François Chollet

DEEP LEARNING

Praca z językiem Python i biblioteką Keras

Helion

Tytuł oryginału: Deep Learning with Python

Tłumaczenie: Konrad Matuk

ISBN: 978-83-283-4778-6

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)

Dodatkowe materiały do książki można znaleźć pod adresem: *ftp://ftp.helion.pl/przyklady/delepy.zip*

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

3 Rozpoczynamy korzystanie z sieci neuronowych

W tym rozdziale opisałem:

- Najważniejsze komponenty sieci neuronowych.
- Wprowadzenie do pakietu Keras.
- Konfigurację stacji roboczej przeznaczonej do uczenia głębokiego.
- Stosowanie sieci neuronowych w celu rozwiązywania podstawowych problemów klasyfikacji i regresji.

Lektura tego rozdziału ma pozwolić Ci rozpocząć stosowanie sieci neuronowych w celu rozwiązywania prawdziwych problemów. Podsumuję wiadomości związane z pierwszym praktycznym przykładem, opisanym w rozdziale 2. Zastosujesz zdobytą wiedzę w celu rozwiązania trzech nowych problemów dotyczących trzech najczęstszych zastosowań sieci neuronowych: klasyfikacji binarnej, klasyfikacji wieloklasowej i regresji skalarnej.

W tym rozdziale przyjrzymy się bliżej głównym elementom sieci neuronowych wprowadzonych w rozdziale 2.: warstwom, sieciom, funkcjom celu i optymalizatorom. Przedstawię krótkie wprowadzenie do pakietu Keras — biblioteki Pythona przeznaczonej do uczenia głębokiego, z której będziemy korzystać w dalszej części tej książki. Skonfigurujesz swój komputer pod kątem uczenia głębokiego — zainstalujesz pakiet TensorFlow, Keras i włączysz obsługę układu graficznego. Rozwiązywanie prawdziwych problemów za pomocą sieci neuronowych przedstawię na podstawie trzech przykładów. Będą to:

Kup książkę Poleć książkę

- klasyfikacja recenzji filmów — dzielenie ich na recenzje pozytywne i negatywne (klasyfikacja binarna);
- podział wiadomości na tematy (klasyfikacja wieloklasowa);
- szacowanie ceny domu na podstawie danych opisujących nieruchomość (regresja).

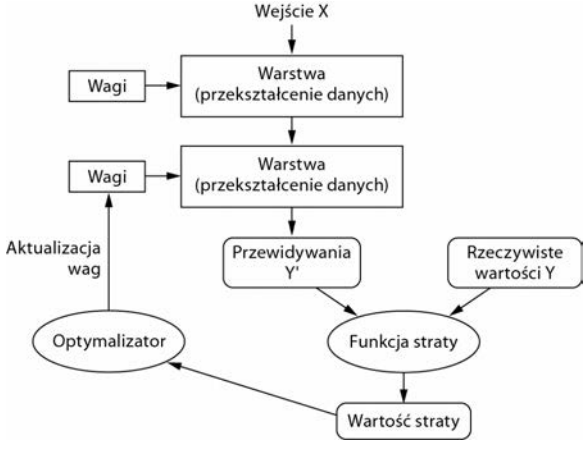
Po przeczytaniu tego rozdziału będziesz w stanie używać sieci neuronowych do rozwiązywania prostych problemów takich jak klasyfikacja lub regresja wektora danych. Lektura tego rozdziału ma Cię przygotować do zrozumienia teorii działania uczenia maszynowego przedstawionej w rozdziale 4.

3.1. Anatomia sieci neuronowej

Przypominam, że trenowanie sieci neuronowej jest związane z rzeczami, którymi są:

- **warstwy**, które po połączeniu ze sobą tworzą **sieć** (lub **model**);
- **dane wejściowe** i odpowiadające im **docelowe etykiety**;
- **funkcja straty**, która definiuje sygnał zwrotny używany w procesie uczenia;
- **optymalizator**, który określa przebieg trenowania.

Zależności między nimi przedstawiono na rysunku 3.1. Sieć składająca się z połączonych ze sobą warstw przypisuje przewidywane wartości wyjściowe do danych wejściowych. Następnie funkcja straty porównuje wyniki przewidywań sieci z docelowymi etykietami, wskutek czego obliczana jest wartość straty (miara tego, czy sieć zwraca oczekiwane wartości). Optymalizator korzysta z wartości straty podczas modyfikowania wag sieci.



Rysunek 3.1. Zależności między siecią, jej warstwami, funkcją straty a optymalizatorem

W kolejnych sekcjach opiszę, czym są warstwy, sieci, funkcje strat i optymalizatory.

3.1.1. Warstwy: podstawowe bloki konstrukcyjne uczenia głębokiego

Warstwa jest podstawową strukturą danych sieci neuronowych (pojęcie to wprowadziłem w rozdziale 2.). Warstwa jest modułem przetwarzania danych, który przyjmuje dane wejściowe z jednego tensora lub kilku tensorów i generuje dane wyjściowe

Kup książkę Poleć książkę

w postaci tensora lub kilku tensorów. Niektóre warstwy nie mają stanów, ale najczęściej warstwy są charakteryzowane przez stan — **wagę**. Wagę tworzy tensor lub kilka tensorów. Wartości wag są ustalane przy użyciu algorytmu stochastycznego spadku wzdłuż gradientu. Wagi tworzą **wiedzę** sieci.

Do przetwarzania różnych typów danych i tensorów o różnych formatach stosuje się różne typy warstw. Proste dane wektorowe są przechowywane w dwuwymiarowych tensorach mających kształt(`próbki, cechy`) — przetwarza się je za pomocą warstw **gęsto połączonych**, zwanych również **warstwami w pełni połączonymi** lub po prostu warstwami **gęstymi** (ich implementacją jest klasa `Dense` pakietu Keras). Dane sekwencyjne są przechowywane w trójwymiarowych tensorach mających kształt (`próbki, znaczniki_`
`↪czasu, cechy`) — przetwarza się je zwykle za pomocą warstw **rekurencyjnych** (np. warstw `LSTM`). Dane obrazu przechowuje się w tensorach czterowymiarowych — przetwarza się je zazwyczaj za pomocą dwuwymiarowych warstw konwolucyjnych (`Conv2D`).

Warstwy można porównać do klocków LEGO uczenia głębokiego. Z metafory tej korzystali między innymi twórcy pakietu Keras. Modele głębokie są budowane w tym pakiecie poprzez łączenie ze sobą kompatybilnych warstw, co umożliwia generowanie praktycznych potoków przekształcających dane. Pojęcie **kompatybilności warstw** odnosi się do tego, że każdy typ warstwy przyjmuje tylko tensory wejściowe o określonym kształcie i doprowadza do powstania tensorów wyjściowych mających określony kształt. Przyjrzyj się następującemu przykładowi:

```
from keras import layers

layer = layers.Dense(32, input_shape=(784,))
```

Warstwa gęsta z 32 jednostkami wyjściowymi.

Utworzyliśmy warstwę, która przyjmuje na wejściu tylko tensory dwuwymiarowe, a pierwszy wymiar tensora musi mieć długość 784 (oś 0 — wymiar próbki — jest nieokreślony, a więc warstwa akceptuje jego dowolną wielkość). Warstwa ta zwróci tensor, którego pierwszy wymiar będzie miał długość równą 32.

W związku z tym warstwa ta może być połączona z kolejną warstwą, która oczekuje na wejściu pojawienia się wektora o 32 wymiarach. Korzystając z pakietu Keras, nie musisz przejmować się kompatybilnością, ponieważ warstwy tworzące model są dynamicznie budowane w celu dopasowania ich do poprzedniej warstwy. Załóżmy, że napisaliśmy następujący kod:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

Do drugiej warstwy nie przekazaliśmy argumentu definiującego kształt tensora wejściowego — parametr ten zostanie automatycznie określony na podstawie kształtu tensora zwracanego przez wcześniejszą warstwę.

Kup książkę Poleć książkę

3.1.2. Modele: sieci warstw

Model uczenia głębokiego jest skierowanym acyklicznym grafem warstw. Najpopularniejszą jego formą jest liniowy stos warstw przypisujący jeden element wyjściowy do jednego elementu wejściowego.

Z czasem poznasz również bardziej zaawansowane topologie sieci, takie jak:

- sieci o dwóch rozgałęzieniach,
- sieci typu multihead,
- bloki incepcji.

Topologia sieci definiuje **przestrzeń hipotezy**. W rozdziale 1. definiowaliśmy uczenie maszynowe jako „poszukiwanie praktycznej reprezentacji danych wejściowych w ramach zdefiniowanej przestrzeni możliwości na podstawie sygnału informacji zwrotnej". Wybierając topologię sieci, ograniczamy **przestrzeń możliwości** (przestrzeń hipotez) do określonej serii operacji tensorowych przypisujących dane wyjściowe do danych wejściowych. Trenowanie sieci polega na poszukiwaniu odpowiedniego zestawu wartości wag operacji przetwarzania tensorów.

Wybór właściwej architektury sieci to raczej kwestia doświadczenia niż analizy naukowej. Co prawda są pewne zasady doboru architektury do problemu, z których możesz korzystać, ale tylko praktyka sprawi, że będziesz w stanie zrobić to naprawdę dobrze. W kolejnych rozdziałach poznasz te zasady, ale pomogę Ci również w zdobyciu pewnej intuicji w tej kwestii, co ułatwi Ci wykonanie tego zadania.

3.1.3. Funkcja straty i optymalizatory: najważniejsze elementy konfiguracji procesu uczenia

Po zdefiniowaniu architektury sieci musisz określić jeszcze dwie rzeczy:

- **Funkcję straty** (**funkcję celu**) — wartość, która będzie minimalizowana w procesie trenowania. Jest miarą sukcesu wykonywanego zadania.
- **Optymalizator** — sposób modyfikowania sieci na podstawie funkcji straty. Implementuje on określony wariant algorytmu stochastycznego spadku wzdłuż gradientu.

Sieć neuronowa, która generuje wiele wartości wyjściowych, może mieć więcej niż jedną funkcję straty (może mieć po jednej funkcji straty dla każdej wartości wyjściowej), ale proces spadku gradientu musi być oparty na *pojedynczej* skalarnej wartości straty; a więc w przypadku sieci z wieloma wartościami straty wszystkie te wartości są łączone (uśredniane) w celu uzyskania jednej wartości skalarnej.

Odpowiedni dobór funkcji celu do problemu jest bardzo ważny — sieć będzie robiła wszystko, by zminimalizować straty, a więc jeżeli funkcja celu nie będzie w pełni skorelowana z osiągnięciem sukcesu w wykonywaniu zadania, to sieć będzie wykonywała niechciane operacje. Wyobraź sobie głupią wszechmogącą sztuczną inteligencję trenowaną za pomocą algorytmu SGD przy źle dobranej funkcji celu w postaci „maksymalizuj średni dobrobyt wszystkich żywych ludzi". Sztuczna inteligencja, aby osiągnąć ten cel w sposób jak najprostszy, może zdecydować się na zabicie wszystkich ludzi poza kilkoma najbogatszymi osobami. Takie rozwiązanie jest możliwe, ponieważ na średni

Kup książkę Poleć książkę

dobrobyt nie wpływa liczba osób pozostałych przy życiu, a prawdopodobnie takiego rozwiązania nie miał na celu autor tej sztucznej inteligencji! Pamiętaj o tym, że wszystkie konstruowane przez Ciebie sztuczne sieci neuronowe będą zachowywały się podobnie — będą starały się za wszelką cenę obniżyć funkcję straty. W związku z tym musisz rozważnie dobierać cele, bo w przeciwnym razie osiągniesz niezamierzone efekty uboczne.

