jądra w pokryciu przestrzeni pikseli; jest to jeden z wielu problemów spotykanych podczas pracy z sieciami GAN

8.5.3. Generator

Zacznijmy od opracowania modelu `generator` zamieniającego wektor pochodzący z niejawnej przestrzeni (podczas trenowania będzie on próbkowany losowo) w obraz. Jednym z typowych problemów spotykanych podczas pracy z sieciami GAN jest stałe generowanie obrazów wyglądających jak szum. Można to rozwiązać, stosując technikę odrzucania w implementacjach dyskryminatora i generatora.

Listing 8.29. Generator sieci GAN

```
library(keras)

latent_dim <- 32
height <- 32
width <- 32
channels <- 3

generator_input <- layer_input(shape = c(latent_dim))


generator_output <- generator_input %>%

  layer_dense(units = 128 * 16 * 16) %>%
  layer_activation_leaky_relu() %>%
  layer_reshape(target_shape = c(16, 16, 128)) %>%

  layer_conv_2d(filters = 256, kernel_size = 5,
```

Zamiana obiektu wejściowego w 128-kanałową mapę cech o wymiarach 16×16.

Kup książkę Poleć książkę

```
                 padding = "same") %>%
  layer_activation_leaky_relu() %>%

  layer_conv_2d_transpose(filters = 256, kernel_size = 4,
                          strides = 2, padding = "same") %>%
  layer_activation_leaky_relu() %>%
```

Zwiększenie rozmiaru do 32×32.

```
  layer_conv_2d(filters = 256, kernel_size = 5,
                padding = "same") %>%
  layer_activation_leaky_relu() %>%
  layer_conv_2d(filters = 256, kernel_size = 5,
                padding = "same") %>%
  layer_activation_leaky_relu() %>%

  layer_conv_2d(filters = channels, kernel_size = 7,
                activation = "tanh", padding = "same")

generator <- keras_model(generator_input, generator_output)
```

Generuje jednokanałową mapę cech o rozmiarze 32×32 (rozmiar ten jest taki sam jak rozmiar obrazów wchodzących w skład zbioru CIFAR10).

Tworzy instancję generatora, która mapuje obiekt wejściowy o kształcie (latent_dim) na obraz o kształcie (32, 32, 3).

8.5.4. Dyskryminator

Teraz możemy przystąpić do pracy nad modelem `discriminator`, który przyjmuje na swoim wejściu obraz (prawdziwy lub sztuczny) i klasyfikuje go do jednej z dwóch klas: „obraz wygenerowany" lub „obraz pochodzący z treningowego zbioru danych".

Listing 8.30. Dyskryminator sieci GAN

```
discriminator_input <- layer_input(shape = c(height, width, channels))

discriminator_output <- discriminator_input %>%
  layer_conv_2d(filters = 128, kernel_size = 3) %>%
  layer_activation_leaky_relu() %>%
  layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
  layer_activation_leaky_relu() %>%
  layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
  layer_activation_leaky_relu() %>%
  layer_conv_2d(filters = 128, kernel_size = 4, strides = 2) %>%
  layer_activation_leaky_relu() %>%
  layer_flatten() %>%

  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 1, activation = "sigmoid")

discriminator <- keras_model(discriminator_input, discriminator_output)

discriminator_optimizer <- optimizer_rmsprop(
    lr = 0.0008,
    clipvalue = 1.0,
```

Warstwa odrzucania. To bardzo ważne rozwiązanie.

Warstwa klasyfikacji.

Tworzenie instancji modelu dyskryminatora zamieniającego obiekt wejściowy mający kształt (32, 32, 3) na wynik klasyfikacji binarnej określającej prawdziwość obrazu.

Optymalizator korzysta z mechanizmu ucinania wartości gradientu.

Kup ksi k   Pole ksi k

```
    decay = 1e-8)
)

discriminator %>% compile(
  optimizer = discriminator_optimizer,
  loss = "binary_crossentropy"
)
```

W celu uzyskania stabilnego przebiegu procesu trenowania korzystamy z parametru rozkładu współczynnika uczenia.