Na szczęście w typowych problemach, takich jak klasyfikacja, regresja i przewidywanie sekwencyjne, można korzystać z prostych wskazówek umożliwiających wybranie właściwej funkcji straty. W przypadku podziału na dwie grupy będziemy posługiwać się entropią krzyżową, a w przypadku dzielenia na wiele grup będziemy korzystać z kategoryzacyjnej entropii krzyżowej. Problem regresji będzie rozwiązywany za pomocą średniego błędu kwadratowego, a podczas pracy nad problemem uczenia sekwencyjnego będziemy korzystać z klasyfikacji CTC (ang. *Connectionist Temporal Classification*). Funkcje celu należy tworzyć samodzielnie w zasadzie tylko podczas pracy nad naprawdę nowym problemem badawczym. W kolejnych rozdziałach opiszę szczegółowo dobór funkcji straty do różnych typowych problemów.

3.2. Wprowadzenie do pakietu Keras

W przykładach kodu przedstawionych w tej książce wykorzystano pakiet Keras (*https://keras.io/*). Jest to rama projektowa uczenia głębokiego języka Python, która umożliwia wygodne definiowanie i trenowanie dowolnych modeli uczenia głębokiego. Biblioteka ta była początkowo tworzona w celu umożliwienia naukowcom szybkiego przeprowadzania eksperymentów.

Główne cechy charakteryzujące tę bibliotekę to:

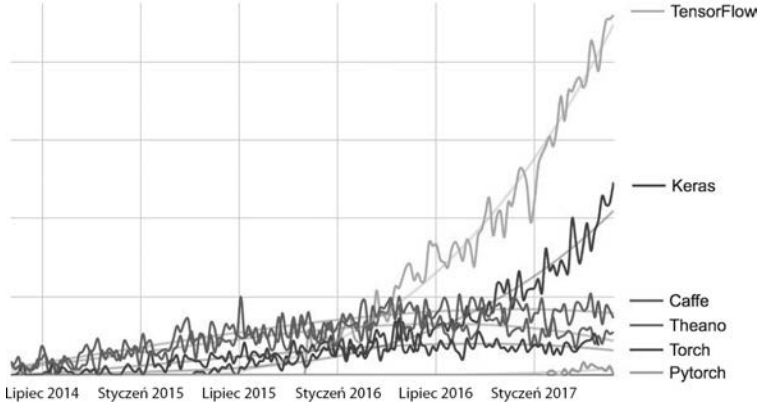
- możliwość bezproblemowego uruchamiania tego samego kodu na procesorach CPU i GPU;
- przyjazny interfejs programistyczny ułatwiający szybkie prototypowanie modeli uczenia głębokiego;
- wbudowana obsługa sieci konwolucyjnych służących do przetwarzania obrazu, sieci rekurencyjnych służących do przetwarzania danych sekwencyjnych i sieci będących połączeniem obu tych rozwiązań;
- obsługa sieci o dowolnych architekturach: modeli obsługujących wiele wejść i wyjść; możliwość współdzielenia warstw i modeli. Dzięki tym możliwościom pakiet Keras nadaje się do budowania praktycznie dowolnych modeli uczenia głębokiego — od generatywnych sieci z przeciwnikiem (sieci GAN) do neuronowych maszyn Turinga.

Pakiet Keras jest dystrybuowany na podstawie liberalnej licencji MIT, a więc można go używać za darmo nawet w projektach komercyjnych. Jest kompatybilny z dowolną wersją Pythona od 2.7 do 3.6 (stan na pierwszą połowę 2017 r.).

Z biblioteki Keras korzysta ponad 200 000 użytkowników — należą do nich naukowcy, inżynierowie, pracownicy małych firm i dużych korporacji, absolwenci uczelni wyższych i zwykli entuzjaści nowych technologii. Z Keras korzystają takie firmy jak Google, Netflix,

Kup książkę Poleć książkę

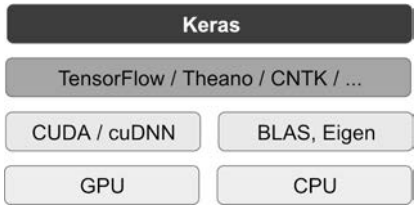
Uber, CERN, Yelp, Square, a także setki małych start-upów pracujących nad różnymi problemami. Biblioteka ta jest również popularna wśród uczestników konkursów Kaggle — praktycznie każdy konkurs dotyczący uczenia głębokiego wygrywa ostatnio model korzystający z Keras.



Rysunek 3.2. Wykres zmian zainteresowania różnymi pakietami uczenia maszynowego sporządzony na podstawie danych wyszukiwarki Google

3.2.1. Keras, TensorFlow, Theano i CNTK

Keras jest biblioteką funkcjonującą na poziomie modelu. Zapewnia ona wysokopoziomowe bloki konstrukcyjne służące do tworzenia modeli uczenia głębokiego. Pakiet ten nie obsługuje niskopoziomowych operacji takich jak przeprowadzanie działań na tensorach i różniczkowanie. W tych kwestiach polega on na wyspecjalizowanej i solidnie zoptymalizowanej bibliotece obsługującej tensory. Pełni ona funkcję **bazowego silnika** pakietu Keras. Autorzy Keras nie wybrali jednej biblioteki obsługującej tensory i nie powiązali implementacji pakietu Keras z żadną konkretną biblioteką. Problem ten został rozwiązany modułowo (patrz rysunek 3.3). Keras może korzystać z kilku różnych silników bazowych. Obecnie są to biblioteki TensorFlow, Theano i Microsoft Cognitive Toolkit (CNTK). W przyszłości zbiór ten może zostać powiększony o kolejne biblioteki bazowe uczenia głębokiego.



Rysunek 3.3. Stos programowy i sprzętowy uczenia głębokiego

Obecnie TensorFlow, CNTK i Theano są podstawowymi platformami uczenia głębokiego. Platforma Theano (*http://deeplearning.net/software/theano*) jest rozwijana przez laboratorium MILA Uniwersytetu Montrealskiego, platforma TensorFlow (*http://www.tensorflow.org*) jest rozwijana przez firmę Google, a platforma CNTK (*https://github.com/Microsoft/CNTK*) — przez Microsoft. Każdy kod korzystający z Keras może być uruchomiony na dowolnej platformie wybranej spośród tych trzech. Kod nie będzie

Kup książkę Poleć książkę

wymagał żadnych modyfikacji. Platformę można zmienić w dowolnym momencie pracy, co przyda się, gdy któraś platforma okaże się szybsza podczas rozwiązywania danego problemu. Polecam traktowanie platformy TensorFlow jako platformy domyślnej. Jest ona najpopularniejsza. Charakteryzuje ją możliwość skalowania i wdrażania w rozwiązaniach produkcyjnych.

Dzięki platformie TensorFlow (lub Theano albo CNTK) pakiet Keras może korzystać z możliwości procesorów CPU i GPU. W przypadku wykonywania kodu na procesorze CPU platforma TensorFlow obudowuje niskopoziomową bibliotekę operacji tensorowych o nazwie Eigen (*http://eigen.tuxfamily.org*). W przypadku pracy na procesorze GPU platforma TensorFlow obudowuje wysoce zoptymalizowaną bibliotekę uczenia głębokiego NVIDIA CUDA Deep Neural Network (cuDNN).

3.2.2. Praca z pakietem Keras: krótkie wprowadzenie

Przedstawiłem już jeden przykład modelu opartego na pakiecie Keras — przykład MNIST. Typowy przepływ roboczy podczas pracy z tym pakietem wygląda tak, jak pokazano we wspomnianym przykładzie:

1. Zdefiniuj dane treningowe: tensory wejściowe i tensory wartości docelowych.
2. Zdefiniuj warstwy sieci (lub *modelu*) przypisującej dane wejściowe do docelowych wartości.
3. Skonfiguruj proces uczenia, wybierając funkcję straty, optymalizator i monitorowane metryki.
4. Wykonaj iteracje procesu uczenia na danych treningowych, wywołując metodę `fit()` zdefiniowanego modelu.

Model może zostać zdefiniowany na dwa sposoby: przy użyciu klasy `Sequential` (tylko w przypadku liniowych stosów warstw, czyli obecnie najpopularniejszej architektury sieci) lub **funkcjonalnego interfejsu API** (w przypadku skierowanych acyklicznych grafów warstw — interfejs ten pozwala na tworzenie sieci o dowolnej architekturze).

Dla przypomnienia — oto przykład dwuwarstwowego modelu zdefiniowanego za pomocą klasy `Sequential` (zauważ, że określono w nim oczekiwany kształt danych wejściowych pierwszej warstwy sieci):

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

Oto definicja tego samego modelu napisana przy użyciu funkcjonalnego interfejsu API:

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)

model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Kup książkę Poleć książkę

Korzystając z funkcjonalnego interfejsu API, przeprowadzasz operacje na tensorach danych przetwarzanych przez model — przetwarzasz tensory przez warstwy sieci tak, jakby były funkcjami.

UWAGA W rozdziale 7. znajdziesz szczegółowy opis możliwości funkcjonalnego interfejsu API. Do tego rozdziału w prezentowanych przykładach kodu będę korzystał tylko z klasy `Sequential`.

Po zdefiniowaniu architektury modelu nie ma znaczenia to, czy został on utworzony za pomocą klasy `Sequential`, czy posługiwałeś się funkcjonalnym interfejsem API. W obu przypadkach następne operacje przebiegają tak samo.

Proces uczenia jest konfigurowany w kroku kompilacji, w którym należy określić optymalizator i funkcję straty (niektóre modele mogą mieć kilka funkcji strat) — elementy, z których powinien korzystać model, a także metryki, które mają być stosowane do monitorowania procesu trenowania. Oto przykład, w którym zastosowano jedną funkcję straty (jest to najczęstszy przypadek):

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
                        loss='mse',
                        metrics=['accuracy'])
```

Ostatni proces — etap uczenia polega na przekazywaniu tablic Numpy zawierających dane wejściowe (i docelowe dane — etykiety próbek) do modelu przy użyciu metody `fit()`. Proces ten przebiega podobnie do procesu używanego podczas pracy z biblioteką Scikit-Learn, a także kilkoma innymi bibliotekami uczenia maszynowego:

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Podczas lektury kilku kolejnych rozdziałów będziesz w stanie intuicyjnie określić typ architektury sieci, który sprawdzi się w danym problemie. Nauczysz się konfigurować proces uczenia i dostrajać model aż do uzyskania oczekiwanych rezultatów. W podrozdziałach 3.4, 3.5 i 3.6 opiszę trzy podstawowe przykłady: przykład klasyfikacji dzielącej na dwie grupy, przykład klasyfikacji dzielącej na wiele grup i przykład regresji.

3.3. Przygotowanie stacji roboczej do uczenia głębokiego

Zanim rozpoczniesz pracę nad tworzeniem aplikacji uczenia głębokiego, musisz skonfigurować swoją stację roboczą. Zalecam wykonywanie kodu uczenia głębokiego na nowoczesnym procesorze graficznym NVIDIA, ale nie jest to konieczne. Niektóre aplikacje, a szczególnie aplikacje przetwarzające obraz za pomocą konwolucyjnych sieci neuronowych i aplikacje przetwarzające sekwencje za pomocą rekurencyjnych sieci neuronowych będą działały bardzo wolno na zwykłym procesorze CPU. Dotyczy to nawet szybkich wielordzeniowych procesorów tego typu. Nawet aplikacje, które mogą być uruchomione na zwykłym procesorze, gdy zostaną uruchomione na nowoczesnym procesorze graficznym, będą działały szybciej od 5 do nawet 10 razy. Jeżeli nie chcesz instalować procesora graficznego w swoim komputerze, to możesz przeprowadzać swoje eksperymenty na procesorze GPU w chmurze Google lub Amazon Web Services

Kup książkę Poleć książkę

(instancja EC2 GPU), ale musisz pamiętać o tym, że na dłuższą metę korzystanie z procesorów GPU dostępnych w chmurze jest drogie.

Niezależnie od tego, czy będziesz pracować na własnym sprzęcie, czy korzystać z chmury, najlepiej jest wybrać środowisko systemu Unix. Pomimo tego, że uruchomienie pakietu Keras w systemie Windows jest, technicznie rzecz biorąc, możliwe (wszystkie trzy silniki przetwarzania danych obsługują system Windows), to nie zalecam tego rozwiązania. W instrukcji instalacji znajdującej się w dodatku A opisałem konfigurację środowiska programistycznego w systemie Ubuntu. Jeżeli korzystasz z systemu Windows, to najprostszym rozwiązaniem jest zainstalowanie na swoim komputerze drugiego systemu operacyjnego (systemu Ubuntu). Może się to wydawać dość kłopotliwe, ale praca w Ubuntu oszczędzi Ci wiele czasu i pozwoli uniknąć kłopotów w przyszłości.

Aby móc korzystać z pakietu Keras, musisz zainstalować platformę TensorFlow, CNTK lub Theano (albo wszystkie te platformy jednocześnie, jeżeli chcesz dysponować możliwością przełączania się między nimi w dowolnej chwili). W tej książce skupię się na platformie TensorFlow, czasami będę udzielał wskazówek dotyczących platformy Theano, ale zupełnie pominę zagadnienia związane z platformą CNTK.

3.3.1. Notatniki Jupyter: najlepszy sposób na eksperymentowanie z uczeniem głębokim

Notatniki Jupyter doskonale nadają się do eksperymentowania z uczeniem głębokim — umożliwiają między innymi uruchomienie kodu opisywanego w tej książce. Jest to narzędzie popularne wśród analityków i osób zajmujących się uczeniem maszynowym. **Notatnik** jest plikiem wygenerowanym przez aplikację Jupyter Notebook (*https://jupyter.org*), który można edytować w oknie przeglądarki internetowej. Umożliwia on wykonywanie kodu Pythona i tworzenie rozbudowanych notatek opisujących jego działanie. Notatnik pozwala na podzielenie długiego eksperymentu na mniejsze etapy, które mogą być wykonywane w sposób niezależny, a więc proces pracy nad programem staje się interaktywny — nie musisz uruchamiać ponownie całego wcześniejszego kodu, jeżeli napotkasz problemy pod koniec eksperymentu.

Polecam korzystanie z notatników Jupyter na początku przygody z pakietem Keras, ale nie jest to wymóg konieczny. Możesz równie dobrze uruchamiać samodzielne skrypty Pythona lub korzystać ze środowiska programistycznego takiego jak PyCharm. Wszystkie przykłady kodu opisanego w tej książce możesz pobrać w formie notatników: *ftp://ftp.helion.pl/przyklady/delepy.zip*.

3.3.2. Dwie opcje uruchamiania pakietu Keras

Polecam Ci skorzystanie z jednej z dwóch następujących opcji w celu rozpoczęcia pracy nad uczeniem maszynowym:

- Skorzystanie z oficjalnej maszyny wirtualnej — obrazu EC2 Deep Learning AMI (*https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html*) — i uruchamianie na niej eksperymentów w postaci notatników Jupyter. Jeżeli nie posiadasz komputera z odpowiednim procesorem graficznym, to warto skorzystać z tej możliwości. Konfigurację tego rozwiązania opisano w dodatku B.

Kup książkę Poleć książkę

- Zainstalowanie wszystkich niezbędnych narzędzi od podstaw na swoim komputerze w systemie Unix. Możliwe wówczas stanie się uruchamianie notatników Jupyter i skryptów Pythona w środowisku lokalnym. Skorzystaj z tej opcji, jeżeli dysponujesz szybkim procesorem graficznym firmy NVIDIA. Proces instalacji i konfiguracji wszystkich narzędzi opisałem w dodatku A.

Przeanalizujmy wady i zalety poszczególnych rozwiązań.

3.3.3. Wady i zalety uruchamiania uczenia głębokiego w chmurze

Jeżeli nie posiadasz procesora graficznego (nowego, wysokiego modelu procesora firmy NVIDIA), który może zostać użyty w celu przetwarzania kodu uczenia maszynowego, to przeprowadzanie eksperymentów w chmurze jest prostym i tanim rozwiązaniem, które nie wymaga zakupu dodatkowego sprzętu. Uruchamianie kodu notatnika Jupyter w chmurze nie różni się niczym od uruchamiania go lokalnie. W połowie 2017 r. najprostszym rozwiązaniem umożliwiającym rozpoczęcie pracy nad uczeniem głębokim jest AWS EC2. W dodatku B znajdziesz wszystkie informacje niezbędne do uruchomienia notatników Jupyter na procesorze EC2 dostępnym w chmurze.

Jeżeli często pracujesz z tego typu kodem, to ciężko będzie Ci korzystać z tego rozwiązania, ponieważ jest ono kosztowne. Obsługa instancji opisanej przeze mnie w dodatku B (instancja `p2.xlarge`, której moc obliczeniowa wcale nie jest duża) kosztuje około 4 zł za godzinę (cena z połowy 2017 r.). Solidny konsumencki procesor graficzny kosztuje od 3500 zł do 6000 zł. Ceny tego typu układów są dość stabilne pomimo tego, że możliwości kolejnych generacji procesorów graficznych są coraz większe. Jeżeli poważnie podchodzisz do uczenia głębokiego, to powinieneś zbudować własny komputer roboczy wyposażony w przynajmniej jeden procesor graficzny.

Ogólnie rzecz biorąc, wykupienie dostępu do maszyny wirtualnej EC2 jest dobrym sposobem na rozpoczęcie pracy z algorytmami uczenia głębokiego. Instancja tej maszyny wyposażona w procesor GPU umożliwia wykonanie całego przykładowego kodu zaprezentowanego w tej książce, ale jeżeli chcesz zajmować się uczeniem głębokim na co dzień, to warto kupić własny układ graficzny.

3.3.4. Jaki procesor graficzny najlepiej nadaje się do uczenia głębokiego?

Jeżeli masz zamiar kupić procesor graficzny, to jaki warto wybrać? Przede wszystkim musi być to układ firmy NVIDIA. Tylko ten producent układów graficznych inwestuje duże pieniądze w rozwój technologii uczenia głębokiego i zarazem nowoczesne pakiety uczenia głębokiego działają tylko na kartach graficznych firmy NVIDIA.

W połowie 2017 r. za układ najlepiej nadający się do uczenia głębokiego uważałem NVIDIA TITAN Xp. Wśród tańszych rozwiązań warto rozważyć kartę GTX 1060. Jeżeli czytasz tę książkę w 2018 r. lub później, to zajrzyj do internetu i poszukaj aktualnych rekomendacji, ponieważ na rynku co roku pojawia się wiele nowych rozwiązań.

Od teraz zakładam, że dysponujesz dostępem do komputera z zainstalowanym pakietem Keras i bibliotekami zależnymi. Najlepiej byłoby, aby był to komputer wypo-

Kup książkę Poleć książkę

sażony w obsługiwany procesor graficzny. Zainstaluj wszystkie niezbędne komponenty, zanim przejdziesz dalej. Wykonaj wszystkie kroki opisane w dodatkach. W razie jakichkolwiek problemów szukaj pomocy w internecie. Znajdziesz w nim wiele poradników instalowania Keras i najczęściej używanych bibliotek zależnych.

Teraz możemy przejść do przykładów praktycznego zastosowania pakietu Keras.

3.4. Przykład klasyfikacji binarnej: klasyfikacja recenzji filmów

Klasyfikacja dzieląca na dwie grupy (klasyfikacja binarna) jest najczęściej spotykanym problemem uczenia maszynowego. Analizując ten przykład, nauczysz się klasyfikować recenzje filmów — dzielić je na pozytywne i negatywne na podstawie ich treści.

3.4.1. Zbiór danych IMDB

Będziemy pracować ze zbiorem IMDB: zbiorem 50 000 bardzo spolaryzowanych recenzji opublikowanych w serwisie Internet Movie Database. Recenzje zostały podzielone na zbiór treningowy (25 000 recenzji) i zbiór testowy (25 000 recenzji). Każdy z tych zbiorów składa się w połowie z recenzji pozytywnych i w połowie z recenzji negatywnych.

Dlaczego korzystamy z oddzielnych zbiorów? Wynika to z tego, że nigdy nie powinno się testować modelu uczenia maszynowego na tych samych danych, które były używane do jego trenowania! To, że model dobrze klasyfikuje dane treningowe, wcale nie oznacza tego, że będzie równie dobrze klasyfikował nowe dane, a tak naprawdę interesuje nas wydajność modelu podczas klasyfikacji nowych danych (znamy etykiety próbek treningowego zbioru danych, a więc to oczywiste, że nie musimy ich przewidywać za pomocą modelu). Model mógłby po prostu *zapamiętać* etykiety treningowego zbioru danych i być zupełnie nieprzydatny podczas przewidywania etykiet nowych recenzji. Zagadnienie to zostanie opisane w sposób bardziej szczegółowy w kolejnym rozdziale.

Zbiór IMDB, podobnie jak zbiór MNIST, jest dołączony do pakietu Keras. Zbiór ten został już przygotowany do analizy: recenzje (sekwencje słów) zostały zamienione na sekwencje wartości całkowitoliczbowych, w których każda wartość symbolizuje obecność w recenzji wybranego słowa ze słownika.

Poniższy kod załaduje zbiór danych (podczas uruchamiania go po raz pierwszy na dysk twardy Twojego komputera pobranych zostanie około 80 MB danych).

Listing 3.1. Ładowanie zbioru danych IMDB

```
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

Argument `num_words=10000` oznacza, że w treningowym zbiorze danych zostanie zachowanych tylko 10 000 słów, występujących najczęściej w tym zbiorze danych. Słowa występujące rzadziej zostaną pominięte. Rozwiązanie to umożliwia pracę z wektorem danych o rozmiarze umożliwiającym jego przetwarzanie.