8.5.5. Sieć z przeciwnikiem

Teraz czas skonfigurować sieć GAN, która łączy generator z dyskryminatorem. Po wytrenowaniu model ten pchnie generator w kierunku usprawniającym oszukiwanie dyskryminatora. Model ten zamienia punkty niejawnej przestrzeni w etykiety klasyfikacji: „prawdziwy" lub „sztuczny" i ma być trenowany na etykietach zawsze wskazujących prawdziwość obrazu. W związku z tym trenowanie modelu `gan` doprowadzi do modyfikacji wartości wag generatora tak, aby zwiększyć prawdopodobieństwo orzeczenia przez dyskryminator analizujący sztuczne obrazy tego, że są one prawdziwe. Podczas trenowania dyskryminator powinien być zamrożony (nie należy go trenować) — w czasie trenowania modelu `gam` wagi dyskryminatora nie będą modyfikowane. Gdyby wagi dyskryminatora były modyfikowane podczas tego procesu, to trenowalibyśmy dyskryminator tak, aby zawsze przewidywał prawdziwość obrazu, a przecież nie tego chcemy!

Listing 8.31. Sieć przeciwników

```
freeze_weights(discriminator)

gan_input <- layer_input(shape = c(latent_dim))
gan_output <- discriminator(generator(gan_input))
gan <- keras_model(gan_input, gan_output)

gan_optimizer <- optimizer_rmsprop(
  lr = 0.0004,
  clipvalue = 1.0,
  decay = 1e-8
)

gan %>% compile(
  optimizer = gan_optimizer,
  loss = "binary_crossentropy"
)
```

Uniemożliwia trenowanie wag dyskryminatora (tylko w modelu gan).

8.5.6. Trenowanie sieci DCGAN

Teraz możemy rozpocząć proces trenowania. Oto lista czynności wykonywanych podczas każdej epoki trenowania (tak właśnie powinna działać pętla trenująca model):

1. Wybierz losowe punkty z niejawnej przestrzeni (losowy szum).
2. Użyj generatora w celu wygenerowania obrazów zawierających losowy szum.
3. Połącz wygenerowane obrazy z prawdziwymi.

Kup ksi k Pole ksi k

4. Wytrenuj dyskryminator przy użyciu wylosowanych obrazów z etykietami określającymi prawdziwość obrazów.
5. Wybierz kolejne losowe punkty z niejawnej przestrzeni.
6. Trenuj model `gan` w tym celu, aby wszystkie obrazy były uznawane przez dyskryminator za prawdziwe. W tym procesie zmodyfikowane zostaną wagi generatora (podczas trenowania modelu `gan` wagi dyskryminatora są zamrożone), tak aby zwiększyć prawdopodobieństwo tego, że wygenerowane obrazy zostaną uznane przez dyskryminator za prawdziwe — generator jest trenowany tak, żeby był w stanie oszukać dyskryminator.

Czas zaimplementować ten mechanizm.

Listing 8.32. Implementacja trenowania modelu GAN

```
cifar10 <- dataset_cifar10()                           ← Ładowanie zbioru danych CIFAR10.
c(c(x_train, y_train), c(x_test, y_test)) %<-% cifar10

 x_train <- x_train[as.integer(y_train) == 6,,,]       ← Wybór obrazów żab (klasa numer 6).
x_train <- x_train / 255                               ← Normalizacja danych.


iterations <- 10000
batch_size <- 20
save_dir <- 'twój_folder'    ← Miejsce zapisu wygenerowanych obrazów.

start <- 1
for (step in 1:iterations) {
  random_latent_vectors <- matrix(rnorm(batch_size * latent_dim),
                                  nrow = batch_size, ncol = latent_dim)
      Dekodowanie punktów w celu wygenerowania sztucznych obrazów.

  generated_images <- generator %>% predict(random_latent_vectors)
      Próbkowanie losowych punktów z niejawnej przestrzeni.


  stop <- start + batch_size - 1
  real_images <- x_train[start:stop,,,]
  rows <- nrow(real_images)
  combined_images <- array(0, dim = c(rows * 2, dim(real_images)[-1]))
  combined_images[1:rows,,,] <- generated_images
  combined_images[(rows+1):(rows*2),,,] <- real_images
      Łączenie obrazów sztucznych z prawdziwymi.

  labels <- rbind(matrix(1, nrow = batch_size, ncol = 1),
                  matrix(0, nrow = batch_size, ncol = 1))
      Tworzenie etykiet umożliwiających odróżnienie obrazów prawdziwych od sztucznych.

  labels <- labels + (0.5 * array(runif(prod(dim(labels))),
                                  dim = dim(labels)))
      Ważny zabieg: wprowadzanie losowego szumu do etykiet.

  d_loss <- discriminator %>% train_on_batch(combined_images, labels)
      Trenowanie dyskryminatora.

  random_latent_vectors <- matrix(rnorm(batch_size * latent_dim),
                                  nrow = batch_size, ncol = latent_dim)
      Losowe próbkowanie punktów w niejawnej przestrzeni.
```