Kup książkę Poleć książkę

Zmienne `train_data` i `test_data` są listami recenzji. Każda recenzja jest listą indeksów słów (zakodowaną sekwencją słów). Zmienne `train_labels` i `test_labels` zawierają etykiety w postaci zer i jedynek: 0 oznacza recenzję **negatywną**, a 1 oznacza recenzję **pozytywną**:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]

>>> train_labels[0]
1
```

Ograniczamy się do 10 000 najczęściej występujących słów, a więc będziemy mieli 10 000 wartości indeksów słów:

```
>>> max([max(sequence) for sequence in train_data])
9999
```

Dla ciekawskich — oto sposób na szybkie odkodowanie jednej z recenzji i odczytanie jej treści w języku angielskim:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
    [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

Słownik word_index przypisuje słowom wartości indeksów.

Odwracajac go, możemy przypisać indeksy do słów.

Kod dekodujący recenzję. Zauważ, że indeksy są przesunięte o 3, ponieważ pod indeksami o numerach 0, 1 i 2 znajdują się indeksy symbolizujące „wypełnienie", „początek sekwencji" i „nieznane słowo".

3.4.2. Przygotowywanie danych

List wartości całkowitoliczbowych nie można przekazać bezpośrednio do sieci neuronowej. Trzeba je zamienić na listę tensorów. Można to zrobić na dwa sposoby:

- Można dopełnić listy tak, aby miały takie same długości, i zamienić je na tensor wartości całkowitoliczbowych mający kształt `(próbki, indeksy_słów)`, a następnie w roli pierwszej warstwy sieci neuronowej zastosować warstwę mogącą przetwarzać tensory wartości całkowitoliczbowych (warstwę `Embedding` — więcej informacji na jej temat znajdziesz w dalszej części tej książki).
- Można zakodować listy tak, aby zamienić je w wektory zer i jedynek. Oznacza to np. zamienienie sekwencji `[3, 5]` na wektor mający 10 000 wymiarów, który będzie wypełniony samymi zerami, a tylko pod indeksami o numerach 3 i 5 znajdą się jedynki. W takiej sytuacji pierwszą warstwą naszej sieci mogłaby być warstwa `Dense`, która potrafi obsłużyć wektory danych zmiennoprzecinkowych.

Skorzystajmy z drugiego rozwiązania i zamieńmy dane na wektory. W celu zachowania przejrzystości kodu zrobimy to ręcznie.

Listing 3.2. Kodowanie sekwencji wartości całkowitoliczbowych do postaci macierzy wartości binarnych

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
```

Kup książkę Poleć książkę

```
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Tworzy macierz wypełnioną zerami o kształcie (len(sequences), dimension).

Pod wybranymi indeksami umieszcza wartość 1.

Zbiór treningowy w postaci wektora.

Zbiór testowy w postaci wektora.

Teraz próbki wyglądają tak:

```
>>> x_train[0]
array([ 0., 1., 1., ..., 0., 0., 0.])
```

Musimy jeszcze wykonać operację zamiany na wektory etykiet próbek:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Teraz dane mogą zostać przetworzone przez sieć neuronową.

3.4.3. Budowa sieci neuronowej

Dane wejściowe są wektorami, a etykiety mają formę wartości skalarnych (jedynek i zer): to najprostsza sytuacja, z jaką można mieć do czynienia. Tego typu problemy najlepiej jest rozwiązywać za pomocą sieci prostego stosu w pełni połączonych warstw (`Dense`) z aktywacjami `relu`: `Dense(16, activation='relu')`.

Argument przekazywany do każdej warstwy `Dense` (16) jest liczbą ukrytych jednostek warstwy. **Jednostka ukryta** jest wymiarem przestrzeni reprezentacji warstwy. W rozdziale 2. pisałem o tym, że każda warstwa `Dense` z aktywacją `relu` implementuje następujący łańcuch operacji tensorowych:

```
output = relu(dot(W, input) + b)
```

Przy 16 ukrytych jednostkach macierz wag `W` będzie miała kształt `(wymiar_wejściowy, 16)`: iloczyn skalarny macierzy `W` będzie rzutował dane wejściowe na 16-wymiarową przestrzeń reprezentacji (następnie dodawany jest wektor wartości progowych `b` i wykonywana jest operacja `relu`). Wymiary przestrzeni reprezentacji danych można rozumieć jako „stopień swobody, jaką dysponuje sieć podczas nauki wewnętrznych reprezentacji danych". Zwiększenie liczby ukrytych jednostek (zwiększenie liczby wymiarów przestrzeni reprezentacji) pozwala sieci na uczenie się bardziej skomplikowanych reprezentacji, ale działanie takiej sieci będzie wymagało większej mocy obliczeniowej i może prowadzić do wytrenowania niechcianych parametrów (prawidłowości, które poprawią wydajność przetwarzania treningowego zbioru danych, ale będą bezużyteczne podczas przetwarzania danych testowych).

Pracując z warstwami `Dense`, należy odpowiedzieć sobie na dwa pytania dotyczące architektury sieci:

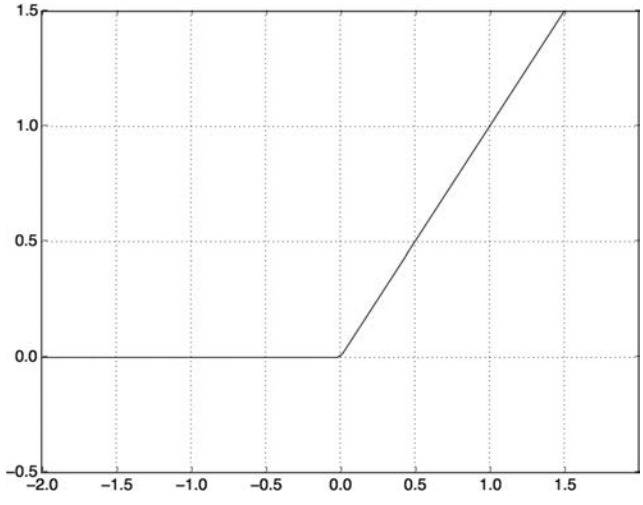
- Ile warstw należy zastosować?
- Ile ukrytych jednostek należy wybrać w każdej z warstw?

W kolejnym rozdziale przedstawię formalne zasady udzielania odpowiedzi na te pytania. Na razie przyjmijmy następujące założenia:

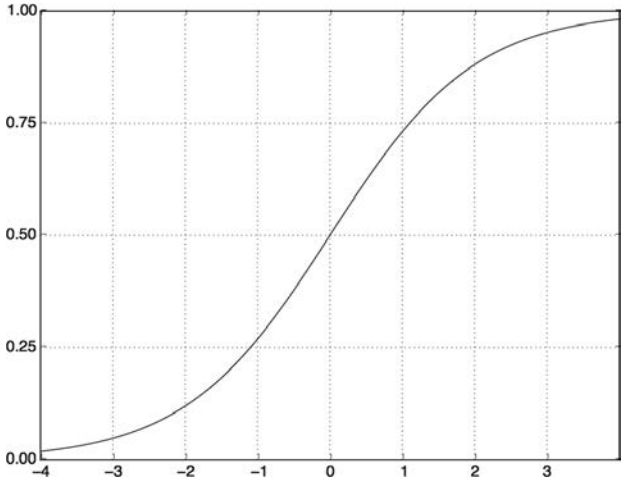
Kup książkę Poleć książkę

- Dwie warstwy pośrednie, zawierające po 16 ukrytych jednostek każda.
- Trzecia warstwa, generująca przewidywanie sentymentu analizowanej recenzji w postaci wartości skalarnej.

Warstwy pośrednie będą korzystały z funkcji aktywacji `relu`, a ostatnia warstwa będzie korzystała z funkcji aktywacji `sigmoid`, co pozwoli na wygenerowanie wartości znajdującej się w zakresie od 0 do 1 określającej prawdopodobieństwo tego, że dana recenzja jest pozytywna. Funkcja `relu` (wyprostowana jednostka liniowa) jest funkcją, która ma wyzerowywać negatywne wartości (patrz rysunek 3.4), a funkcja `sigmoid` „upycha" wartości tak, aby znalazły się w zakresie od 0 do 1 (patrz rysunek 3.5), co pozwala sieci na generowanie wartości, które można interpretować jako prawdopodobieństwo.



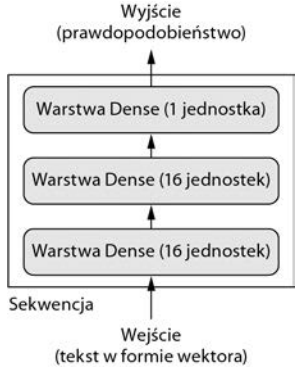
Rysunek 3.4. Funkcja relu



Rysunek 3.5. Funkcja sigmoid

Kup książkę Poleć książkę

Na rysunku 3.6 pokazano schemat sieci. Oto kod implementacji sieci za pomocą pakietu Keras (przypomina on implementację sieci z zaprezentowanego wcześniej przykładu przetwarzania zbioru MNIST).



Rysunek 3.6. Sieć składająca się z trzech warstw

Listing 3.3. Definicja modelu

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Czym są funkcje aktywacji i dlaczego musimy z nich korzystać?

Warstwa `Dense` bez funkcji aktywacji takiej jak funkcja `relu` (zwana również **funkcją nieliniową**) składałaby się z dwóch operacji liniowych — iloczynu skalarnego i dodawania:

```
output = dot(W, input) + b
```

W związku z tym warstwa mogłaby uczyć się tylko **przekształceń liniowych** (przekształceń afinicznych) danych wejściowych. **Przestrzeń hipotez** takiej warstwy byłaby zestawem wszystkich możliwych przekształceń liniowych zbioru danych w przestrzeni o 16 wymiarach. Taka przestrzeń hipotez jest zbyt ograniczona i nie wykorzystalibyśmy tego, że dysponujemy wieloma warstwami reprezentacji (głęboki stos warstw liniowych implementowałby tylko operacje liniowe i dodawanie kolejnych warstw nie rozszerzałoby przestrzeni hipotez).

W celu uzyskania dostępu do o wiele bogatszej przestrzeni hipotez i skorzystania z możliwości oferowanych przez głębokie reprezentacje danych potrzebujemy nieliniowości zapewnianej przez funkcję aktywacji. Funkcją najczęściej stosowaną w uczeniu głębokim tego typu jest funkcja `relu`, ale istnieje wiele innych funkcji o podobnie dziwnych nazwach: `prelu`, `elu` itd.

Na koniec musimy wybrać funkcję straty i optymalizator. Pracujemy nad problemem klasyfikacji binarnej, a sieć zwraca wartości prawdopodobieństwa (na końcu sieci znajduje się warstwa jednej jednostki z funkcją aktywacji `sigmoid`), a więc najlepiej jest skorzystać z funkcji straty `binary_crossentropy` (binarnej entropii krzyżowej). Nie jest to jedyna opcja, z której możemy skorzystać. Możemy również użyć np. funkcji średniego błędu

Kup książkę Poleć książkę

kwadratowego `mean_squared_error`, ale entropia krzyżowa jest zwykle najlepszą opcją w przypadku modeli zwracających wartości prawdopodobieństwa. Termin **entropia krzyżowa** wywodzi się z teorii informacji. Jest to miara odległości między rozkładami prawdopodobieństwa a w tym przypadku rozkładem prawdziwych wartości i rozkładem przewidywanych wartości.

Oto kod konfigurujący model. Wybieramy w nim optymalizator `rmsprop` i funkcję straty `binary_crossentropy`. Zauważ, że podczas trenowania monitorować będziemy również dokładność (`accuracy`).

Listing 3.4. Kompilowanie modelu

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Metryka, optymalizator i funkcja straty są definiowane za pomocą łańcuchów. Jest to możliwe, ponieważ `rmsprop`, `binary_crossentropy` i `accuracy` to pakiety wchodzące w skład biblioteki Keras. Czasami zachodzi konieczność skonfigurowania parametrów optymalizatora lub przekazania samodzielnie wykonanej funkcji straty lub funkcji metryki. Można to zrobić, przekazując instancję klasy optymalizatora jako argument `optimizer` (patrz listing 3.5) i przekazując funkcję obiektów jako argumenty `loss` i `metrics` (patrz listing 3.6).

Listing 3.5. Konfiguracja optymalizatora

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Listing 3.6. Korzystanie z własnych funkcji straty i metryki

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

3.4.4. Walidacja modelu

W celu monitorowania dokładności modelu w czasie trenowania utworzymy zbiór danych, które nie były używane do trenowania modelu. Zrobimy to, odtłaczając 10 000 próbek od treningowego zbioru danych.

Listing 3.7. Tworzenie zbioru walidacyjnego

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Kup książkę Poleć książkę

Teraz będziemy trenować model przez 20 epok (wykonamy 20 iteracji wszystkich próbek znajdujących się w tensorach `x_train` i `y_train`) z podziałem na wsady po 512 próbek. Jednocześnie będziemy monitorować funkcje straty i dokładności modelu przy przetwarzaniu 10 000 próbek, które przed chwilą odłożyliśmy na bok. W tym celu musimy przekazać zbiór walidacyjny (kontrolny) jako argument `validation_data`:

Listing 3.8. Trenowanie modelu:

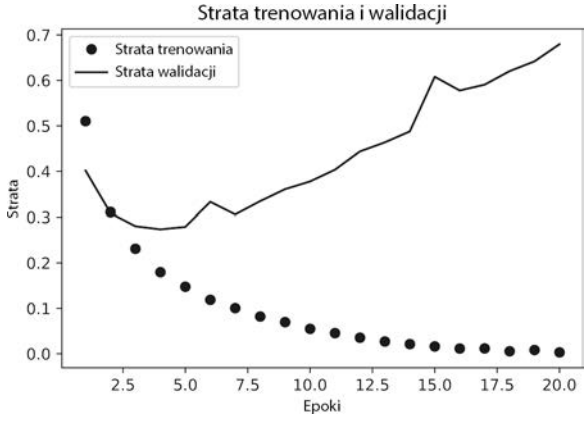
```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

W przypadku trenowania na procesorze CPU przetworzenie jednej epoki procesu zajmuje mniej niż 2 sekundy — cały proces trwa około 20 sekund. Pod koniec każdej epoki algorytm zatrzymuje się na chwilę, ponieważ model oblicza stratę i dokładność, korzystając z 10 000 próbek walidacyjnego zbioru danych.

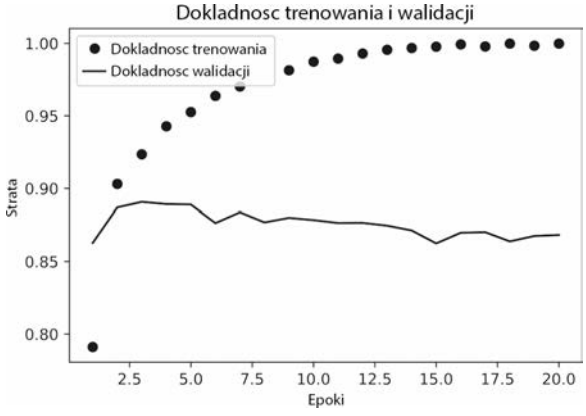
Zwróć uwagę na to, że wywołanie metody `model.fit()` zwraca obiekt `History` (historia). Obiekt ten mieści element o nazwie `history`, który jest słownikiem zawierającym dane dotyczące przebiegu procesu trenowania. Przyjrzyjmy się mu:

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

Słownik zawiera cztery elementy: po jednym związanym z każdą z metryk monitorowanych podczas trenowania i walidacji. Utwórzmy wykres porównujący stratę treningu i walidacji (patrz rysunek 3.7), a także wykres zmian dokładności trenowania i walidacji (patrz rysunek 3.8). Uzyskane przez Ciebie wyniki mogą nieco odbiegać od moich z powodu losowego inicjowania sieci.



Rysunek 3.7. Strata trenowania i walidacji



Rysunek 3.8. Dokładność trenowania i walidacji

Listing 3.9. Tworzenie wykresu strat trenowania i walidacji

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, loss, 'bo', label='Strata trenowania')
plt.plot(epochs, val_loss, 'b', label='Strata walidacji')
plt.title('Strata trenowania i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()

plt.show()
```

Parametr bo definiuje linię przerywaną w postaci niebieskich kropek.

Parametr b definiuje ciągłą niebieską linię.

Listing 3.10 Tworzenie wykresu dokładności trenowania i walidacji

```
plt.clf()
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc']

plt.plot(epochs, acc, 'bo', label='Dokladnosc trenowania')
plt.plot(epochs, val_acc, 'b', label='Dokladnosc walidacji')
plt.title('Dokladnosc trenowania i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()

plt.show()
```

Czyszczenie rysunku.

Jak widać, strata trenowania spada z każdą kolejną epoką, a dokładność trenowania wzrasta. Tego oczekujemy od optymalizacji algorytmem spadku gradientu — wartość, którą staramy się minimalizować, powinna maleć w każdej kolejnej iteracji, ale w czwartej epoce strata walidacji i dokładność walidacji rosną. To właśnie przykład sytuacji, przed którą ostrzegałem wcześniej — model sprawdzający się lepiej na treningowym zbiorze

Kup książkę Poleć książkę

danych wcale nie musi sprawdzać się lepiej podczas przetwarzania nowych danych. W praktyce jest to przykład **nadmiernego dopasowania** — po drugiej epoce model jest zbytnio optymalizowany na treningowym zbiorze danych i uczy się konkretnej reprezentacji treningowego zbioru danych, a nie ogólnej wizji sprawdzającej się również poza treningowym zbiorem danych.

W tym przypadku nadmiernemu dopasowaniu możemy zapobiec, przerywając działanie algorytmu po 3 epokach, ale możemy skorzystać z wielu technik zapobiegających nadmiernemu dopasowaniu modelu, które opiszę w kolejnym rozdziale.

Przeprowadźmy trenowanie nowej sieci od podstaw (zróbmy to przez cztery epoki), a następnie dokonajmy ewaluacji na podstawie testowego zbioru danych.

Listing 3.11. Ponowne uczenie modelu od początku

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

Tym razem uzyskaliśmy następujące wyniki:

```
>>> results
[0.2929924130630493, 0.88327999999999995]
```

To dość naiwne rozwiązanie pozwoliło uzyskać dokładność na poziomie 88%. Dopracowane modele powinny zbliżyć się do 95%.

3.4.5. Używanie wytrenowanej sieci do generowania przewidywań dotyczących nowych danych

Po wytrenowaniu sieci możemy jej użyć w celu zrobienia czegoś praktycznego. Aby wygenerować wartość określającą prawdopodobieństwo tego, że recenzja jest pozytywna, wystarczy skorzystać z metody `predict`:

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.86563045],
       [ 0.99936908],
       ...,
       [ 0.45731062],
       [ 0.0038014 ],
       [ 0.79525089]], dtype=float32)
```

Jak widać, w przypadku nowych próbek sieć jest bardzo pewna swojego werdyktu (generuje wartości zbliżone do 0,99 lub 0,01), ale w przypadku innych generuje o wiele mniej pewne wyniki, takie jak 0,6 lub 0,4.

Kup książkę Poleć książkę

3.4.6. Dalsze eksperymenty

Oto eksperymenty, które pomogą Ci utwierdzić się w przekonaniu, że wybraliśmy całkiem sensowną architekturę, z tym że można ją jeszcze usprawnić:

- Korzystaliśmy z dwóch warstw ukrytych. Spróbuj dodać jedną lub trzy warstwy ukryte i sprawdź, jak wpłynie to na dokładność walidacji i testu.
- Spróbuj użyć warstw z większą lub mniejszą liczbą ukrytych jednostek: wypróbuj warstwy z np. 32 i 64 jednostkami.
- Zamiast funkcji straty `binary_crossentropy` skorzystaj z funkcji straty `mse`.
- Wypróbuj działanie funkcji aktywacji `tanh` (funkcja ta była popularna na początku rozwoju sieci neuronowych) — zastąp nią funkcję `relu`.

3.4.7. Wnioski

Oto wnioski, które należy wynieść z tego przykładu:

- Zwykle dane wymagają przeprowadzenia wstępnej obróbki, po której można skierować je w formie tensorów do wejścia sieci neuronowej. Sekwencja słów może być przedstawiona w formie wektorów wartości binarnych, ale można to zrobić również na inne sposoby.
- Stosy warstw `Dense` z aktywacją `relu` mogą służyć do rozwiązywania różnych problemów (między innymi klasyfikacji tonu wypowiedzi). W związku z tym najprawdopodobniej będziesz często korzystać z nich w przyszłości.
- W przypadku problemu klasyfikacji binarnej (dwie klasy wyjściowe) na końcu sieci powinna znajdować się warstwa `Dense` z jedną jednostką i funkcją aktywacji `sigmoid` — wartości wyjściowe generowane przez sieć powinny być skalarami znajdującymi się w zakresie od 0 do 1 (powinny określać prawdopodobieństwo).
- W takiej konfiguracji warstwy wyjściowej sieci funkcją straty powinna być binarna entropia krzyżowa (`binary_crossentropy`).
- Optymalizator `rmsprop` jest — ogólnie rzecz biorąc — dobrym wyborem do każdego problemu. W związku z tym masz o jedną rzecz mniej do przeanalizowania.
- Sieci neuronowe wraz z coraz lepszym poznawaniem danych treningowych zaczynają się nadmiernie do nich dopasowywać, co prowadzi do pogorszenia rezultatów przetwarzania nowych danych. Musisz stale monitorować wydajność sieci podczas przetwarzania danych niewchodzących w skład zbioru treningowego.

3.5. Przykład klasyfikacji wieloklasowej: klasyfikacja krótkich artykułów prasowych

W poprzednim podrozdziale przedstawiłem klasyfikację wektorów przy podziale na dwie rozłączne klasy za pomocą ściśle połączonej sieci neuronowej. Co się dzieje, gdy mamy więcej klas?

W tym podrozdziale zbudujemy sieć klasyfikującą doniesienia prasowe Agencji Reutera na 46 niezależnych tematów. Podział ma być dokonany na wiele grup, a więc mamy tym razem do czynienia z problemem **klasyfikacji wieloklasowej**. Każdy element zbioru danych może być przypisany tylko do jednej kategorii, a więc problem ten możemy

Kup książkę Poleć książkę

określić mianem **jednoetykietowej klasyfikacji wieloklasowej**. Gdyby element zbioru danych mógł należeć jednocześnie do wielu kategorii (w tym przypadku do wielu tematów), to mielibyśmy do czynienia z problemem **wieloetykietowej klasyfikacji wieloklasowej**.

3.5.1. Zbiór danych Agencji Reutera

W tym podrozdziale będziemy pracować nad zbiorem danych Agencji Reutera — zestawem krótkich informacji prasowych dotyczących określonego tematu, które zostały opublikowane przez tę agencję w 1986 r. Jest to prosty i popularny zbiór danych, doskonale nadający się do eksperymentowania z klasyfikacją tekstu. Zbiór ten zawiera 46 różnych tematów. Do niektórych z nich należy o wiele więcej informacji prasowych niż do innych, ale każdy z tematów ma przynajmniej 10 przykładów w treningowym zbiorze danych.

Zbiór Agencji Reutera, podobnie jak zbiory IMDB i MNIST, wchodzi w skład pakietu Keras. Przyjrzyjmy się jego zawartości.

Listing 3.12. Ładowanie zbioru danych Agencji Reutera

```
from keras.datasets import reuters

(train_data, train_labels), (test_data, test_labels) = reuters.load_data(num_words=10000)
```

Podobnie jak w przypadku zbioru IMDB stosujemy argument `num_words=10000`, który ogranicza nasze działania do 10 000 słów występujących najczęściej w analizowanym zbiorze danych.