Kup ksi k Pole ksi k

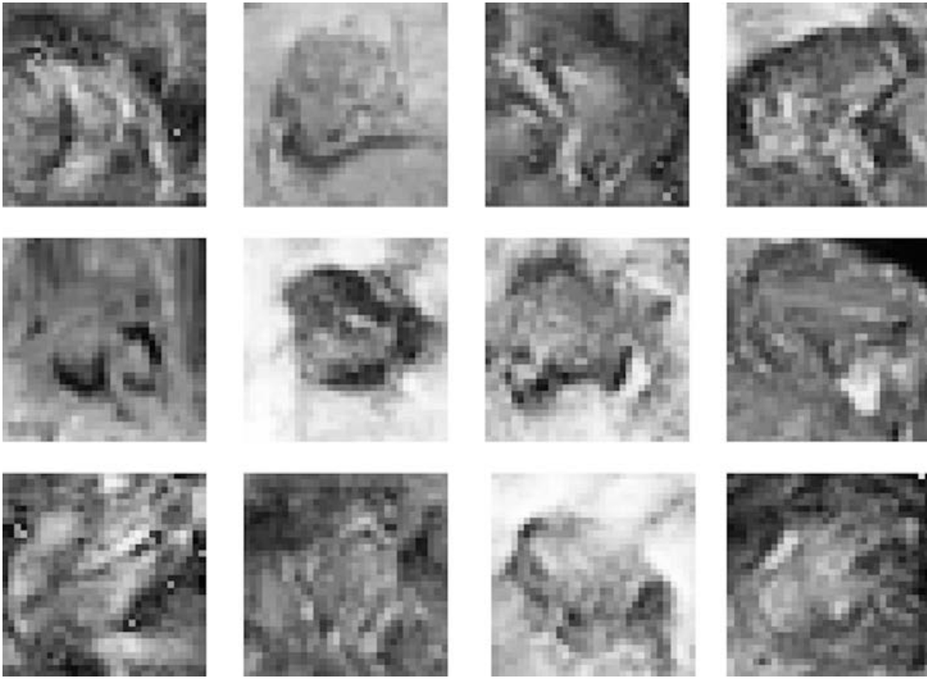
```
    misleading_targets <- array(0, dim = c(batch_size, 1))   # Tworzenie fałszywych etykiet stwierdzających oryginalność wszystkich obrazów.

    a_loss <- gan %>% train_on_batch(      # Trenowanie generatora przy użyciu modelu gan
      random_latent_vectors,               # i zamrożeniu wag dyskryminatora.
      misleading_targets
    )
    start <- start + batch_size
    if (start > (nrow(x_train) - batch_size))
      start <- 1

    if (step %% 100 == 0) {   # Okazjonalny zapis obrazów.

      save_model_weights_hdf5(gan, "gan.h5")   # Zapis wag modelu.

      cat("strata dyskryminatora w kroku :", d_loss, "\n")   # Wyświetlanie metryk.
      cat("strata przeciwna:", a_loss, "\n")

      image_array_save(                     # Zapis jednego wygenerowanego obrazu.
        generated_images[1,,,] * 255,
        path = file.path(save_dir, paste0("generated_frog", step, ".png"))
      )

      image_array_save(                     # Zapis jednego prawdziwego obrazu w celach porównawczych.
        real_images[1,,,] * 255,
        path = file.path(save_dir, paste0("real_frog", step, ".png"))
      )
    }
  }
```

Podczas trenowania można zaobserwować duży wzrost przeciwnej straty przy stracie dyskryminacyjnej zbliżonej do zera — generator może zostać zdominowany przez dyskryminator. W takim przypadku należy zmniejszyć wartość współczynnika uczenia dyskryminatora i zwiększyć wartość współczynnika mechanizmu odrzucania zaimplementowanego w dyskryminatorze. Przyjrzyj się rysunkowi 8.18, aby zrozumieć problemy, jakim sprostać musi dyskryminator.