Dysponujemy 8982 przykładami treningowymi i 2246 przykładami testowymi:

```
>>> len(train_data)
8982
>>> len(test_data)
2246
```

Każdy przykład jest listą wartości całkowitoliczbowych (indeksów słów) — takie samo rozwiązanie zostało zaprezentowane w przykładzie zbioru IMDB:

```
>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Poniższy kod umożliwia odkodowanie słów (możesz go uruchomić w celu zaspokojenia swojej ciekawości dotyczącej treści).

Listing 3.13. Dekodowanie indeksów

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_newswire=''.join([reverse_word_index.get(i - 3, '?')
↪for i in train_data[0]])
```

Kod dekodujący recenzję. Zauważ, że indeksy są przesunięte o 3, ponieważ pod indeksami o numerach 0, 1 i 2 znajdują się indeksy symbolizujące „wypełnienie", „początek sekwencji" i „nieznane słowo".

Kup książkę Poleć książkę

Tabela etykiet przykładów zawiera wartości całkowitoliczbowe znajdujące się w zakresie od 0 do 45 (są to indeksy tematów):

```
>>> train_labels[10]
3
```

3.5.2. Przygotowywanie danych

Dane możemy zamienić na wektory za pomocą tego samego kodu, z którego korzystaliśmy w poprzednim przykładzie.

Listing 3.14. Konwersja danych

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data) <-- Zbiór treningowy w postaci wektora.
x_test = vectorize_sequences(test_data) <-- Zbiór testowy w postaci wektora.
```

Operację przekształcania etykiet na wektor można przeprowadzić na dwa sposoby: poprzez rzutowanie listy etykiet na tensor wartości całkowitoliczbowych lub poprzez kodowanie z gorącą jedynką. Kodowanie z gorącą jedynką jest używane zwykle w przypadku danych kategorialnych — często określa się je mianem **kodowania kategorialnego**. Szczegółowe wyjaśnienie tego algorytmu kodowania znajdziesz w podrozdziale 6.1. W tym przypadku kodowanie z gorącą jedynką przeprowadzone na zbiorze etykiet będzie polegało na osadzeniu każdej etykiety w formie wektora wypełnionego samymi zerami z liczbą 1 umieszczoną w miejscu indeksu etykiety. Przyjrzyj się kodowi implementującemu ten sposób kodowania:

```
def to_one_hot(labels, dimension=46):
    results = np.zeros((len(labels), dimension))
    for i, label in enumerate(labels):
        results[i, label] = 1.
    return results

one_hot_train_labels = to_one_hot(train_labels) <-- Etykiety zbioru treningowego w postaci wektora.
one_hot_test_labels = to_one_hot(test_labels) <-- Etykiety zbioru testowego w postaci wektora.
```

Operację tę można wykonać za pomocą kodu wbudowanego w pakiet Keras (jego działanie widzieliśmy już podczas pracy nad zbiorem MNIST):

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Kup książkę Poleć książkę

3.5.3. Budowanie sieci

Problem klasyfikacji tematów przypomina nieco problem klasyfikacji recenzji filmów: w obu sytuacjach próbujemy dokonać klasyfikacji krótkiego fragmentu tekstu. Tym razem mamy dodatkowe ograniczenie: liczba klas wyjściowych wzrosła z 2 do 46. W związku z tym przestrzeń wyjściowa ma o wiele więcej wymiarów.

W przypadku stosowanego wcześniej stosu warstw `Dense` każda warstwa ma dostęp tylko do informacji wygenerowanych przez poprzednią warstwę. Jeżeli jakaś informacja ważna z punktu widzenia klasyfikacji zostanie pominięta przez którąś z warstw, to nie ma możliwości jej odzyskania przez kolejne warstwy. Każda warstwa może potencjalnie stać się informacyjnym wąskim gardłem. W poprzednim przykładzie stosowaliśmy warstwy pośrednie o 16 wymiarach, ale przestrzeń 16 wymiarów może być zbyt ograniczona, aby nauczyć się rozdzielania danych na 46 różnych klas. Tak małe warstwy mogą stać się wąskimi gardłami, przez które nie będą przechodziły dalej ważne informacje.

W związku z tym będziemy tworzyli większe warstwy. Zacznijmy od 64 jednostek.

Listing 3.15. Definicja modelu

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Warto zwrócić uwagę na jeszcze dwie rzeczy związane z tą architekturą:

- Sieć jest zakończona warstwą `Dense` o rozmiarze równym 46. W związku z tym każda próbka danych wejściowych spowoduje wygenerowanie przez sieć wektora o 46 wymiarach. Każdy element tego wektora (każdy jego wymiar) będzie zawierał informację odwołującą się do innej klasy.
- W ostatniej warstwie zastosowano funkcję aktywacji `softmax`. Rozwiązanie to widzieliśmy w przykładzie zbioru MNIST. Dzięki niemu sieć będzie zwracała **rozkład prawdopodobieństwa** 46 różnych klas — dla każdej próbki wejściowej sieć wygeneruje 46-wymiarowy wektor wyjściowy, w którym element `output[i]` jest prawdopodobieństwem tego, że próbka należy do klasy `i`. Wszystkie 46 wyników po zsumowaniu da wartość 1.

W tej sytuacji najlepiej jest skorzystać z funkcji straty w postaci kategorialnej entropii krzyżowej (`categorical_crossentropy`). Funkcja ta mierzy odległość między dwoma rozkładami prawdopodobieństw: rozkładem prawdopodobieństw zwróconych przez sieć i rozkładem prawdopodobieństw określanym przez etykiety. Minimalizacja odległości między tymi rozkładami pozwala na trenowanie sieci w celu zwracania wartości jak najbardziej zbliżonych do prawdziwych etykiet.

Kup książkę Poleć książkę

Listing 3.16. Kompilacja modelu

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

3.5.4. Walidacja modelu

Odłączmy 1000 próbek od treningowego zbioru danych w celu użycia ich w charakterze zbioru kontrolnego.

Listing 3.17. Tworzenie zbioru kontrolnego

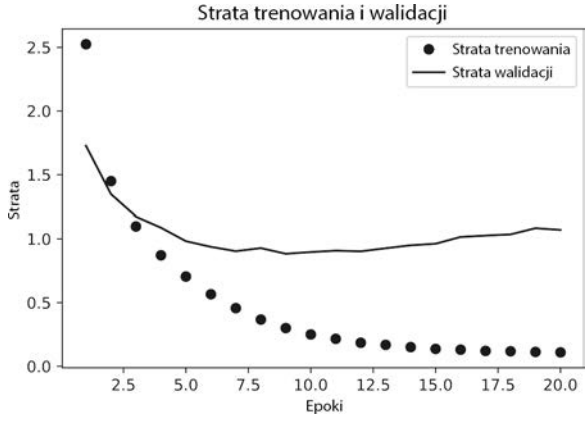
```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

Czas uruchomić trenowanie sieci trwające 20 epok.

Listing 3.18. Trenowanie modelu

```
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Teraz możemy wyświetlić wykresy krzywych strat i dokładności (patrz rysunki 3.9 i 3.10).

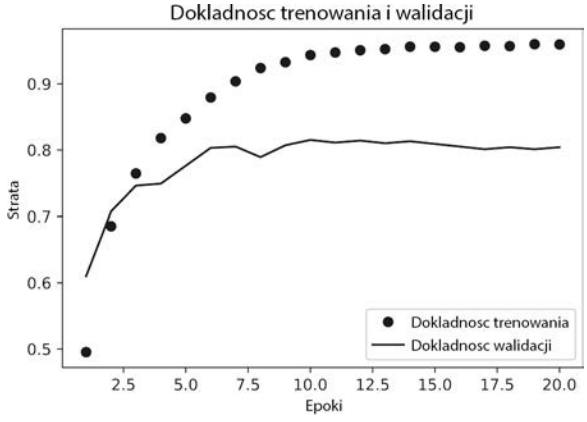


Rysunek 3.9. Strata trenowania i walidacji

Listing 3.19. Tworzenie wykresu strat trenowania i walidacji

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(loss) + 1)
```

Kup książkę Poleć książkę



Rysunek 3.10. Dokładność trenowania i walidacji

```
plt.plot(epochs, loss, 'bo', label='Strata trenowania')
plt.plot(epochs, val_loss, 'b', label='Strata walidacji')
plt.title('Strata trenowania i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()

plt.show()
```

Listing 3.20. Tworzenie wykresu dokładności trenowania i walidacji

```
plt.clf() <------ Czyszczenie rysunku.

acc = history.history['acc']
val_acc = history.history['val_acc']

plt.plot(epochs, acc, 'bo', label='Dokladnosc trenowania')
plt.plot(epochs, val_acc, 'b', label='Dokladnosc walidacji')
plt.title('Dokladnosc trenowania i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()

plt.show()
```

Po dziewięciu epokach sieć zaczyna ulegać przeuczeniu. Spróbujmy uruchomić jeszcze raz proces uczenia sieci, ale tym razem ograniczymy jego działanie do 9 epok. Następnie sprawdzimy działanie sieci na testowym zbiorze danych.

Listing 3.21. Ponowne trenowanie modelu od podstaw

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
```

Kup książkę Poleć książkę

```
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=8,
          batch_size=512,
          validation_data=(x_val, y_val))
results = model.evaluate(x_test, one_hot_test_labels)
```

Oto uzyskane przez nas wyniki:

```
>>> results
[0.9565213431445807, 0.79697239536954589]
```

Uzyskaliśmy dokładność na poziomie zbliżonym do 80%. W wyważonej klasyfikacji binarnej losowy klasyfikator uzyskałby wynik 50%, ale w tym przypadku jego wynik byłby zbliżony do 19%. Dokładność naszego klasyfikatora wydaje się całkiem dobra, jeżeli porówna się ją z klasyfikatorem przyporządkowującym etykiety w sposób losowy:

```
>>> import copy
>>> test_labels_copy = copy.copy(test_labels)
>>> np.random.shuffle(test_labels_copy)
>>> hits_array = np.array(test_labels) == np.array(test_labels_copy)
>>> float(np.sum(hits_array)) / len(test_labels)
0.18655387355298308
```

3.5.5. Generowanie przewidywań dotyczących nowych danych

Możemy zweryfikować zwracanie przez metodę `predict` naszej instancji modelu rozkładu prawdopodobieństwa wszystkich 46 tematów. Wygenerujmy przewidywania dla wszystkich elementów testowego zbioru danych.

Listing 3.22. Generowanie przewidywań nowych danych

```
predictions = model.predict(x_test)
```

Każdy element zmiennej `predictions` jest wektorem o długości równej 46:

```
>>> predictions[0].shape
(46,)
```

Suma wszystkich wartości tego wektora wynosi 1:

```
>>> np.sum(predictions[0])
1.0
```

Najwyższa wartość wektora wskazuje przewidywaną klasę — klasę, do której najprawdopodobniej należy dana próbka:

```
>>> np.argmax(predictions[0])
4
```

3.5.6. Inne sposoby obsługi etykiet i funkcji straty

Wcześniej pisałem o tym, że drugim sposobem kodowania etykiet jest rzutowanie ich jako tensory całkowitoliczbowe w następujący sposób:

Kup książkę Poleć książkę

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)
```

Skorzystanie z tego rozwiązania wymaga wprowadzenia jednej zmiany w modelu — wybrania innej funkcji straty. W listingu 3.21 wybraliśmy funkcję `categorical_crossentropy`, która wymaga podawania etykiet zakodowanych kategorialnie. Przy etykietach mających postać liczb całkowitych powinniśmy użyć funkcji `sparse_categorical_crossentropy`:

```
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Nowa funkcja straty z punktu widzenia matematyki działa tak samo jak funkcja `categorical_crossentropy`, ale wyposażono ją w inny interfejs.

3.5.7. Dlaczego warto tworzyć odpowiednio duże warstwy pośrednie?

Stwierdziłem wcześniej, że sieć generuje dane wyjściowe mające 46 wymiarów, a więc powinniśmy unikać warstw pośrednich mających mniej niż 46 ukrytych jednostek. Sprawdźmy, co się stanie, jeżeli wprowadzimy do sieci takie informacyjne wąskie gardło. Wprowadźmy do sieci warstwy pośrednie mające mniej niż 46 wymiarów: zacznijmy od przykładowej warstwy z 4 wymiarami.

Listing 3.23. Model z wąskim gardłem nieprzepuszczającym wystarczająco dużo informacji

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=128,
          validation_data=(x_val, y_val))
```

Dokładność sieci w procesie walidacji osiąga teraz wartość szczytową na poziomie około 71%, a więc dokładność spadła w skali bezwzględnej o 8%. Wynika to głównie z tego, że sieć stara się skompresować wiele informacji w zbyt małej liczbie wymiarów warstwy pośredniej. Co prawda sieć jest w stanie dokonać podziału na 46 klas, a więc może ona zakodować większość niezbędnych informacji w formie ośmiowymiarowych reprezentacji, ale przestrzeń taka jest zbyt mała, aby umieścić w niej wszystkie informacje.

Kup książkę Poleć książkę

3.5.8. Dalsze eksperymenty

- Spróbuj tworzyć większe lub mniejsze warstwy: warstwy zawierające np. po 32 lub 128 jednostek.
- Korzystaliśmy z dwóch ukrytych warstw. Spróbuj utworzyć sieć zawierającą jedną taką warstwę, a następnie sieć składającą się z trzech takich warstw.

3.5.9. Wnioski

Oto wnioski, które należy wynieść z tego przykładu:

- Podczas klasyfikacji danych na *N* klas sieć powinna kończyć się warstwą `Dense` o rozmiarze równym *N*.
- W razie wystąpienia problemu wieloklasowej klasyfikacji elementów opisywanych przy użyciu jednej etykiety sieć powinna kończyć się funkcją aktywacji `softmax`, która umożliwi wygenerowanie rozkładu wartości prawdopodobieństw *N* klas.
- Kategorialna entropia krzyżowa jest prawie zawsze funkcją straty — powinna być używana podczas pracy z tego typu problemami. Minimalizuje ona odległość między rozkładem prawdopodobieństwa wygenerowanym przez sieć a rozkładem wartości docelowych.
- Podczas klasyfikacji wieloklasowej można korzystać z jednego z dwóch sposobów obsługi etykiet:
  - Kodowania etykiet za pomocą kodowania kategorycznego (kodowania z gorącą jedynką) i używania funkcji `categorical_crossentropy` jako funkcji straty.
  - Kodowania etykiet za pomocą wartości całkowitoliczbowych i używania funkcji `sparse_categorical_crossentropy` jako funkcji straty.
- Jeżeli musisz podzielić dane na dużą liczbę kategorii, to staraj się unikać tworzenia wąskich gardeł blokujących przepływ informacji w postaci zbyt małych warstw pośrednich.

3.6. Przykład regresji: przewidywanie cen mieszkań

Dwa poprzednie przykłady były problemami klasyfikacji, w których celem było przewidzenie etykiety opisującej dane wejściowe. Innym spotykanym często problemem uczenia maszynowego jest **regresja**, która polega na przewidywaniu wartości o charakterze ciągłym, a nie dyskretnej etykiety. Przykładem regresji jest przewidywanie jutrzejszej temperatury na podstawie zgromadzonych danych meteorologicznych lub przewidywanie czasu potrzebnego na skończenie oprogramowania na podstawie jego specyfikacji.

UWAGA Nie myl **regresji** z algorytmem **regresji logistycznej**. Regresja logistyczna wbrew swej nazwie nie jest algorytmem regresji. Jest algorytmem klasyfikacji.

3.6.1. Zbiór cen mieszkań w Bostonie

Będziemy starali się przewidzieć medianę cen mieszkań w podmiejskich dzielnicach Bostonu w połowie lat 70. XX w. na podstawie danych określających konkretną dzielnicę, takich jak współczynnik przestępczości czy lokalny podatek od nieruchomości. Tym razem będziemy korzystali ze zbioru danych, który różni się od wcześniejszych zbiorów

Kup książkę Poleć książkę

dwiema rzeczami. Ma dość mało elementów: tylko 506 (podzielono je na zbiór treningowy zawierający 404 próbki i zbiór testowy zawierający 102 próbki). Każda **cecha** danych wejściowych (przykładem cechy jest współczynnik przestępczości) jest wyrażona w innej skali. Niektóre wartości są ułamkami przyjmującymi wartości od 0 do 1, inne przyjmują wartości z zakresu od 1 do 12, a jeszcze inne — od 0 do 100 itd.

Listing 3.24. Ładowanie zbioru danych cen mieszkań w Bostonie

```
from keras.datasets import boston_housing

(train_data, train_targets), (test_data, test_targets) =  boston_housing.load_data()
```

Przyjrzyjmy się danym:

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
```

Jak widać, zbiór treningowy składa się z 404 próbek, a zbiór testowy ze 102 próbek. Każda próbka jest opisana za pomocą 13 cech numerycznych, takich jak współczynnik przestępczości, średnia liczba pokoi w mieszkaniu czy dostęp do dróg szybkiego ruchu.

Celem jest określenie median wartości domów zamieszkanych przez właścicieli — wyrażonych w tysiącach dolarów:

```
>>> train_targets
[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

Ceny zwykle wahają się od 10 000 dolarów do 50 000 dolarów. Tanio? Pamiętaj o tym, że to wartości z połowy lat 70. Ceny w tym zbiorze nie uwzględniają inflacji.

3.6.2. Przygotowywanie danych

Ładowanie do sieci neuronowej wartości należących do kilku różnych zakresów może sprawić problem. Niektóre sieci są w stanie automatycznie dopasować do siebie tak różne dane, ale z pewnością utrudni to proces uczenia. Najlepszą praktyką podczas pracy z takimi danymi jest przeprowadzenie normalizacji poszczególnych cech: w przypadku każdej cechy danych wejściowych (kolumny macierzy danych wejściowych) należy przeprowadzić operację odejmowania od wartości średniej i dzielenia przez odchylenie standardowe — wówczas wartości cech zostaną wyśrodkowane wokół zera i będą charakteryzowały się jednostkowym odchyleniem standardowym. Operację tę można z łatwością przeprowadzić dzięki bibliotece Numpy.

Listing 3.25. Normalizowanie danych

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std

test_data -= mean
test_data /= std
```

Kup książkę Poleć książkę

Zwróć uwagę na to, że wielkości używane podczas normalizacji testowego zbioru danych są obliczane na podstawie treningowego zbioru danych. Nigdy nie powinniśmy korzystać z wartości obliczonych na podstawie treningowego zbioru danych. Dotyczy to nawet tak prostych zadań jak normalizacja danych.

3.6.3. Budowanie sieci

Dysponujemy małą liczbą próbek, a więc zbudujemy bardzo małą sieć zawierającą dwie warstwy ukryte, składające się z 64 jednostek każda. Ogólnie rzecz biorąc, im mniejszą ilością danych treningowych dysponujemy, tym bardziej jesteśmy narażeni na nadmierne dopasowanie sieci. W celu zminimalizowania efektu nadmiernego dopasowania można między innymi korzystać z małej sieci.

Listing 3.26. Definicja modelu

```
from keras import models
from keras import layers

def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Będziemy tworzyć wiele instancji tego samego modelu, a więc konstruując je, będziemy korzystać z funkcji.

Sieć kończy się pojedynczą jednostką bez funkcji aktywacji (jest to warstwa liniowa). To typowe rozwiązanie stosowane w regresji skalarnej (regresji, w której próbuje się przewidzieć jedną wartość o charakterze ciągłym). Zastosowanie funkcji aktywacji ograniczyłoby zakres wartości wyjściowych możliwych do wygenerowania. Gdybyśmy zastosowali w ostatniej warstwie tej sieci funkcję aktywacji `sigmoid`, to sieć mogłaby generować tylko wartości znajdujące się w zakresie od 0 do 1. W praktyce zastosowaliśmy ostatnią warstwę o charakterze liniowym, a więc możemy przewidywać dowolne wartości.

Zwróć uwagę na to, że sieć jest kompilowana z funkcją straty `mse` (średniego błędu kwadratowego). Funkcja ta oblicza kwadrat różnicy między wartościami przewidywanymi przez sieć i wartościami docelowymi. Ta funkcja straty jest często używana w czasie rozwiązywania problemów regresji.

Podczas trenowania monitorowana jest nowa metryka: **średni błąd bezwzględny** (`mae`). Jest to bezwzględna wartość różnicy między wartościami przewidywanymi przez sieć a wartościami docelowymi. Średnia wartość błędu bezwzględnego o wartości równej np. 0,5 w przypadku tego problemu oznacza, że przewidywane ceny średnio odbiegają od wartości docelowych o 500 dolarów.

3.6.4. K-składowa walidacja krzyżowa

W celu oceny sprawności działania sieci podczas dostrajania jej parametrów, takich jak liczba epok trenowania, możemy tak jak wcześniej podzielić dane treningowe na podzbiór treningowy i podzbiór walidacyjny, ale nasz zbiór jest na tyle mały, że podzbiór

Kup książkę Poleć książkę

walidacyjny utworzony w ten sposób byłby bardzo mały (zawierałby np. tylko 100 elementów). W związku z tym wynik walidacji mógłby ulegać dużej zmianie w zależności od tego, które elementy treningowego zbioru danych byłyby używane podczas walidacji, a które podczas trenowania. Wyniki walidacji mogłyby charakteryzować się dużą **wariancją** zależną od podziału zbioru testowego na podzbiór testowy i walidacyjny. W takiej sytuacji nie można przeprowadzić wiarygodnej walidacji.

W takim przypadku najlepiej jest skorzystać z **walidacji krzyżowej *k*-składowych** (patrz rysunek 3.11). Polega ona na podziale dostępnych danych na k części (zwykle 4 lub 5), utworzeniu k identycznych modeli i trenowaniu każdego z nich na $k - 1$ częściach zbioru oraz przeprowadzaniu ewaluacji na pozostałej, nieużytej wcześniej części zbioru dostępnych danych. Wynik walidacji modelu jest średnią wyników walidacji wszystkich składowych modeli. Kod implementujący to rozwiązanie jest dość prosty.

Rysunek 3.11. Metoda 3-składowej walidacji krzyżowej

Listing 3.27. Algorytm walidacji k-składowej

```
import numpy as np

k = 4
num_val_samples = len(train_data) // k
num_epochs = 100
all_scores = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
```

Przygotuj dane walidacyjne: dane z k-tej składowej.

Przygotuj dane treningowe: dane z pozostałych składowych.

Zbuduj model Keras (model został skompilowany wcześniej).

Trenuj model w trybie cichym (parametr verbose = 0).