8.5.7. Wnioski

- Sieć GAN składa się z sieci generatora sprzężonej z siecią dyskryminatora. Dyskryminator jest trenowany do odróżniania obrazów wygenerowanych przez generator od prawdziwych obrazów pochodzących z zewnętrznego zbioru. Generator jest natomiast trenowany w celu coraz lepszego oszukiwania dyskryminatora. Generator nigdy nie widzi bezpośrednio obrazów wchodzących w skład treningowego zbioru danych — dysponuje tylko informacjami przekazywanymi przez dyskryminator.
- Trenowanie sieci GAN jest trudne, ponieważ jest to proces dynamiczny i nie stosuje się w nim prostego algorytmu spadku gradientowego. Poprawne zaimplementowanie procesu trenowania sieci GAN wymaga stosowania wielu heurystycznych rozwiązań, a także dokładnego dostrajania parametrów.

Kup ksi k Pole ksi k



Rysunek 8.18. Pobaw się w dyskryminator: w każdej kolumnie znajdują się dwa obrazy wygenerowane przez sieć GAN i jeden obraz pochodzący z treningowego zbioru danych. Czy możesz odróżnić je od siebie? (Poprawne odpowiedzi: prawdziwe obrazy w kolejnych kolumnach umieszczono na środku, u góry, u dołu i na środku)

- Sieci GAN mogą potencjalnie wygenerować bardzo realistyczne obrazy, ale w przeciwieństwie do koderów VAE tworzona przez nie niejawna przestrzeń nie ma uporządkowanej ciągłej struktury, a więc nie nadaje się do niektórych praktycznych zastosowań, takich jak edycja obrazów przy użyciu wektorów koncepcyjnych.

8.6. Podsumowanie rozdziału

- Techniki uczenia głębokiego mogą być używane w wielu kreatywnych zastosowaniach wykraczających poza nadawanie etykiet — można z nich korzystać w celu generowania różnych treści. W tym rozdziale opisałem:
  - Generowanie kolejnych elementów sekwencji danych. Technika ta sprawdza się podczas generowania tekstu, muzyki (nuta po nucie), a także innych danych mających formę szeregu czasowego.
  - Działanie algorytmu DeepDream (maksymalizowanie aktywacji warstwy konwolucyjnej przez wzrost gradientu przestrzeni wejściowej).
  - Przeprowadzanie operacji transferu stylu (łączenie treści jednego obrazu ze stylem drugiego obrazu w celu wygenerowania ciekawych efektów wizualnych).

  - Sieci GAN i kodery VAE, a także używanie tych technik w celu generowania nowych obrazów. Ponadto poruszyłem zagadnienia związane z używaniem wektorów koncepcyjnych podczas edycji obrazów.
- To zaledwie kilka technik należących do szybko rozwijającej się dziedziny. Istnieje jeszcze wiele innych rozwiązań — generatywne uczenie głębokie to dziedzina zasługująca na przedstawienie w odrębnej książce.

Kup ksi k Pole ksi k

Skorowidz

A

abstrakcja, 347
abstrakcyjne modele, 339
aktywacja, 169
algorytm, 40
  DeepDream, 290–296
  Grad-CAM, 183
  lasów losowych, 35
  LSTM, 38, 211, 285
  propagacji wstecznej, 28, 66, 343
  spadku gradientowego, 252, 303
  stochastycznego spadku wzdłuż gradientu, 64, 176
  walidacji k-składowej, 102
analiza obrazu, 137
anatomia sieci neuronowej, 72
antropomorfizacja modeli, 337
API, 78
architektura sieci, 45, 330
augmentacja danych, 151, 153
autoenkodery, 306, 308

B

baza konwolucyjna, 158
  jednowymiarowa, 240
błąd przewidywań, 110
brzytwa Ockhama, 121
budowa sieci neuronowej, 147

C

całkowita strata wariacji, 301
chciwe próbkowanie, 282
chmura, 80
  AWS, 359

D

dane
  docelowe, 110
  objętościowe, 331
  obrazy, 52
  pogodowe, 215
  sekwencyjne, 51, 52, 331
  szeregu czasowego, 51, 52, 215
  tekstowe, 188
  warunkujące, 282
  wektorowe, 51, 330
  wideo, 53
  wyjściowe, 110
DeepDream, 290–296
definiowanie tensora straty, 175
destylacja danych, 45
dopasowanie słabe, 118
dopełnianie, 140
dostrajanie, 155, 163
drzewo decyzyjne, 35
dwukierunkowe
  rekurencyjne sieci neuronowe, 228
  warstwy rekurencyjne, 214
dynamiczne dostrajanie wartości, 261
dyskryminator sieci GAN, 319
działanie
  algorytmu DeepDream, 293
  sieci konwolucyjnej, 136, 139
  uczenia głębokiego, 27
dźwięk, 330

E

ekstrakcja cech, 155, 159, 168
  z augmentacją danych, 161
entropia krzyżowa, 86
etykieta, 44, 46, 110

Kup książkę
Poleć książkę

ewaluacja
  jednowymiarowej sieci konwolucyjnej, 236
  modelu, 222, 225

F

filtrowanie grafiki, 175
funkcja
  compile(), 46, 68
  generator, 218
  str(), 45
  wizualizacji kanału, 171
funkcje
  aktywacji, 40, 86
  aktywacji ostatniej warstwy, 128
  celu, 28
  ciągłe, 61
  gładkie, 62
  jądra, 34
  krzywizna, 63
  minimum lokalne, 65
  próbkujące, 286
  różniczkowalne, 62
  straty, 28, 46, 74, 97, 128
funkcjonalny interfejs API, 247, 251

G

GAN, generative adversarial network, 306, 315, 322
generator
  danych, 240
  sieci GAN, 318
  wsadów, 150
generatywne
  sieci rekurencyjne, 281
  sieci z przeciwnikiem, GAN, 306, 315
generowanie
  danych, 279
  danych sekwencyjnych, 282
  obrazów, 306
  przewidywań, 90, 96, 170
  siatki wzorców, 178
  tekstu, 281, 285, 287
  wykresów, 150, 161
  wykresów charakterystyki modelu, 223
  zdjęć, 152
gęste
  połączenie, 45
  próbkowanie, 339
głębia uczenia głębokiego, 26
głęboka
  konwolucyjna sieć GAN, 316
  sieć neuronowa, 27
gradient, 61, 62
  mini-batch, 64
  straty, 176
gradientowe wzmacnianie maszyn, 35
grafy, 248
  skierowane acykliczne, 254
  warstw, 246, 254
granice decyzyjne, 34

H

haszowanie z gorącą jedynką, 191
hiperparametry modelu, 111, 272
historia uczenia maszynowego, 32

I

idea
  śniącej maszyny, 281
  transferu stylu, 297
iloczyn tensorowy, 55
implementacja
  algorytmu DeepDream, 291
  algorytmu LSTM, 285
  jednowymiarowej sieci konwolucyjnej, 235
  neuronowego transferu stylu, 299
  rekurencyjnej sieci neuronowej, 204
  sieci GAN, 316
  trenowania modelu GAN, 321
incepcja, 254
  ekstremalna, 256
instalowanie pakietu Keras, 77, 353, 364
interfejs
  API, 247
  programistyczny pakietu Keras, 246
interpretacja geometryczna
  operacji tensorowych, 58
  uczenia głębokiego, 59
inwestycje, 40
iteracyjna walidacja k-krotna, 127

J

jądro konwolucji, 138
jednoetykietowa klasyfikacja kategorialna, 331
jednostka ukryta, 83
jednowymiarowe sieci konwolucyjne, 187, 241

K

Keras, 75
  instalowanie pakietu, 77
  uruchamianie pakietu, 79
klasa, 44, 110
klasyfikacja
  binarna, 110, 331
  kategorialna, 332
  wieloetykietowa, 110
  wieloklasowa, 91, 110
klasyfikator, 160
kodery VAE, 314
kodowanie
  kategorialne, 93
  słów, 189
konfiguracja
  algorytmu DeepDream, 292
  algorytmu spadku gradientowego, 303
  optymalizacji, 128
konwersja zbioru danych, 202
konwolucja
  głębokości, 270
  punktowa, 255
konwolucyjne sieci neuronowe, 134, 167, 185, 234
kroki procesu konwolucji, 140
krzywa ROC, 127

L

lasy losowe, 35
losowa inicjalizacja, 60
losowanie kolejności próbek, 114
LSTM, Long Short Term Memory, 38

Ł

łączenie
  danych sekwencyjnych, 235
  pochodnych, 66
  sieci konwolucyjnych i rekurencyjnych, 237

M

macierz, 48
  Grama, 298
  osadzeń słów, 199
  transpozycja, 58
maksymalizacja
  marginesu, 34
  straty, 176
mapa
  ciepła aktywacji klasy, 184
  odpowiedzi filtra, 138
mapowanie
  danych szeregów czasowych, 335
  danych wektorowych, 335
  materiału wideo, 336
  na dane wektorowe, 335
  obrazów
    na obrazy, 336
    na tekst, 336
  tekstu
    na obrazy, 336
    na tekst, 335
maszyny wzmacniane gradientowo, 35
materiały wideo, 51
metoda
  gorącej jedynki, 189
  plot(), 89
metody jądrowe, 33
metryki, 46
  trenowania, 150, 213
  walidacji, 213
model, 326
  jako program, 342
  języka, 282, 289
  klasyfikacji tekstu, 264
  pełniący funkcję warstw, 259
  sekwencyjny, 246
  wielowejściowy, 251
  z wieloma wejściami, 249
  z wieloma wyjściami, 247, 251, 253
modelowanie probabilistyczne, 32
moduł Inception, 247, 254
modułowe procedury składowe, 345
moduły
  algorytmiczne, 346
  geometryczne, 346

modyfikowanie obrazów, 290
monitorowanie
    metryk modelu, 266
    modeli, 260

N

naddatek danych, 114
nadmierne dopasowanie, 47, 89, 118, 129, 168, 225
narzędzie TensorBoard, 260, 264
neuronowy transfer stylu, 297
n-gramy, 188
niejawna przestrzeń obrazów, 306
normalizacja
    danych, 218
    gradientu, 176
    wartości, 115
    wsadu, 269

O

obrazy, 51, 52, 330
obróbka cech, 36
obsługa
    etykiet, 97
    tensorów R, 50
odkładanie danych, 127
odrzucanie rekurencyjne, 214
odwrotne różniczkowanie, 66
ograniczenia uczenia głębokiego, 336
operacja
    max-pooling, 141
    relu, 54
operacje tensorowe, 53, 55
    interpretacja geometryczna, 58
operator
    potokowy, 67
    przypisania wielokrotnego, 82
optymalizacja, 118
    gradientowa, 60
    hiperparametru, 272
optymalizator, 46, 65, 69, 74
osadzanie słów, 192, 197
    trenowane wcześniej, 196
oś czasu, 114

P

pakiet
    Keras, 75, 353
    TensorFlow, 264
parametr warstwy, 28
parsowanie pliku
    osadzeń, 198
    tekstowego, 285
percepcja, 326
pętla trenowania, 28, 60
pobieranie
    danych, 144
    osadzeń słów, 198
pochodna
    funkcji, 61
    operacji tensorowej, 62
pojemność pamięci, 119
połączenia szczątkowe, 256
porzucanie, 123, 124
prawda, 110
problem
    prognozowania temperatury, 214
    zaniku gradientu, 209
procedury numeryczne BLAS, 54
procesor graficzny, 80
    EC2, 359
prognozowanie temperatury, 214
propagacja
    gradientu, 40
    wsteczna, 66
próbka, 44
    danych wejściowych, 110
próbkowanie, 282, 306
    niejawnej przestrzeni, 311
    siatki punktów, 313
    stochastyczne, 283
przepływ roboczy, 125
    uczenia maszynowego, 329
przestrzeń hipotez, 26, 86, 330
    możliwości, 74, 334
przetwarzanie
    cech, 116
    mapy ciepła, 183
    obrazu, 133
    sekwencji, 234
    tekstu i sekwencji, 187

Kup ksi k Pole ksi k

tensora na obraz, 177
zdjęcia, 170
przygotowywanie
danych, 127, 217
generatorów, 220
punkt odniesienia, 220, 223

R

redukcja rozmiaru sieci, 119
regresja, 99
skalarna, 110
wektora ciągłych wartości, 332
wektorowa, 110
regularyzacja wag, 121
rekurencyjna maska odrzucania, 225
rekurencyjne
porzucanie, 225
sieci neuronowe, 187, 203, 242, 334
reprezentacja danych, 47, 114
rozkład
normalny, 317
prawdopodobieństwa, 94, 284
różniczkowalność, 61, 62
różniczkowanie symboliczne, 66

S

schemat
inicjacji wag, 40
modułu Inception, 255
optymalizacji, 40
sieci neuronowe, 26, 35, 328
anatomia, 72
głębokie, *Patrz także* uczenie głębokie
tworzenie, 276
konwolucyjne, 36, 134, 167, 185, 234, 330, 333
dodawanie klasyfikatora, 162
dostrajanie, 155, 163
działanie, 136
ekstrakcja cech, 155, 159, 161
implementacja, 235
jednowymiarowe, 187, 235, 241
klasyfikacja, 135
o separowalnej głębokości, 333
trenowanie, 135, 143
tworzenie, 147
wizualizacja efektów uczenia, 168
wizualizacja filtrów, 175
wstępna obróbka danych, 148
wytrenowane uprzednio, 155
z warstwą odrzucania, 153
zamrażanie bazy, 162
zapisywanie modelu, 154
optymalizacja gradientowa, 60
rekurencyjne, 187, 203, 242, 334
dwukierunkowe, 228
reprezentacja danych, 47
samonormalizujące się, 270
warstwa, 45, 73
sieć
dekodera VAE, 311
DCGAN, 316, 320
dyskryminatora, 315
GAN, 306, 315, 322
generatora, 315
gęsto połączona, 330, 331
kodera VAE, 310
LSTM, 281
silnik sieci neuronowych, 60
skalar, 48, 252
skalowanie
max-pooling, 141
obrazu, 293
w górę, 129
modeli, 274
słabe dopasowanie, 118
spadek gradientowy, 65
sterowanie procesem trenowania modelu, 260
stochastyczny spadek wzdłuż gradientu, 63
stos warstw rekurencyjnych, 226
strata, 69
regularyzacji, 310
rekonstrukcji, 310
stylu, 298, 301
treści, 298, 301
wariacji, 301
suma dwóch wektorów, 59
synteza programów, 342
systemy eksperckie, 31
szczątkowe połączenia, 247
szeregi czasowe, 330
sztuczka jądra, 34
sztuczna inteligencja, 22, 31, 326

Kup ksi k
Pole ksi k

Ś

środowisko RStudio Server, 359

T

tekst, 188, 330
tensor, 48
  czterowymiarowy, 53
  krojenie, 50
  kształt, 49
  liczba osi, 49
  operacje, 53
  oś próbek, 50
  oś wsadu, 51
  pięciowymiarowy, 48
  straty, 175
  trójwymiarowy, 49
  typ danych, 49
  zmiana kształtu, 57
test Turinga, 23
token, 188
tokenizacja tekstu, 198
transfer stylu, 297, 303
transpozycja macierzy, 58
trenowanie, 60, 68
  dwukierunkowej warstwy
    GRU, 232
    LSTM, 231
  jednowymiarowej sieci konwolucyjnej, 236
  klasyfikatora, 160
  kodera VAE, 312
  konwolucyjnej sieci neuronowej, 143
  modelu, 87, 200, 222–227
  modelu języka, 286
  sieci DCGAN, 320
  sieci konwolucyjnej, 135
tworzenie
  punktów kontrolnych, 261
  stosów warstw rekurencyjnych, 214
  wywołań zwrotnych, 262

U

Ubuntu
  instalowanie pakietu Keras, 353
uczenie częściowo nadzorowane, 109
uczenie
  częściowo nadzorowane, 109
  głębokie, 24, 29, 36, 187, 326
    działanie, 27
    generowanie danych, 279
    głębia, 26
    interpretacja geometryczna, 59
    monitorowanie modeli, 260
    najlepsze praktyki, 245
    ograniczenia, 336
    prostota, 42
    skalowalność, 42
    stacja robocza, 79
    uruchamianie w chmurze, 80
    wąskie gardła, 257
    wielokrotne stosowanie, 42
    zanik gradientów, 258
  maszynowe, 22, 32, 37, 107, 221, 326
    model programowania, 23
    uniwersalny przepływ roboczy, 125
    zautomatyzowane, 344
  nadzorowane, 108
  nienadzorowane, 108
  osadzeń słów, 193
  płytkie, 26
  przez wzmacnianie, 109
udostępnianie wag warstwy, 258
uogólnianie
  ekstremalne, 339
  lokalne, 339
uruchamianie pakietu Keras, 79

W

waga warstwy, 28, 60, 73, 258, 326
walidacja, 87, 94, 102–104
  k-krotna, 127
    krzyżowa, 113
    z losowaniem, 114
  krzyżowa, 101
  na odłożonych danych, 112
wariacyjne autoenkodery, 306, 308
wariancja, 101
warstwa, 45, 72, 326
  embedding, 194, 195, 199
  GRU, 209, 225, 227, 240
  LSTM, 209, 210, 212, 230

Kup książkę Poleć książkę

warstwy
  gęsto połączone, 331
  konwolucyjne, 235, 255, 332
  pośrednie, 97
  rekurencyjne, 73, 206, 226
  różniczkowalne, 343
wczesne
  sieci neuronowe, 33
  zatrzymywanie, 261
wczytywanie obrazów, 149
wektor, 48
  pięciowymiarowy, 48
  uśmiechu, 308
wektory
  koncepcyjne, 307
  słów, 192
wektoryzacja
  danych, 115
  tekstu, 188
wideo, 53, 331
wiedza
  modelu, 326
  sieci, 73
wizualizacja
  danych, 264
  działania biblioteki TensorFlow, 268
  efektów uczenia, 168
  filtrów, 175, 177
  kanału, 171
  map ciepła, 181
  osadzeń słów, 267
  pośrednich aktywacji, 169
worki słów, 189
wsad danych, 50, 110
współczynnik porzucania, 123
współdzielona sieć LSTM, 258
wstępna obróbka obrazu, 182
wstępne przetwarzanie osadzeń, 198
wydajność modelu, 127, 200
wymiar
  próbek, 50
  wsadu, 51
wymiarowość, 48
wywołania zwrotne, 260, 261
wzmacnianie gradientowe, 35
wzorce filtrów warstwy, 179

Z

zamrażanie warstwy, 162
zapisywanie
  metryk trenowania, 261
  modelu, 150
zastosowania
  rekurencyjnych sieci neuronowych, 214
zbiory
  testowe, 45, 111
  treningowe, 45, 111
  walidacyjne, 111
zbiór danych
  Agencji Reutera, 91
  danych IMDB, 81
  MNIST, 44
zima sztucznej inteligencji, 30
zmiana układu współrzędnych, 25

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

W ostatnich latach byliśmy świadkami ogromnego postępu technik sztucznej inteligencji, uczenia maszynowego oraz uczenia głębokiego. Konsekwencje tego błyskawicznego rozwoju są odczuwalne w niemal każdej dziedzinie. Wydaje się, że to jedna z tych technologii, które powinny być dostępne dla jak najszerszej grupy ludzi. Dopiero wtedy uczenie głębokie wykorzysta w pełni swój potencjał i stanie się prawdziwym impulsem rozwoju naszej cywilizacji. Co prawda na pierwszy rzut oka ta niesamowita technologia może wydawać się wyjątkowo skomplikowana i trudna do zrozumienia, warto jednak wykorzystać dostępne narzędzia, takie jak biblioteka Keras i język R, aby implementować mechanizmy uczenia głębokiego wszędzie tam, gdzie okażą się przydatne.

Ta książka jest znakomitym przewodnikiem po technikach uczenia głębokiego. Poza wyczerpująco przedstawionymi podstawami znajdziesz tu zasady implementacji tych technik z wykorzystaniem języka R i biblioteki Keras. Dzięki przystępnym wyjaśnieniom i praktycznym przykładom szybko zrozumiesz nawet bardziej skomplikowane zagadnienia uczenia głębokiego. Poznasz koncepcje i dobre praktyki związane z tworzeniem mechanizmów analizy obrazu, przetwarzania języka naturalnego i modeli generatywnych. Przeanalizujesz ponad 30 przykładów kodu uzupełnionego dokładnymi komentarzami. W efekcie szybko przygotujesz się do korzystania z uczenia głębokiego w celu rozwiązania konkretnych problemów.

W tej książce między innymi:

- podstawowe koncepcje sztucznej inteligencji, uczenia maszynowego i uczenia głębokiego
- wprowadzenie do budowy i trenowania sieci neuronowych
- uczenie głębokie w przetwarzaniu obrazów
- modele generatywne tworzące obrazy i tekst
- perspektywy i ograniczenia uczenia głębokiego

Uczenie głębokie: zafascynuj się i zaimplementuj!

François Chollet zajmuje się uczeniem głębokim w firmie Google. Jest twórcą biblioteki Keras, a także współtwórcą pakietu TensorFlow. Zajmuje się również rozwojem technik uczenia głębokiego związanych z przetwarzaniem obrazu oraz procesami logicznego myślenia.

J.J. Allaire jest twórcą zintegrowanego środowiska programistycznego RStudio. Opracował interfejsy pozwalające na stosowanie pakietów Keras i TensorFlow podczas pracy w R.

ISBN 978-83-283-4780-9

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 77,00 zł