Kup książkę Poleć książkę

```
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

Przeprowadź ewaluację modelu przy użyciu danych walidacyjnych.

Przy parametrze `num_epochs = 100` uzyskamy następujące rezultaty:

```
>>> all_scores
[2.588258957792037, 3.1289568449719116, 3.1856116051248984, 3.0763342615401386]
>>> np.mean(all_scores)
2.9947904173572462
```

Podczas poszczególnych iteracji uzyskujemy dość zróżnicowane wartości walidacji (od 2,6 do 3,2). Średnia wartość (3,0) jest wartością, na której można o wiele bardziej polegać niż na poszczególnych wynikach walidacji składowych — właśnie to chcieliśmy uzyskać, stosując *k*-składową walidację krzyżową. W tym przypadku odchodzimy od wartości docelowych średnio o 3000 dolarów, co jest znaczącą kwotą przy cenach znajdujących się w zakresie od 10 000 do 50 000 dolarów.

Spróbujmy wydłużyć proces trenowania do 500 epok. W celu obserwacji wydajności modelu w każdej epoce zmodyfikujemy pętlę treningową tak, aby zapisywała wynik walidacji poszczególnych epok w dzienniku pracy.

Listing 3.28. Zapisywanie wyników walidacji każdej składowej

```
num_epochs = 500
all_mae_histories = []
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate(
        [train_data[:i * num_val_samples],
         train_data[(i + 1) * num_val_samples:]],
        axis=0)
    partial_train_targets = np.concatenate(
        [train_targets[:i * num_val_samples],
         train_targets[(i + 1) * num_val_samples:]],
        axis=0)

    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
                        validation_data=(val_data, val_targets),
                        epochs=num_epochs, batch_size=1, verbose=0)
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

Przygotowuje dane walidacyjne: dane z k-tej składowej.

Przygotowuje dane treningowe: dane z pozostałych składowych.

Buduje model Keras (model został skompilowany wcześniej).

Przeprowadza ewaluację modelu przy użyciu danych walidacyjnych.

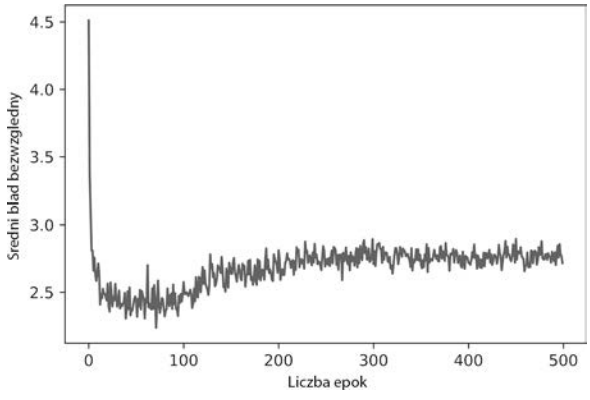
Teraz możemy obliczyć średni wynik walidacji wszystkich składowych poszczególnych epok.

Listing 3.29. Obliczanie wyników walidacji kolejnych epok procesu uczenia

```
average_mae_history = [
    np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

Kup książkę Poleć książkę

Przedstawmy średni błąd bezwzględny na wykresie (patrz rysunek 3.12).



Rysunek 3.12. Średni błąd bezwzględny w poszczególnych epokach

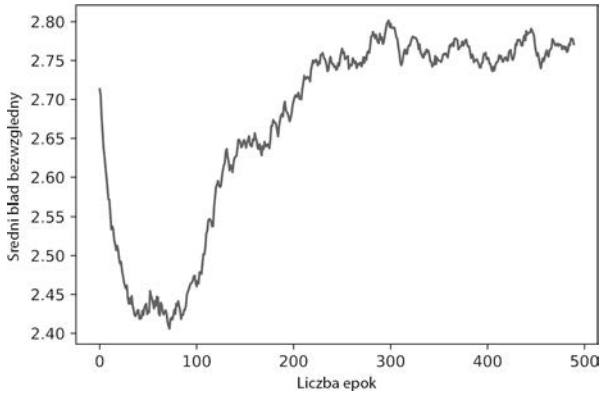
Listing 3.30. Przedstawianie wyników walidacji na wykresie

```
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1), average_mae_history)
plt.xlabel('Liczba epok')
plt.ylabel('Sredni blad bezwzgledny')
plt.show()
```

Wykres może okazać się dość trudny do przeanalizowania z powodu skali i dużej wariancji. Przeprowadźmy następujące operacje:

- Pomińmy 10 pierwszych punktów wykresu (wymagają one zastosowania innej skali niż reszta krzywej).
- Zastąpmy każdy punkt wykresu ruchomą średnią wykładniczą poprzednich punktów (spowoduje to wyrównanie przebiegu krzywej).

Po zastosowaniu tych modyfikacji uzyskamy wykres przedstawiony na rysunku 3.13.



Rysunek 3.13. Średni błąd bezwzględny w poszczególnych epokach (bez 10 pierwszych obserwacji)

Kup książkę Poleć książkę

Listing 3.31. Tworzenie wykresu wyników walidacji pomijającego 10 pierwszych epok

```
def smooth_curve(points, factor=0.9):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Liczba epok')
plt.ylabel('Sredni blad bezwzgledny')
plt.show()
```

Z ostatniego wykresu wynika, że średni błąd bezwzględny przestaje ulegać poprawie po 80 epokach. Po przekroczeniu tego punktu model zaczyna ulegać przeuczeniu.

Po zakończeniu dostrajania pozostałych parametrów modelu (poza liczbą epok możemy zmienić również rozmiar warstw ukrytych) przeprowadzamy trenowanie ostatecznej wersji modelu na całym zbiorze danych treningowych (w procesie tym korzystamy z optymalnych parametrów), a następnie sprawdzamy jego wydajność na zbiorze testowym.

Listing 3.32. Trenowanie ostatniej wersji modelu

```
model = build_model()                    <-- Utwórz nową, skompilowaną wersję modelu.
model.fit(train_data, train_targets,     <-- Trenuj model na całym zbiorze
          epochs=80, batch_size=16, verbose=0)   danych treningowych.

test_mse_score, test_mae_score = model.evaluate(test_data, test_targets)
```

Oto wynik uzyskany przez finalny model:

```
>>> test_mae_score
2.5532484335057877
```

Przewidywane ceny wciąż odbiegają od rzeczywistych średnio o 2550 dolarów.

3.6.5. Wnioski

Oto wnioski, które należy wynieść z tego przykładu:

- Regresję przeprowadza się przy użyciu innej funkcji straty od tej, z której korzystaliśmy podczas klasyfikacji. W czasie rozwiązywania problemów dotyczących regresji często stosuje się funkcję straty `mse` (średniego błędu kwadratowego).
- Podczas pracy nad problemem regresji stosuje się również inne metryki ewaluacyjne (nie używa się tych samych metryk, które są stosowane w problemach klasyfikacji). Oczywiście wynika to z tego, że koncepcja dokładności nie odnosi się do regresji. Metryką stosowaną w regresji jest średni błąd bezwzględny realizowany przez funkcję `mae`.

Kup książkę Poleć książkę

- Jeżeli cechy danych wejściowych przyjmują wartości o różnych zakresach, to każda cecha powinna zostać niezależnie przeskalowana podczas wstępnej obróbki danych.
- W pracy z małym zbiorem danych ewaluację modelu warto jest oceniać przy użyciu techniki *k*-składowej walidacji krzyżowej.
- W pracy z małym zbiorem danych lepiej jest korzystać z małej sieci z kilkoma warstwami ukrytymi (zwykle stosuje się od 1 do 2 warstw), co pozwoli uniknąć przeuczenia modelu.

Kup książkę Poleć książkę

Podsumowanie rozdziału

- Potrafisz już pracować nad typowymi problemami związanymi z danymi wejściowymi w postaci wektora: klasyfikacją binarną, klasyfikacją wieloklasową z pojedynczymi etykietami i regresją skalarną. W sekcjach „Wnioski” znajdziesz najważniejsze rzeczy, których powinieneś się nauczyć w celu wykonywania tego typu zadań.
- Dane przed przekazaniem do sieci neuronowej zwykle należy przygotować.
- Jeżeli cechy danych wejściowych przyjmują wartości o różnych zakresach, to każda cecha powinna zostać niezależnie przeskalowana podczas wstępnej obróbki danych.
- Sieć neuronowa w pewnym momencie procesu trenowania zaczyna ulegać nadmiernemu dopasowaniu do treningowego zbioru danych, co powoduje pogorszenie rezultatu przetwarzania danych spoza tego zbioru.
- W pracy z małym zbiorem danych lepiej jest korzystać z małej sieci z kilkoma ukrytymi warstwami (zwykle stosuje się od 1 do 2 warstw), co pozwoli uniknąć przeuczenia modelu.
- Jeżeli dane są podzielone na wiele kategorii, to musisz uważać na to, aby nie tworzyć zbyt małych warstw pośrednich, ponieważ będą one blokowały przepływ informacji przez sieć neuronową.
- Regresję przeprowadza się przy użyciu innej funkcji straty od tej, z której korzystaliśmy podczas klasyfikacji.
- W pracy z małym zbiorem danych ewaluację modelu warto jest oceniać przy użyciu techniki *k*-składowej walidacji krzyżowej.

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

C

D

E

F

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

R

S

Kup książkę Poleć książkę

T

U

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę


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
GRUPA
Helion

W skrócie uczenie maszynowe polega na wyodrębnianiu informacji z surowych danych i budowie modelu, który służy do przetwarzania kolejnych surowych danych. Technologia ta od kilku lat intensywnie się rozwija, a w miarę wzrostu jej możliwości rosną również zainteresowanie i oczekiwania architektów i użytkowników. Niektórzy widzą w głębokim uczeniu poważne zagrożenie, jednak obietnice, jakie daje ten rodzaj sztucznej inteligencji, są fascynujące. Narzędzia służące do programowania uczenia maszynowego, takie jak zaimplementowana w Pythonie biblioteka Keras, są dostępne dla każdego, kto chce wykorzystać tę technologię do własnych celów.

Niniejsza książka jest praktycznym przewodnikiem po uczeniu głębokim. Znalazły się tu dokładne informacje o istocie uczenia głębokiego, o jego zastosowaniach i ograniczeniach. Wyjaśniono zasady rozwiązywania typowych problemów uczenia maszynowego. Pokazano, jak korzystać z pakietu Keras przy implementacji rozpoznawania obrazu, przetwarzania języka naturalnego, klasyfikacji obrazów, przewidywania danych szeregu czasowego, analizy sentymentu, generowania tekstu i obrazu. Nawet dość skomplikowane zagadnienia, włączając w to koncepcje i dobre praktyki, zostały wyjaśnione w sposób bardzo przystępny i zrozumiały, tak aby umożliwić samodzielne stosowanie technik uczenia głębokiego w kolejnych projektach.

W tej książce między innymi:

- **kontekst i ogólne koncepcje sztucznej inteligencji, uczenia maszynowego i uczenia głębokiego**
- **sieci neuronowe i pakiet Keras**
- **typowe sposoby pracy nad projektami uczenia głębokiego**
- **rozbudowane modele uczenia głębokiego oraz modele generatywne**
- **perspektywy i ograniczenia technologii**

François Chollet jest znany przede wszystkim jako autor biblioteki uczenia głębokiego Keras. Obecnie pracuje w firmie Google w Mountain View w Kalifornii. Jest niekwestionowanym autorytetem w takich dziedzinach jak uczenie maszynowe i rozwój sztucznej inteligencji. Zajmuje się również rozwojem technik uczenia głębokiego związanych z przetwarzaniem obrazu oraz procesami logicznego myślenia. Zabierał głos na najważniejszych konferencjach branżowych.

Uczenie głębokie. Nikt nie zna granic tej technologii!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!

SZKOLENIA

AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-283-4778-6

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł