O'REILLY

TinyML

Wykorzystanie TensorFlow Lite do uczenia maszynowego na Arduino i innych mikrokontrolerach

Pete Warden
Daniel Situnayake

Tytuł oryginału: TinyML: Machine Learning with TensorFlow Lite on Arduino
and Ultra-Low-Power Microcontrollers

Tłumaczenie: Anna Mizerska

ISBN: 978-83-283-8362-3

© 2022 Helion S.A.

Authorized Polish translation of the English edition TinyML ISBN 9781492052043 © 2020 Pete Warden and Daniel Situnayake.

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

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

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

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

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 11

1. Wprowadzenie 15
Urządzenia z systemem wbudowanym 17
Ciągły rozwój 17

2. Informacje wstępne 19
Do kogo skierowana jest ta książka? 19
Jaki sprzęt będzie Ci potrzebny? 20
Jakie oprogramowanie będzie Ci potrzebne? 21
Czego nauczysz się dzięki tej książce? 22

3. Wprowadzenie do uczenia maszynowego 23
Czym właściwie jest uczenie maszynowe? 24
Proces uczenia głębokiego 25
Określenie celu 25
Zebranie zestawu danych 26
Zaprojektowanie architektury modelu 28
Trenowanie modelu 32
Przekształcenie modelu 37
Uruchomienie procesu wnioskowania 37
Ocena i rozwiązanie ewentualnych problemów 38
Podsumowanie 39

4. „Witaj, świecie” TinyML: budowa i trenowanie modelu 40
Co będziemy budować? 41
Nasz zestaw narzędzi do uczenia maszynowego 43
Python i Jupyter Notebooks 43
Google Colaboratory 43
TensorFlow i Keras 44

Kup ksi k Pole ksi k

Budowa naszego modelu 44
Importowanie pakietów 46
Generowanie danych 48
Rozdzielanie danych 50
Definiowanie podstawowego modelu 51
Trenowanie naszego modelu 55
Wskaźniki treningu 57
Wykres historii 58
Ulepszenie naszego modelu 61
Test 65
Konwertowanie modelu na potrzeby TensorFlow Lite 67
Konwertowanie na plik C 71
Podsumowanie 71

5. „Witaj, świecie” TinyML: budowanie aplikacji 72
Omówienie testów 73
Dodawanie zależności 74
Przygotowanie testów 75
Przygotowanie do rejestrowania danych 75
Mapowanie naszego modelu 76
Klasa AllOpsResolver 78
Alokacja pamięci dla modelu 79
Tworzenie interpretera 79
Sprawdzenie tensora wejścia 80
Uruchamianie procesu wnioskowania 82
Odczytywanie danych wyjściowych 84
Uruchamianie testów 86
Budowa pliku z projektem 89
Omówienie kodu źródłowego 90
Początek pliku main_functions.cc 90
Obsługa wyjścia za pomocą output_handler.cc 94
Koniec pliku main_functions.cc 94
Omówienie pliku main.cc 95
Uruchomienie aplikacji 95
Podsumowanie 96

6. „Witaj, świecie” TinyML: uruchomienie aplikacji na mikrokontrolerze 97
Czym właściwie jest mikrokontroler? 97
Arduino 99
Obsługa wyjścia na Arduino 99
Uruchomienie przykładu 102
Wprowadzanie własnych zmian 106

Kup ksi k Pole ksi k

SparkFun Edge 107
Obsługa wyjścia na SparkFun Edge 107
Uruchomienie przykładu 110
Testowanie programu 117
Sprawdzanie danych o przebiegu programu 117
Wprowadzanie własnych zmian 117
Zestaw ST Microelectronics STM32F746G Discovery 118
Obsługa wyjścia na STM32F746G 118
Uruchomienie przykładu 122
Wprowadzanie własnych zmian 124
Podsumowanie 124

7. Wykrywanie słowa wybudzającego: budowanie aplikacji 125
Co będziemy tworzyć? 126
Architektura aplikacji 127
Wprowadzenie do naszego modelu 127
Wszystkie elementy aplikacji 129
Omówienie testów 131
Podstawowy przepływ danych 131
Element dostarczający dane audio 135
Element dostarczający cechy 136
Element rozpoznający polecenia 142
Element reagujący na polecenia 147
Nasłuchiwanie słów wybudzających 148
Uruchomienie naszej aplikacji 151
Uruchomienie aplikacji na mikrokontrolerach 152
Arduino 152
SparkFun Edge 158
Zestaw ST Microelectronics STM32F746G Discovery 168
Podsumowanie 172

8. Wykrywanie słowa wybudzającego: trenowanie modelu 173
Trenowanie naszego nowego modelu 174
Trenowanie w Colab 174
Wykorzystanie modelu w naszym projekcie 187
Zastępowanie modelu 187
Zmiana etykiet 188
Zmiany w kodzie command_responder.cc 188
Inne sposoby uruchamiania skryptów 190
Zasada działania modelu 192
Wizualizacja danych wejściowych 192
Zasada działania generowania cech 195

Kup ksi k Pole ksi k

Architektura modelu 197
Dane wyjściowe modelu 200
Trenowanie modelu z własnymi danymi 201
Zestaw danych Speech Commands 201
Trenowanie modelu na własnych danych 202
Nagrywanie własnych dźwięków 203
Powiększenie zestawu danych 205
Architektury modeli 205
Podsumowanie 206

9. Wykrywanie osoby: budowanie aplikacji 207
Co będziemy budować? 208
Architektura aplikacji 209
Wprowadzenie do naszego modelu 210
Wszystkie elementy aplikacji 211
Omówienie testów 212
Podstawowy przepływ danych 213
Element dostarczający obrazy 216
Element reagujący na wykrycie człowieka 217
Wykrywanie ludzi 218
Uruchomienie aplikacji na mikrokontrolerach 221
Arduino 221
SparkFun Edge 229
Podsumowanie 239

10. Wykrywanie osoby: trenowanie modelu 240
Wybór maszyny 240
Konfiguracja instancji Google Cloud Platform 240
Wybór platformy programistycznej do treningu 248
Tworzenie zestawu danych 248
Trenowanie modelu 249
TensorBoard 251
Ocena modelu 253
Eksportowanie modelu do TensorFlow Lite 253
Eksportowanie do pliku GraphDef Protobuf 253
Zamrażanie wag 254
Kwantyzacja i konwertowanie na potrzeby TensorFlow Lite 254
Konwertowanie na plik źródłowy C 255
Trenowanie dla innych kategorii 255
Architektura MobileNet 256
Podsumowanie 256

Kup ksi k Pole ksi k

11. Magiczna różdżka: budowanie aplikacji ... 257
Co będziemy tworzyć? 260
Architektura aplikacji 261
Wprowadzenie do naszego modelu 261
Wszystkie elementy aplikacji 262
Omówienie testów 263
Podstawowy przepływ danych 263
Element obsługujący akcelerometr 267
Element przewidujący gesty 268
Element reagujący na wykrycie gestu 271
Wykrywanie gestu 271
Uruchomienie aplikacji na mikrokontrolerach 274
Arduino 275
SparkFun Edge 286
Podsumowanie 300

12. Magiczna różdżka: trenowanie modelu ... 301
Trenowanie modelu 302
Trening w Colab 302
Inne sposoby uruchamiania skryptów 310
Zasada działania modelu 310
Wizualizacja danych wejściowych 310
Architektura modelu 313
Trenowanie modelu z własnymi danymi 318
Przechwytywanie danych 319
Modyfikacja skryptów trenujących 321
Trening 322
Wykorzystanie nowego modelu 322
Podsumowanie 322
Uczenie się uczenia maszynowego 323
Co dalej? 323

13. TensorFlow Lite dla mikrokontrolerów ... 324
Czym jest TensorFlow Lite dla mikrokontrolerów? 324
TensorFlow 324
TensorFlow Lite 325
TensorFlow Lite dla mikrokontrolerów 325
Wymagania 326
Dlaczego model potrzebuje interpretera? 328
Generowanie projektu 329
Kompilatory 330
Wyspecjalizowany kod 331

Kup ksi k   Pole ksi k

Pliki Makefile 335
Pisanie testów 337
Obsługa nowej platformy sprzętowej 338
Wyświetlanie rejestru zdarzeń 339
Wdrożenie funkcji DebugLog() 341
Uruchamianie wszystkich plików źródłowych 343
Integracja z plikami Makefile 343
Obsługa nowego IDE lub kompilatora 344
Integrowanie zmian w kodzie projektu z repozytoriami 345
Wnoszenie swojego wkładu do kodu z otwartym źródłem 346
Obsługa nowego akceleratora sprzętowego 347
Format pliku 348
Biblioteka FlatBuffers 349
Przenoszenie operacji TensorFlow Lite Mobile na wersję dla mikrokontrolerów 355
Oddzielanie kodu odniesienia 356
Utworzenie kopii operatora dla mikrokontrolera 356
Tworzenie wersji testów dla mikrokontrolerów 357
Tworzenie testu Bazel 358
Dodanie swojego operatora do obiektu AllOpsResolver 358
Kompilacja testu pliku Makefile 358
Podsumowanie 359

14. Projektowanie własnych aplikacji TinyML 360
Projektowanie 360
Czy potrzebny jest mikrokontroler, czy może być większe urządzenie? 361
Co jest możliwe? 362
Podążanie czyimiś śladami 362
Podobne modele do trenowania 363
Sprawdzenie danych 364
Magia „Czarnoksiężnika z krainy Oz” 365
Poprawnie działająca wersja na komputerze jako pierwszy etap 366

15. Optymalizacja prędkości działania programu 367
Prędkość modelu a prędkość ogólna aplikacji 367
Zmiany sprzętu 368
Ulepszenia modelu 368
Ocena opóźnienia modelu 369
Przyspieszanie modelu 370
Kwantyzacja 370
Etap projektowania produktu 372
Optymalizacje kodu 372
Profilowanie wydajności 373

Kup ksi k Pole ksi k

Optymalizowanie operacji 374
Implementacje już zoptymalizowane 375
Tworzenie własnej zoptymalizowanej implementacji 375
Wykorzystanie funkcjonalności sprzętu 378
Akceleratory i koprocesory 378
Wnoszenie swojego wkładu do kodu z otwartym źródłem 379
Podsumowanie 380

16. Optymalizacja poboru mocy 381
Rozwijanie intuicji 381
Pobór mocy standardowych elementów 382
Wybór sprzętu 383
Pomiar rzeczywistego poboru mocy 384
Oszacowanie poboru mocy modelu 385
Ulepszenia związane z zużyciem energii 385
Cykl pracy 386
Projektowanie kaskadowe 386
Podsumowanie 387

17. Optymalizacja modelu i rozmiaru pliku binarnego 388
Zrozumienie ograniczeń własnego systemu 388
Oszacowanie zużycia pamięci 389
Zużycie pamięci flash 389
Zużycie pamięci RAM 390
Szacunkowe wartości dokładności i rozmiaru modelu przy różnych problemach 391
Model rozpoznający słowa wybudzające 392
Model predykcyjnego utrzymania 392
Wykrywanie obecności człowieka 392
Wybór modelu 393
Zmniejszenie rozmiaru pliku wykonywalnego 393
Mierzenie rozmiaru kodu 393
Ile miejsca zajmuje TensorFlow Lite dla mikrokontrolerów? 394
OpResolver 394
Rozmiar pojedynczych funkcji 396
Stałe w platformie TensorFlow Lite 398
Naprawdę malutkie modele 399
Podsumowanie 399

18. Debugowanie 400
Różnica w dokładności między treningiem a wdrożeniem 400
Różnice we wstępnym przetwarzaniu danych 400
Debugowanie wstępnego przetwarzania danych 401
Ocena działania programu na urządzeniu docelowym 402

Kup ksi k Pole ksi k

Różnice liczbowe 403
Czy różnice stanowią problem? 403
Ustalenie wskaźnika 403
Punkt odniesienia 404
Zamiana implementacji 404
Tajemnicze awarie 405
Debugowanie na pulpicie 405
Sprawdzanie rejestru 405
Debugowanie metodą strzelby 406
Błędy związane z pamięcią 407
Podsumowanie 408

19. Przenoszenie modelu z TensorFlow do TensorFlow Lite 409
Określenie wymaganych operacji 409
Operacje obsługiwane w TensorFlow Lite 410
Przeniesienie wstępnego i końcowego przetwarzania do kodu aplikacji 410
Implementacja niezbędnych operacji 412
Optymalizacja operacji 412
Podsumowanie 412

20. Prywatność, bezpieczeństwo i wdrażanie 413
Prywatność 413
PDD 413
Używanie PDD 415
Bezpieczeństwo 416
Ochrona modeli 416
Wdrożenie 417
Przejście od płytki do produktu 418
Podsumowanie 418

21. Poszerzanie wiedzy 419
Fundacja TinyML 419
SIG Micro 419
Strona internetowa TensorFlow 420
Inne platformy programistyczne 420
Twitter 420
Przyjaciele TinyML 420
Podsumowanie 421

A Używanie i tworzenie biblioteki Arduino w formacie ZIP 423

B Przechwytywanie dźwięku na Arduino 425

Kup ksi k Pole ksi k

ROZDZIAŁ 4.

„Witaj, świecie” TinyML: budowa i trenowanie modelu

W rozdziale 3. poznałeś podstawowe koncepcje uczenia maszynowego i ogólny przebieg procesu. W tym i następnym rozdziale zaczniemy wprowadzać tę wiedzę w życie. Zbudujemy i wytrenujemy model od zera, a następnie zintegrujemy go z prostym programem dla mikrokontrolera.

Podczas tego procesu ubrudzisz sobie ręce kilkoma narzędziami programistycznymi, które są używane każdego dnia przez nowatorskich praktyków uczenia maszynowego. Ponadto dowiesz się, jak zintegrować model uczenia maszynowego z programem napisanym w C++ i uruchomić go na mikrokontrolerze, by sterować przepływem prądu w obwodzie. Pierwszy raz poczujesz, jak to jest połączyć sprzęt z uczeniem maszynowym, co powinno Ci się spodobać!

Możesz przetestować swój kod napisany w tych rozdziałach na swoim komputerze z systemem macOS, Linux lub Windows, ale żeby doświadczyć w pełni prezentowanych tutaj umiejętności, będziesz potrzebował jednego z urządzeń z systemem wbudowanym wspomnianych w podpunkcie „Jaki sprzęt będzie Ci potrzebny” w rozdziale 2.:

- Arduino Nano 33 BLE Sense (*https://store.arduino.cc/arduino-nano-33-ble-sense-with-headers*),
- SparkFun Edge (*https://www.sparkfun.com/products/15170*),
- zestawu STM32F746G Discovery z mikrokontrolerem i ekranem dotykowym (*https://os.mbed.com/platforms/ST-Discovery-F746NG/*).

Aby stworzyć nasz model uczenia maszynowego, użyjemy Pythona oraz programów TensorFlow i Google Colaboratory, który jest opartym na chmurze interaktywnym notatnikiem do eksperymentowania z kodem napisanym w języku Python. To kilka z najważniejszych narzędzi używanych na co dzień przez inżynierów uczenia maszynowego, a wszystkie są darmowe.

Zastanawia Cię tytuł tego rozdziału? To już tradycja w programowaniu, że nowa technologia jest wprowadzana za pomocą przykładowego programu, który pokazuje, jak zrobić coś prostego. Często takie zadanie polega na stworzeniu programu, który wyświetla angielskie słowa *Hello, world!* („Witaj, świecie”) (*https://pl.wikipedia.org/wiki/Hello_world*). Nie ma dokładnego odpowiednika w uczeniu maszynowym, ale stosujemy takie określenie, aby się odnieść do prostego, łatwego do odczytania przykładu kompletnej aplikacji TinyML.

Kup książkę Poleć książkę

W tym rozdziale zrealizujemy następujące zadania:

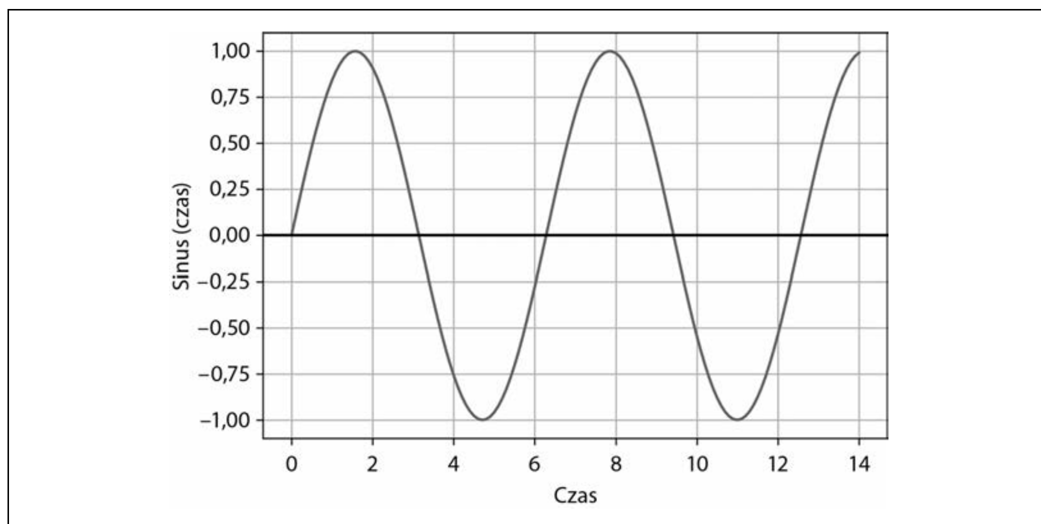
1. Uzyskamy prosty zestaw danych.
2. Wytrenujemy model uczenia głębokiego.
3. Ocenimy skuteczność modelu.
4. Przekształcimy model, by móc go uruchomić na urządzeniu docelowym.
5. Napiszemy kod procesu wnioskowania, który będzie uruchamiany na urządzeniu docelowym.
6. Skompilujemy kod.
7. Uruchomimy skompilowany kod na mikrokontrolerze.

Kod, z którego będziemy korzystać, jest dostępny w repozytorium TensorFlow na GitHubie (*https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/micro*).

Zalecamy, abyś najpierw przeczytał wszystkie podrozdziały, a dopiero potem spróbował uruchomić kod. Znajdziesz tutaj instrukcje, jak to zrobić. Ale zanim zaczniemy, omówmy, co w ogóle będziemy budować.

Co będziemy budować?

W rozdziale 3. omówiliśmy, jak sieci uczenia głębokiego uczą się znajdować wzorce w swoich danych treningowych do prognozowania. Zaraz wytrenujemy sieć, aby wymodelować bardzo proste dane. Zapewne słyszałeś o funkcji sinus (*https://pl.wikipedia.org/wiki/Funkcje_trygonometryczne*). Jest używana w trygonometrii do opisu własności trójkątów prostokątnych. Naszymi danymi treningowymi będzie sinusoida (*https://pl.wikipedia.org/wiki/Fala_sinusoidalna*), która jest wykresem funkcji sinus w czasie (zobacz rysunek 4.1).



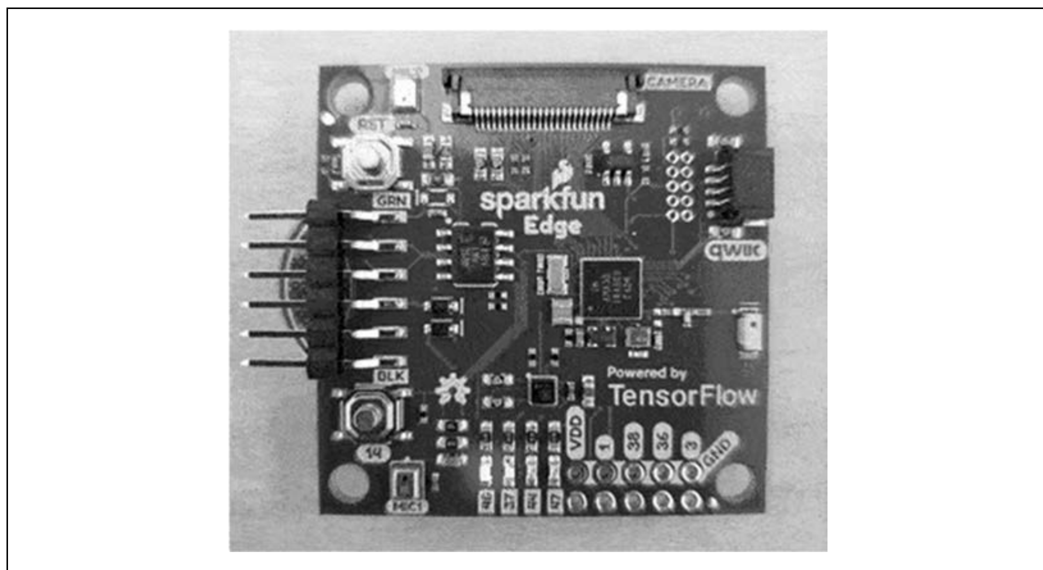
Rysunek 4.1. Fala sinusoidalna

Kup ksi k Pole ksi k

Naszym celem jest wytrenowanie modelu, który może pobrać wartość *x* i przewidzieć jej sinus, *y*. W rzeczywistości, jeżeli chciałbyś wyznaczyć sinus *x*, mógłbyś go po prostu obliczyć. Jednak przez wytrenowanie modelu, by uzyskać przybliżony wynik, możemy pokazać podstawy uczenia maszynowego.

Druga część naszego projektu będzie polegać na uruchomieniu modelu na urządzeniu. Z wyglądu sinusoida jest regularną krzywą, która biegnie od –1 do 1 i z powrotem. Z tego powodu jest idealna do sterowania miłym dla oka pokazem świateł! Dane wyjściowe naszego modelu będą nam służyć do sterowania migającymi diodami LED lub graficzną animacją, w zależności od możliwości urządzenia.

Pod adresem *https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/images/animation_on_sparkfun_edge.gif* możesz zobaczyć animowany plik GIF pokazujący kod uruchomiony na SparkFun Edge. Rysunek 4.2 przedstawia jedno ujęcie tej animacji — dwie zapalone diody LED na płytce SparkFun Edge. Może nie jest to szczególnie praktyczne zastosowanie uczenia maszynowego, ale w duchu przykładów z serii „Witaj, świecie” ten projekt jest prosty, sprawia radość i pomaga przyswoić podstawowe zasady, które musisz znać.



Rysunek 4.2. Kod uruchomiony na SparkFun Edge

Gdy się upewnimy, że nasz podstawowy kod działa, będziemy go wdrażać na trzy różne urządzenia: SparkFun Edge, Arduino Nano 33 BLE Sense i ST Microelectronics STM32F746G Discovery.

Ponieważ przykłady z tej książki są częścią aktywnie rozwijającego się projektu z otwartym źródłem, ciągle się zmieniają, gdyż je optymalizujemy, naprawiamy błędy i dodajemy obsługę kolejnych urządzeń. Prawdopodobnie zauważysz różnice między kodem z książki a najnowszym kodem z repozytorium TensorFlow.[1] Pomimo tego, że kod może się z czasem nieco zmieniać, podstawowe zasady, które tu poznasz, pozostaną niezmienne.

[1] Dodatkowo dla wygody czytelnika przykłady zamieszczone w książce zostały spolonizowane — *przyp. red.*

Kup ksi k Pole ksi k

Nasz zestaw narzędzi do uczenia maszynowego

Do budowy części opartej na uczeniu maszynowym będziemy korzystać z tych samych narzędzi, z których na co dzień korzystają inżynierowie zajmujący się tym zawodowo. Ten podrozdział to wprowadzenie, które pomoże Ci rozpocząć z nimi pracę.

Python i Jupyter Notebooks

Python jest ulubionym językiem programowania naukowców i inżynierów zajmujących się uczeniem maszynowym. Łatwo się go nauczyć, sprawdza się dobrze w wielu różnych zastosowaniach i ma mnóstwo przydatnych bibliotek dla zadań związanych z danymi i obliczeniami matematycznymi. Ogromna większość badań związana z uczeniem głębokim jest wykonywana z użyciem Pythona, a badacze często dzielą się kodem źródłowym Pythona stworzonych przez siebie modeli.

Python w połączeniu z narzędziem o nazwie Jupyter Notebooks (*https://jupyter.org/*) jest wyjątkowo wygodny. Jest to szczególny format dokumentu, który pozwala na łączenie tekstu, grafiki i kodu, który może być uruchomiony przez kliknięcie przycisku. Notatniki Jupyter są szeroko stosowane jako metoda opisu, wyjaśniania i badania kodu uczenia maszynowego oraz problemów.

Nasz model będziemy tworzyć w notatniku Jupyter, który umożliwia robienie niesamowitych rzeczy w celu wizualizacji naszych danych podczas tworzenia oprogramowania, w tym tworzenie wykresów, które pokazują dokładność i zbieżność modelu.

Jeśli masz już jakieś doświadczenie programistyczne, łatwo nauczysz się pisać i czytać kod Pythona. Nie powinieneś mieć problemów z tym projektem.

Google Colaboratory

Do uruchamiania naszego notatnika będziemy używać narzędzia, które nazywa się Colaboratory (*https://colab.research.google.com/notebooks/intro.ipynb*), w skrócie Colab. To narzędzie zostało stworzone przez Google i zapewnia środowisko online do uruchamiania notatników Jupyter. Jest dostępne za darmo, by wspierać badania i rozwój uczenia maszynowego.

Kiedyś musiałeś stworzyć notatnik lokalnie, na swoim komputerze. To wymagało instalacji wielu zależności, takich jak biblioteki Pythona, i zdarzało się, że była to droga przez mękę. Dzielenie się notatnikiem z innymi ludźmi również było trudne, gdyż mogli mieć różne wersje zależności, co wiązało się z niespodziewanym działaniem notatnika. Ponadto uczenie maszynowe wymaga sporej mocy obliczeniowej, zatem trenowanie modeli wykonywane lokalnie na komputerze może być wolne.

Colab umożliwia uruchamianie notatników na mocnym sprzęcie Google, bez żadnych dodatkowych kosztów. Możesz edytować i otwierać swoje notatniki w dowolnej przeglądarce i możesz się nimi dzielić z innymi ludźmi, którzy mają gwarancję, że otrzymają takie same rezultaty jak Ty. Możesz nawet skonfigurować Colab, by uruchamiać kod na specjalnie przyspieszonym sprzęcie, dzięki czemu można przeprowadzić trening szybciej niż na zwykłym komputerze.

Kup ksi k Pole ksi k

TensorFlow i Keras

TensorFlow (*https://www.tensorflow.org/*) to zestaw narzędzi do budowy, trenowania, oceny i wdrażania modeli uczenia maszynowego. Pierwotnie rozwijane przez Google, teraz jest projektem z otwartym źródłem, budowanym i utrzymywanym przez tysiące współautorów na całym świecie. To najpopularniejsza i powszechnie stosowana platforma programistyczna do uczenia maszynowego. Większość twórców oprogramowania korzysta z TensorFlow przez jego biblioteki Pythona.

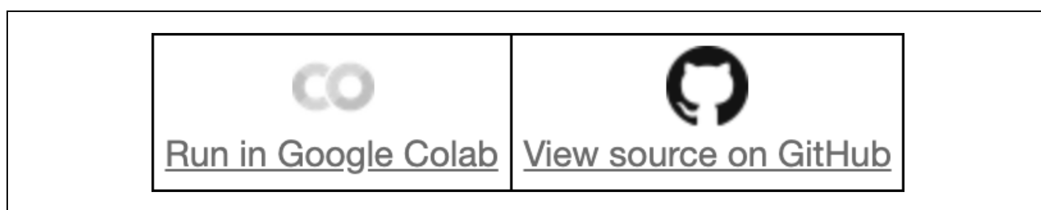
TensorFlow robi wiele różnych rzeczy. W tym rozdziale będziemy korzystać z interfejsu Keras (*https://www.tensorflow.org/guide/keras/sequential_model*), API wysokiego poziomu (interfejsu programowania aplikacji), który ułatwia budowanie i trenowanie sieci uczenia głębokiego. Będziemy również korzystać z TensorFlow Lite (*https://www.tensorflow.org/lite*), zestawu narzędzi do wdrażania modeli TensorFlow na urządzenia mobilne i z systemem wbudowanym, by uruchamiać nasz model bezpośrednio na urządzeniu.

W rozdziale 13. omówimy TensorFlow bardziej szczegółowo. Na tym etapie powinieneś wiedzieć, że to narzędzie daje dużo możliwości i jest wykorzystywane przez profesjonalistów, i że będzie spełniać Twoje wymagania zarówno na początku Twojej drogi, jak i wtedy, gdy już będziesz ekspertem w dziedzinie uczenia głębokiego.

Budowa naszego modelu

Przejdziemy teraz przez proces budowania, trenowania i przekształcania naszego modelu. W tym rozdziale zawrzemy cały kod, ale możesz również wykonywać zadania równolegle z czytaniem i uruchamiać kod na bieżąco w programie Colab.

Najpierw otwórz notatnik (*https://github.com/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/train/train_hello_world_model.ipynb*). Gdy strona się otworzy, kliknij przycisk *Run in Google Colab* (uruchom w Google Colab), który pokazano na rysunku 4.3. W ten sposób notatnik zostanie skopiowany z GitHuba na Colab, gdzie będziesz mógł go uruchamiać i edytować.



Rysunek 4.3. Przycisk Run in Google Colab

Domyślnie notatnik poza kodem zawiera próbkę danych wyjściowych, których powinieneś się spodziewać po uruchomieniu kodu. Jako że w tym rozdziale będziemy stopniowo wykonywać ten kod, usuńmy je wszystkie, by wrócić do stanu początkowego. Aby to zrobić, w menu programu Colab kliknij *Edytuj*, a następnie wybierz *Wyczyść wszystkie dane wyjściowe*, tak jak pokazano na rysunku 4.4.

Kup ksi k Pole ksi k

Problemy z wgraniem notatnika

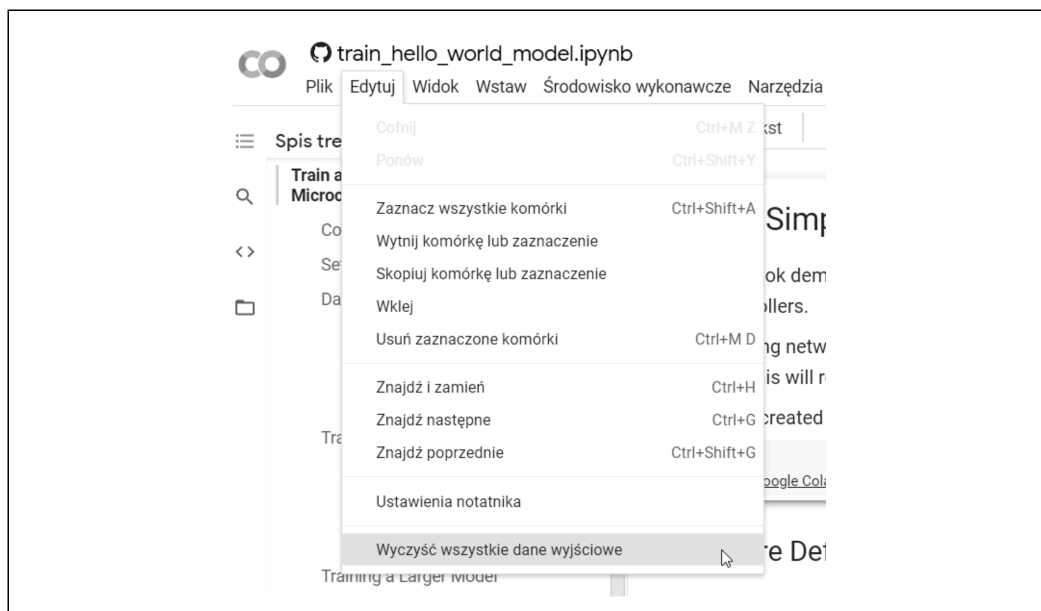
W czasie gdy ta książka powstawała, podczas wyświetlania notatników Jupyter często występował błąd (*https://github.com/jupyter/notebook/issues/3035*, opis błędu w języku angielskim). Jeśli próbując otworzyć notatnik, widzisz informację typu „Coś poszło nie tak. Wgrać ponownie?”, możesz otworzyć go bezpośrednio w programie Colab. Skopiuj fragment adresu URL notatnika na GitHubie, bez części początkowej *https://github.com/*:

/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/train/train_hello_world_model.ipynb

Następnie dodaj go do *https://colab.research.google.com/github/*, a w rezultacie otrzymasz pełny adres URL:

https://colab.research.google.com/github/tensorflow/tflite-micro/blob/main/tensorflow/lite/micro/examples/hello_world/train/train_hello_world_model.ipynb

Skopiuj powyższy odnośnik i wklej go w pasku adresu swojej przeglądarki, by otworzyć notatnik bezpośrednio w Colab.



Rysunek 4.4. Opcja Wyczyść wszystkie dane wyjściowe

Dobra robota. Nasz notatnik jest teraz gotowy do pracy!

Jeśli masz już pewne doświadczenie z uczeniem maszynowym, TensorFlow i interfejsem Keras, możesz przejść do podrozdziału, w którym przekształcamy nasz model na potrzeby TensorFlow Lite — „Konwertowanie modelu na potrzeby TensorFlow Lite”. W programie Colab zaś przewiń w dół, do nagłówka *Generate a TensorFlow Lite Model* (generowanie modelu TensorFlow Lite).

Kup ksi k

Pole ksi k

Importowanie pakietów

Nasze pierwsze zadanie polega na zaimportowaniu wszystkich niezbędnych zależności. W notatnikach Jupyter tekst i kod są umieszczone w **komórkach**. Są komórki *kodu* z wykonywalnym kodem Pythona i komórki *tekstowe*, które zawierają sformatowany tekst.

Nasza pierwsza komórka kodu znajduje się w punkcie *Import Dependencies* (import pakietów). Zawarty w niej kod ustawia wszystkie potrzebne biblioteki do wytrenowania i przekształcenia naszego modelu. Oto kod:

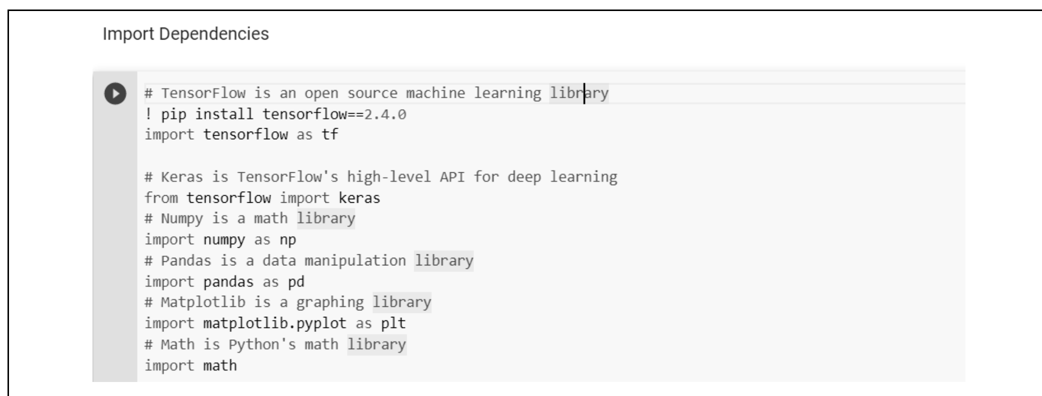
```
# TensorFlow jest biblioteką uczenia maszynowego z otwartym źródłem
! pip install tensorflow==2.4.0
import tensorflow as tf

# Keras to wysokopoziomowe API TensorFlow do uczenia głębokiego
from tensorflow import keras
# Numpy to biblioteka do obliczeń matematycznych
import numpy as np
# Pandas to biblioteka do przetwarzania danych
import pandas as pd
# Matplotlib to biblioteka ułatwiająca tworzenie wykresów
import matplotlib.pyplot as plt
# Math to biblioteka Pythona zawierająca działania matematyczne
import math
```

W Pythonie wyrażenie `import` wgrywa biblioteki, dzięki czemu można z nich korzystać w kodzie. Z kodu i komentarzy możesz wywnioskować, że ta komórka wykonuje następujące zadania:

- Instaluje TensorFlow 2.4.0 za pomocą polecenia `pip` — menedżera pakietów dla Pythona.
- Importuje biblioteki TensorFlow, NumPy, *matplotlib* i bibliotekę *math* Pythona.

Importując bibliotekę, możemy nadać jej skróconą nazwę (alias), tak by później było się łatwiej do niej odwoływać. I tak na przykład w podanym powyżej kodzie używamy `import numpy as np`, by zaimportować bibliotekę NumPy i nadać jej alias `np`. W ten sposób w naszym kodzie możemy odwoływać się do niej przez podanie skróconej nazwy `np`.

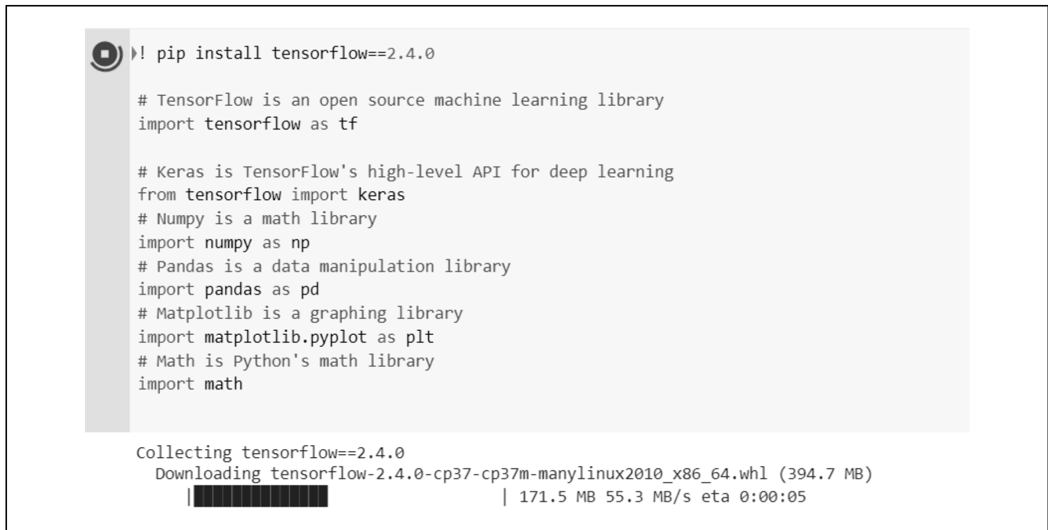
Kod w komórkach może być uruchamiany przez kliknięcie przycisku widocznego u góry po lewej stronie, po wybraniu danej komórki. W sekcji *Import Dependencies* kliknij w dowolnym miejscu w pierwszej komórce kodu, by ją wybrać. Na rysunku 4.5 widać, jak wygląda wybrana komórka.



Rysunek 4.5. Komórka importująca zależności po wybraniu

Kup ksi k Pole ksi k

Aby uruchomić kod, kliknij przycisk, który pojawia się po lewej stronie u góry. Gdy kod się wykonuje, wokół przycisku pojawia się animowane kółko, co zostało pokazane na rysunku 4.6.



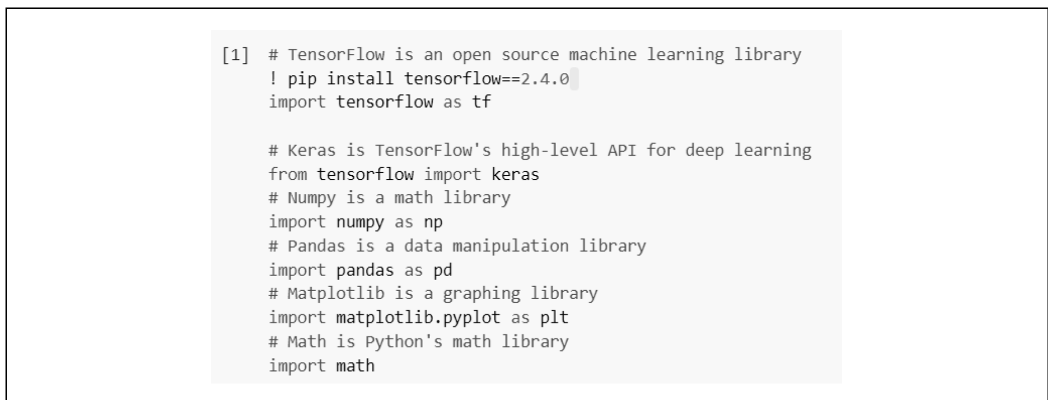
```
! pip install tensorflow==2.4.0

# TensorFlow is an open source machine learning library
import tensorflow as tf

# Keras is TensorFlow's high-level API for deep learning
from tensorflow import keras
# Numpy is a math library
import numpy as np
# Pandas is a data manipulation library
import pandas as pd
# Matplotlib is a graphing library
import matplotlib.pyplot as plt
# Math is Python's math library
import math

Collecting tensorflow==2.4.0
  Downloading tensorflow-2.4.0-cp37-cp37m-manylinux2010_x86_64.whl (394.7 MB)
     |█████████████                    | 171.5 MB 55.3 MB/s eta 0:00:05
```

Rysunek 4.6. Komórka importująca zależności po uruchomieniu

Zależności zaczną się instalować i zobaczysz pojawiające się informacje wyjściowe. Na końcu powinna się wyświetlić linijka informująca o pomyślnym zainstalowaniu biblioteki TensorFlow:

```
Successfully installed tensorboard-2.4.0 tensorflow-2.4.0 tensorflowestimator-2.4.0
```

Po wykonaniu komórki, gdy komórka nie jest już wybrana, w lewym górnym rogu wyświetla się cyfra 1, co widać na rysunku 4.7. Ta cyfra wzrasta o jeden po każdym uruchomieniu komórki.



```
[1]  # TensorFlow is an open source machine learning library
     ! pip install tensorflow==2.4.0
     import tensorflow as tf

     # Keras is TensorFlow's high-level API for deep learning
     from tensorflow import keras
     # Numpy is a math library
     import numpy as np
     # Pandas is a data manipulation library
     import pandas as pd
     # Matplotlib is a graphing library
     import matplotlib.pyplot as plt
     # Math is Python's math library
     import math
```

Rysunek 4.7. Licznik uruchomień komórki widoczny w lewym górnym rogu

Dzięki temu masz informacje, która komórka była już wykonywana i ile razy.

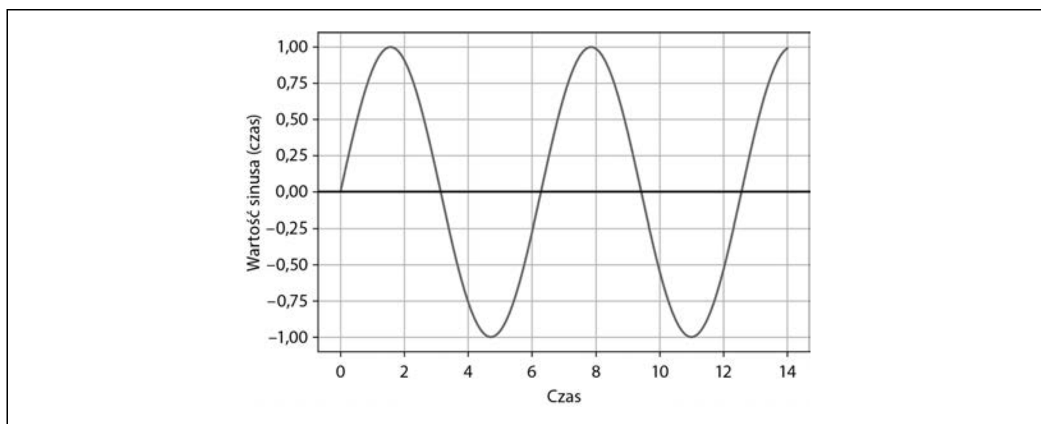
Kup ksi k Pole ksi k

Generowanie danych

Sieci uczenia głębokiego uczą się tworzyć wzorce z dostarczonych danych. Jak już wspomnieliśmy, wytrenujemy sieć w celu przygotowania modelu danych wygenerowanych przez funkcję sinus. W rezultacie otrzymamy model, który pobiera wartość x i przewiduje jej sinus — y.

Zanim przejdziemy dalej, potrzebujemy danych. W rzeczywistym zastosowaniu pewnie zbieralibyśmy dane z czujników i rejestratorów procesu produkcyjnego. Jednak w tym przykładzie użyjemy prostego kodu do wygenerowania zestawu wartości.

Kod w następnej komórce jest za to odpowiedzialny. Planujemy wygenerować 1000 wartości, które reprezentują losowe punkty na sinusoidzie. Spójrzmy na rysunek 4.8, by przypomnieć sobie, jak wygląda fala sinusoidalna.



Rysunek 4.8. Sinusoida

Każdy pełny cykl fali jest nazywany **okresem**. Z wykresu możemy odczytać, że jeden okres powtarza się co mniej więcej 6 jednostek na osi x. Właściwie okres sinusoidy wynosi $2 \cdot \pi$ lub 2π.

W celu uzyskania pełnej sinusoidy z danymi, z którymi warto przeprowadzić trening, nasz kod wygeneruje losowe wartości x od 0 do 2π. Następnie dla każdej z tych wartości obliczy sinusa.

Oto cały kod dla tej komórki, który wykorzystuje bibliotekę NumPy (`np`, zaimportowanej na początku), by wylosować liczby i obliczyć wartość sinusa:

```
# Wygenerujemy taką liczbę losowych danych.
SAMPLES = 1000

# Ustawienie wartości ziarna, abyśmy otrzymywali te same liczby losowe przy każdym uruchomieniu notatnika.
# Można podać dowolną liczbę.
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)

# Wygenerowanie równomiernie rozłożonego zestawu liczb losowych z zakresu od 0 do 2π, co pokrywa cały okres sinusoidy
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Przetasowanie liczb, by nie były ustawione w kolejności
np.random.shuffle(x_values)
```


Kup ksi k Pole ksi k

```
# Obliczenie wartości sinusa
y_values = np.sin(x_values)

# Tworzenie wykresu danych. Argument 'b.' oznacza, że dane wykresu będą zaznaczone za pomocą niebieskich kropek
plt.plot(x_values, y_values, 'b.')
plt.show()
```

Poza tym, co już powiedzieliśmy, jest jeszcze kilka kwestii, które warto poruszyć. Przede wszystkim zauważysz, że używamy metody `np.random.uniform()`, by wygenerować nasze losowe wartości *x*. Ta metoda zwraca tablicę losowych liczb z określonego zakresu. Biblioteka NumPy zawiera wiele przydatnych metod, które działają na całych tablicach wartości, co jest bardzo wygodne, gdy pracujemy z danymi.

Po drugie, po wygenerowaniu danych tasujemy je. Jest to ważne, gdyż trenowanie w uczeniu głębokim zależy od dostarczonych danych, które są ułożone w naprawdę losowy sposób. Jeśli dane byłyby podane w kolejności, otrzymany model byłby mniej dokładny.

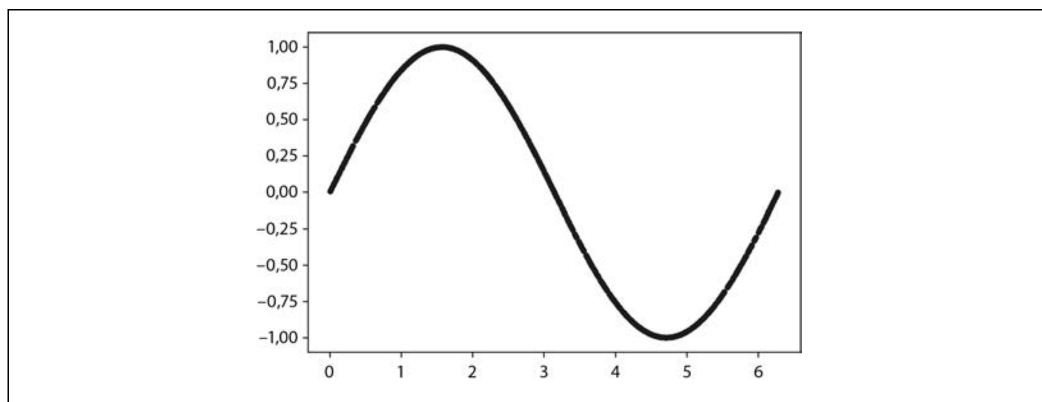
Następnie zauważ, że używany metody `sin()` z biblioteki NumPy, by obliczyć wartości sinusa. Biblioteka, o której tu mowa, może to zrobić za nas dla wszystkich naszych wartości *x* naraz, zwracając tablicę. Biblioteka NymPy jest świetna!

Na koniec zobaczysz pewien tajemniczy kod wywołujący `plt`, co jest naszym aliasem dla biblioteki *matplotlib*:

```
# Tworzenie wykresu danych. Argument 'b.' oznacza, że dane wykresu będą zaznaczone za pomocą niebieskich kropek.
plt.plot(x_values, y_values, 'b.')
plt.show()
```

Co ten kod robi? Tworzy wykres naszych danych. Jedną z najlepszych zalet notatników Jupyter jest to, że mogą wyświetlać grafikę, która jest wynikiem działania wykonanego kodu. Biblioteka *matplotlib* jest doskonałym narzędziem do tworzenia wykresów na podstawie danych. Jako że wizualizacja danych jest kluczową częścią procesu uczenia maszynowego, będzie to niesamowicie pomocne podczas treningu naszego modelu.

By wygenerować dane i przedstawić je w formie wykresu, uruchom kod w komórce. Gdy kod się wykona, pod kodem powinieneś zobaczyć wykres, taki jak ten pokazany na rysunku 4.9.



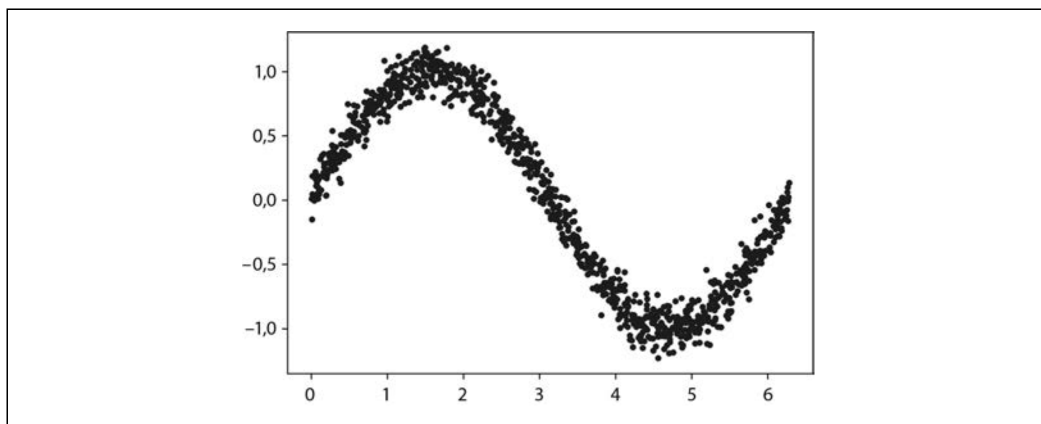
Rysunek 4.9. Wykres wygenerowanych danych

Kup ksi k   Pole ksi k

To są nasze dane! Jest to wybór losowych punktów wzdłuż płynnej fali sinusoidalnej. Moglibyśmy ich użyć, by wytrenować nasz model. Jednak to byłoby zbyt łatwe. Jedną z najbardziej fascynujących rzeczy w uczeniu głębokim jest zdolność wykrywania wzorców w zagmatwanych danych. Dzięki temu modele mogą prognozować, nawet jeśli są wytrenowane na nieuporządkowanych danych. Aby to udowodnić, dodajmy losowego szumu do naszych danych i narysujmy kolejny wykres.

```
# Dodanie małej losowej liczby do wartości y
y_values += 0.1 * np.random.randn(*y_values.shape)

# Wykres naszych danych
plt.plot(x_values, y_values, 'b.')
plt.show()
```

Uruchom tę komórkę i spójrz na rezultat, pokazany również na rysunku 4.10.



Rysunek 4.10. Wykres danych po dodaniu szumu

O wiele lepiej! Nasze punkty są teraz rozłożone wokół sinusoidy i nie tworzą już sinusoidy bez zakłóceń. To znacznie bardziej oddaje rzeczywistość, w której dane są ogólnie dość zagmatwane.

Rozdzielanie danych

Może pamiętasz z poprzedniego rozdziału, że zestaw danych dzielimy na trzy części: *trening*, *walidację* i *test*. W celu oceny dokładności trenowanego przez nas modelu musimy porównać jego prognozy z rzeczywistymi danymi i zobaczyć, jak się do siebie mają.

Taka ocena odbywa się w trakcie treningu (wtedy nazywamy ją walidacją) i po (wtedy mówimy o testach). To ważne, by w każdym z tych dwóch przypadków użyć świeżych danych, które nie były wykorzystane podczas trenowania modelu.

Aby się upewnić, że mamy dane, których możemy użyć do oceny, część z nich odłożymy przed rozpoczęciem treningu. Zarezerwujmy 20% danych na walidację i kolejne 20% na test. Pozostałe 60% wykorzystamy do wytrenowania modelu. Jest to typowy podział w procesie trenowania.

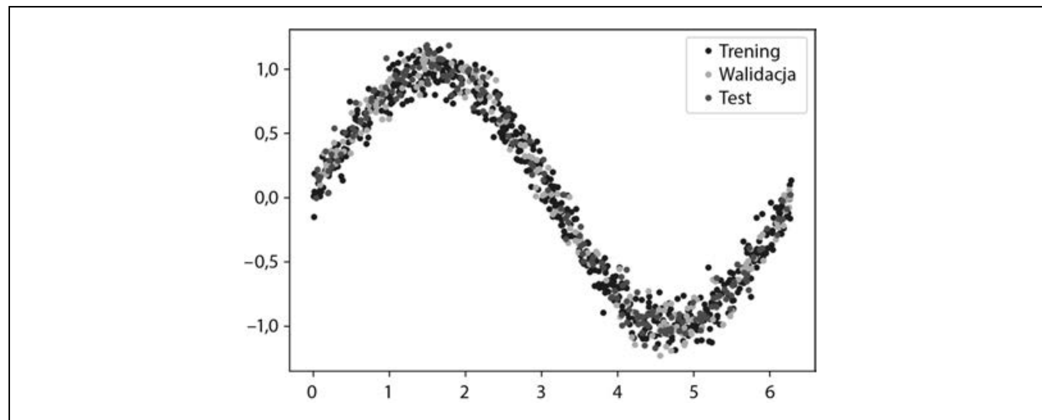
Poniżej pokazany kod dzieli nasze dane, a następnie tworzy wykres, na którym każda część danych jest zaznaczona innym kolorem.

Kup ksi k Pole ksi k

```
# Użyjemy 60% z naszych danych do trenowania, a 20% do testów.
# Pozostałe 20% zostanie użyte do walidacji. Oblicz indeks początku każdej części.
TRAIN_SPLIT = int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)

# Użyj np.split, by podzielić dane na trzy części.
# Drugim argumentem metody np.split jest tablica indeksów, zawierająca indeksy miejsc, w których tablica ma się podzielić.
# Musimy podać dwa indeksy, by dane zostały podzielone na trzy części.
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Upewnij się, że nasze dane poprawnie się podzieliły.
assert (x_train.size + x_validate.size + x_test.size) == SAMPLES

# Narysuj wykres, gdzie dane z każdej części będą oznaczone innym kolorem.
plt.plot(x_train, y_train, 'b.', label="Trening")
plt.plot(x_validate, y_validate, 'y.', label="Walidacja")
plt.plot(x_test, y_test, 'r.', label="Test")
plt.legend()
plt.show()
```

W celu podzielenia naszych danych użyjemy innej przydatnej metody z biblioteki NumPy — `split()`. Ta metoda pobiera tablicę danych i tablicę indeksów, a następnie dzieli dane na części według podanych indeksów.

Uruchom tę komórkę, by zobaczyć wynik podziału. Każdy typ danych jest przedstawiony za pomocą innego koloru (lub odcienia, jeśli czytasz drukowaną wersję tej książki), co widać na rysunku 4.11.



Rysunek 4.11. Dane wykresu podzielone na zestawy treningowe, walidacyjne i testowe

Definiowanie podstawowego modelu

Teraz, gdy mamy nasze dane, musimy stworzyć model, który wytrenujemy.

Stworzymy model pobierający wartość wejściową (w tym przypadku *x*), której użyje do przewidzenia liczbowej wartości wyjściowej (sinusa dla *x*). Taki typ problemu nazywamy **regresją**. Modele regresji możemy wykorzystać do każdego typu zadań, w którym wymagana jest liczbowa wartość

Kup ksi k Pole ksi k

wyjściowa. Na przykład model regresji mógłby przewidzieć prędkość w kilometrach na godzinę, z jaką biegnie dana osoba, na podstawie danych z akcelerometru.

By stworzyć model, zaprojektujemy prostą sieć neuronową. Model korzysta z warstw neuronów, by spróbować nauczyć się wzorców w danych treningowych, dzięki czemu będzie mógł przewidywać.

Potrzebny do tego kod jest dość prosty. Korzysta z Keras (*https://www.tensorflow.org/guide/keras/sequential_model*), wysokiego poziomu API oferowanego przez TensorFlow do tworzenia sieci uczenia głębokiego.

```
# Będziemy korzystać z API Keras, by stworzyć prostą architekturę modelu.
from tf.keras import layers
model_1 = tf.keras.Sequential()

# Pierwsza warstwa pobiera na wejściu skalar i przekazuje go do 16 neuronów. Neurony decydują, czy się aktywować,
# z wykorzystaniem funkcji aktywacji 'relu'.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Ostatnia warstwa składa się z pojedynczego neuronu, gdyż na wyjściu chcemy otrzymać jedną liczbę.
model_1.add(layers.Dense(1))

# Kompilacja modelu za pomocą standardowego optymalizatora i funkcji straty
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Wyświetlenie podsumowania architektury modelu
model_1.summary()
```

Najpierw za pomocą Keras tworzymy model sekwencyjny (`Sequential`), co oznacza po prostu model, w którym każda warstwa neuronów jest ułożona na kolejnej, tak jak widzieliśmy na rysunku 3.1. Następnie definiujemy dwie warstwy. Oto definicja pierwszej z nich:

```
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))
```

Pierwsza warstwa ma pojedyncze wejście — nasz *x* — i 16 neuronów. Jest to gęsta warstwa (`Dense`), zwana również *w pełni połączoną*, co oznacza, że wartość wejściowa będzie przekazywana do każdego z neuronów podczas procesu wnioskowania. Każdy neuron do pewnego stopnia zostanie *aktywowany*. Poziom aktywacji danego neuronu zależy zarówno od jego *wagi*, jak i *wartości przesunięcia* uzyskanych podczas treningu i w wyniku działania **funkcji aktywacji**. Poziom aktywacji neuronu ma wartość liczbową.

Aktywacja jest obliczana z prostego wzoru wyrażonego za pomocą Pythona. Nigdy nie będziemy musieli sami pisać do tego kodu, gdyż to działanie jest obsługiwane przez TensorFlow i Keras, ale warto, żebyś znał ten wzór, gdy będziesz wiedział już więcej o uczeniu głębokim.

```
activation = activation_function((input * weight) + bias)
```

By obliczyć poziom aktywacji neuronu (`activation`), jego wejście (`input`) jest mnożone przez jego wagę (`weight`), a do wyniku dodawana jest wartość przesunięcia (`bias`). Obliczona wartość jest przekazywana do funkcji aktywacji (`activation_function`). Otrzymany wynik jest poziomem aktywacji neuronu.

Funkcja aktywacji jest funkcją matematyczną służącą do nadania kształtu danej wyjściowej neuronu. W naszej sieci używamy funkcji aktywacji o nazwie **poprawiona funkcja liniowa** (ReLU, ang. *rectified linear unit*). W interfejsie Keras funkcja ta jest określana przez argument `activation=relu`.

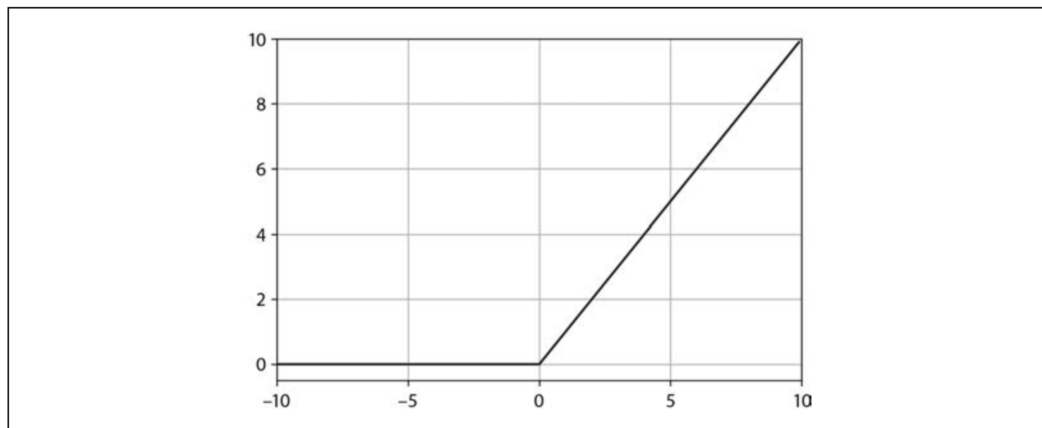
Kup ksi k Pole ksi k

ReLU jest prostą funkcją. Oto jej implementacja w Pythonie:

```
def relu(input):
    return max(0.0, input)
```

ReLU zwraca większą wartość z dwóch podanych: zero lub wartość wejściową. Jeśli na wejściu jest liczba ujemna, funkcja ReLU zwraca zero. Jeśli na wejściu jest liczba dodatnia, ReLU zwraca ją w niezmienionej postaci.

Rysunek 4.12 pokazuje dane zwracane przez funkcję ReLU dla zakresu wartości wejściowych.



Rysunek 4.12. Wykres funkcji ReLU dla wartości wejściowych z zakresu od –10 do 10

Bez funkcji aktywacji wyjście neuronu byłoby zawsze funkcją liniową swojego wejścia. W rezultacie sieć mogłaby modelować jedynie zależności liniowe, w których stosunek między *x* a *y* pozostaje taki sam przez cały zakres wartości. To uniemożliwiłoby sieci wymodelowanie naszej sinusoidy, gdyż fala sinusoidalna jest nieliniowa.

Jako że ReLU nie jest liniowa, pozwala mnożyć warstwy neuronów, by połączyć ich siły i tworzyć modele złożonych nieliniowych zależności, w których wartość *y* nie rośnie o tę samą liczbę przy każdym wzroście *x*.

Istnieją inne funkcje aktywacji, ale ReLU jest stosowana najczęściej. W Wikipedii, w artykule o funkcjach aktywacji, możesz znaleźć informacje na temat innych możliwości (*https://pl.wikipedia.org/wiki/Funkcja_aktywacji*). Każda funkcja ma swoje wady i zalety, a inżynierowie uczenia maszynowego eksperymentują, by znaleźć najlepszą funkcję aktywacji dla danej architektury.

Liczby aktywacji z naszej pierwszej warstwy będą przekazane jako dane wejściowe do drugiej warstwy, która jest zdefiniowana przez następującą linijkę:

```
model_1.add(layers.Dense(1))
```

Ponieważ ta warstwa ma tylko jeden neuron, przyjmie 16 wartości wejściowych od każdego neuronu z poprzedniej warstwy. Jej celem jest połączenie wszystkich aktywacji z poprzedniej warstwy w jedną wartość wyjściową. Jako że jest to nasza warstwa wyjścia, nie określamy żadnej funkcji aktywacji — chcemy dostać nieprzetworzony wynik.

Kup ksi k Pole ksi k

Ze względu na to, że ten neuron ma wiele wejść, dla każdego z nich ma określoną wagę. Wyjście neuronu jest obliczane według następującego wzoru:

```
# Wejścia i wagi są 16-elementowymi tablicami biblioteki NumPy.
output = sum((inputs * weights)) + bias
```

Wyjście (`output`) jest obliczane przez pomnożenie każdego wejścia (`inputs`) przez odpowiadającą mu wagę (`weights`) i zsumowanie wyników (`sum`), a następnie dodanie przesunięcia (`bias`) neuronu.

Przesunięcia i wagi sieci są efektem procesu trenowania. Pokazany wcześniej w tym rozdziale krok kompilacji (`compile()`) w kodzie ustawia kilka ważnych argumentów wykorzystywanych w procesie uczenia i przygotowuje model do treningu.

```
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
```

Argument `optimizer` określa algorytm, który dostosuje sieć do wymodelowania danych wejściowych podczas treningu. Jest wiele możliwości, a znalezienie najlepszej z nich sprowadza się do przeprowadzenia wielu prób. O dostępnych opcjach możesz przeczytać w dokumentacji (w języku angielskim) interfejsu Keras (*https://keras.io/api/optimizers/*).

Argument `loss` określa metodę używaną podczas treningu do obliczenia, jak daleko prognozy sieci są od rzeczywistości. Taka metoda jest nazywana **funkcją straty**. Tutaj używamy metody `mse` (ang. *mean squared error*, błąd średniokwadratowy). Ta funkcja straty jest stosowana w przypadku problemów regresji, gdy próbujemy przewidzieć liczbę. W interfejsie Keras jest wiele dostępnych funkcji straty, które są wymienione w dokumentacji (*https://keras.io/api/losses/*, strona w języku angielskim).

Argument `metrics` pozwala na określenie dodatkowych funkcji, które są używane do oceny wydajności modelu. Podajemy funkcję `mae` (ang. *mean absolute error*, średni błąd bezwzględny), która jest pomocna w mierzeniu osiągnięć modelu regresji. Ten wskaźnik będzie brany pod uwagę podczas treningu i będziemy mieć do niego dostęp po zakończeniu treningu.

Po skompilowaniu naszego modelu możemy za pomocą poniżej pokazanej linijce wyświetlić kilka informacji na temat jego architektury:

```
# Wyświetlenie podsumowania architektury modelu
model_1.summary()
```

Uruchom komórkę w programie Colab, by stworzyć nasz model. W rezultacie zobaczysz następujące dane wyjściowe:

```
Model: "sequential"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
dense (Dense) (None, 16) 32
_________________________________________________________________
dense_1 (Dense) (None, 1) 17
=================================================================
Total params: 49
Trainable params: 49
Non-trainable params: 0
_________________________________________________________________
```


Kup ksi k Pole ksi k

Ta tabela pokazuje warstwy sieci, kształty danych wyjściowych i liczbę ich *parametrów*. Rozmiar sieci — to, jak dużo pamięci zużywa — zależy głównie od liczby jej parametrów, czyli łącznej liczby wag i przesunięć. Ten wskaźnik może być przydatny, gdy mówimy o rozmiarze i złożoności modelu.

W przypadku prostych jak nasz modeli liczba wag może być ustalona przez obliczenie liczby połączeń między neuronami, przy założeniu, że każde połączenie ma wagę.

Sieć, którą właśnie zaprojektowaliśmy, składa się z dwóch warstw. Nasza pierwsza warstwa ma 16 połączeń, między wejściem a neuronami. Nasza druga warstwa ma jeden neuron, który również ma 16 połączeń, po jednym do każdego neuronu pierwszej warstwy. To daje łącznie 32 połączenia.

Skoro każdy neuron ma przesunięcie, sieć ma ich 17, co oznacza, że ma w sumie 32 + 17 = 49 parametrów.

Omówiliśmy kod, który definiuje nasz model. W następnym kroku rozpoczniemy proces trenowania.

Trenowanie naszego modelu

Po zdefiniowaniu naszego modelu nadszedł czas, by go wytrenować, a następie ocenić jego skuteczność. Patrząc na wskaźniki, możemy zadecydować, czy jest wystarczająco dobry lub czy powinniśmy wprowadzić zmiany do projektu i ponownie przeprowadzić trening.

Aby wytrenować model za pomocą interfejsu Keras, wystarczy wywołać metodę `fit()` i przekazać do niej wszystkie dane i kilka innych ważnych argumentów. Kod w następnej komórce pokazuje, jak to zrobić.

```
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
validation_data=(x_validate, y_validate))
```

Uruchom kod w komórce, by rozpocząć trening. Zobaczysz pojawiające się na bieżąco informacje o przebiegu treningu.

```
Train on 600 samples, validate on 200 samples
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae: 0.7848 -
↪val_loss: 0.5824 - val_mae: 0.6867
Epoch 2/1000
600/600 [==============================] - 0s 155us/sample - loss: 0.4883 - mae: 0.6194 -
↪val_loss: 0.4742 - val_mae: 0.6056
```

Nasz model właśnie trenuje. Zajmie to pewien czas, więc czekając, omówmy nasze wywołanie metody `fit()`:

```
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
validation_data=(x_validate, y_validate))
```

Przede wszystkim zauważysz, że wartość zwracaną przez funkcję `fit()` przypisujemy do zmiennej `history_1`. Ta zmienna zawiera mnóstwo informacji na temat przebiegu naszego treningu i wykorzystamy ją później, by sprawdzić, jak przebiegł.

Kup ksi k Pole ksi k

Następnie przyjrzymy się argumentom funkcji `fit()`:

`x_train`, `y_train`

Pierwsze dwa argumenty metody `fit()` to wartości `x` i `y` z naszych danych treningowych. Pamiętaj, że część danych odłożyliśmy do walidacji oraz testowania, więc tylko zestaw treningowy jest użyty do treningu.

`epochs`

Następny argument określa, ile razy nasz zestaw treningowy przejdzie przez sieć. Im więcej epok, tym dłuższy trening. Możesz pomyśleć, że im więcej model będzie trenował, tym lepiej. Jednak niektóre sieci zaczną przetrenowywać swoje dane po określonej liczbie epok, zatem możemy chcieć skrócić trening.

Co więcej, nawet jeśli model się nie przetrenowuje, sieć po określonej liczbie epok może przestać się ulepszać. Ze względu na to, że trening zajmuje czas i zasoby obliczeniowe, lepiej nie trenować dalej, jeśli nie ma dalszych postępów!

Zaczynamy od 1000 epok. Po zakończeniu treningu możemy przeanalizować nasze wskaźniki, by sprawdzić, czy podana liczba epok jest poprawna.

`batch_size`

Argument przed zmierzeniem dokładności sieci i przed zaktualizowaniem jej wag i przesunięć określa, jak wiele porcji danych treningowych do niej przekazać. Gdybyśmy chcieli, moglibyśmy podać 1, co oznaczałoby, że przeprowadzilibyśmy proces wnioskowania na pojedynczych danych z każdego źródła, zmierzyli stratę, zaktualizowali wagi i przesunięcia, by następnym razem prognozy były dokładniejsze, a następnie powtarzać te kroki dla reszty danych.

Ponieważ mamy 600 pojedynczych wartości, każda epoka skutkowałaby 600 aktualizacjami sieci. Wymaga to wielu obliczeń, przez co nasz trening trwałby wieki! Alternatywnym rozwiązaniem może być uruchomienie procesu wnioskowania na wielu grupach danych, zmierzenie łącznej straty, a następnie odpowiednie zaktualizowanie sieci.

Jeżeli argument `batch_size` ustawilibyśmy na 600, każda porcja zawierałaby wszystkie nasze dane treningowe. W ten sposób musielibyśmy zaktualizować naszą sieć tylko raz po każdej epoce — znacznie szybciej. Jednak skutkowałoby to mniej dokładnym modelem. Badania pokazały, że modele trenowane dużymi porcjami mają mniejszą zdolność do uogólniania nowych danych — są bardziej podatne na przetrenowanie.

Kompromis leży pośrodku. W naszym treningu stosujemy porcję wielkości 16. Oznacza to, że losowo wybierzemy 16 grup danych, uruchomimy proces wnioskowania, obliczymy łączną stratę i zaktualizujemy sieć. Zatem jeśli mamy 600 grup danych treningowych, sieć zostanie zaktualizowana około 38 razy na epokę, czyli o wiele lepiej niż 600.

Przy wybraniu rozmiaru porcji musimy wypośrodkować między wydajnością treningu a dokładnością modelu. Idealny rozmiar modelu będzie się różnił w zależności od modelu. Dobrze jest zacząć od rozmiaru 16 lub 32 i poeksperymentować, by zobaczyć, co działa lepiej.

`validation_data`

To tutaj podajemy nasz zestaw danych do walidacji. Dane z tego zestawu będą przepuszczane przez sieć podczas całego procesu treningowego, a prognozy sieci będą porównywane z oczekiwanymi wartościami. Wyniki walidacji będziemy mogli sprawdzić w informacjach o przebiegu treningu oraz w obiekcie `history_1`.

Kup ksi k Pole ksi k

Wskaźniki treningu

Miejmy nadzieję, że trening już się skończył. Jeśli nie, zaczekajmy jeszcze chwilę.

Teraz sprawdzimy kilka różnych wskaźników, by zobaczyć, jak przebiegł proces uczenia. Na początku spójrzmy na informacje z przebiegu treningu. Z nich dowiemy się, jakie sieć robiła postępy w porównaniu ze stanem początkowym.

Oto informacje dotyczące pierwszej i ostatniej epoki:

```
Epoch 1/1000
600/600 [==============================] - 1s 1ms/sample - loss: 0.7887 - mae: 0.7848 -
↪val_loss: 0.5824 - val_mae: 0.6867
Epoch 1000/1000
600/600 [==============================] - 0s 124us/sample - loss: 0.1524 - mae: 0.3039 -
↪val_loss: 0.1737 - val_mae: 0.3249
```

Wartości `loss`, `mae`, `val_loss` oraz `val_mae` mówią nam różne rzeczy.

`loss`

Jest to wyjście naszej funkcji straty. Używamy błędu średniokwadratowego, który jest wyrażany liczbą dodatnią. Ogólnie im mniejsza wartość straty, tym lepiej, więc jest to dobry wskaźnik do obserwacji.

Porównanie pierwszej i ostatniej epoki pokazuje, że sieć podczas treningu wyraźnie poprawiła swoje wyniki, zmniejszając stratę z około 0,7 do około 0,15. Spójrzmy na inne liczby, by sprawdzić, czy ta poprawa jest wystarczająca!

`mae`

To jest średni błąd bezwzględny naszych danych treningowych. Pokazuje średnią różnicę między prognozami sieci a spodziewanymi wartościami *y* z danych treningowych.

Możemy się spodziewać, że na początku nasz błąd będzie dość duży, zważywszy na to, że jest oparty na niewytrenowanej sieci. W tym przypadku prognozy sieci są rozbieżne średnio o 0,78, co jest dużą liczbą, gdy zakres akceptowalnych wartości mieści się w zakresie od –1 do 1!

Jednak nawet po szkoleniu nasz średni błąd bezwzględny wynosi około 0,30. Oznacza to, że nasze prognozy różnią się od właściwych o średnio 0,30, a to wciąż jest dość dużo.

`val_loss`

To jest wartość uzyskana w wyniku działania naszej funkcji straty na danych walidacyjnych. W ostatniej epoce strata treningu (około 0,15) jest nieco mniejsza niż strata walidacji (około 0,17). Jest to oznaka, że nasza sieć może się przetrenowywać, ponieważ działa gorzej na danych, z którymi wcześniej nie miała styczności.

`val_mae`

To jest średni błąd bezwzględny naszych danych walidacyjnych. Wartość 0,32 jest gorsza niż średni błąd bezwzględny naszych danych treningowych, co jest kolejnym sygnałem, że nasza sieć może się przetrenowywać.

Kup ksi k Pole ksi k

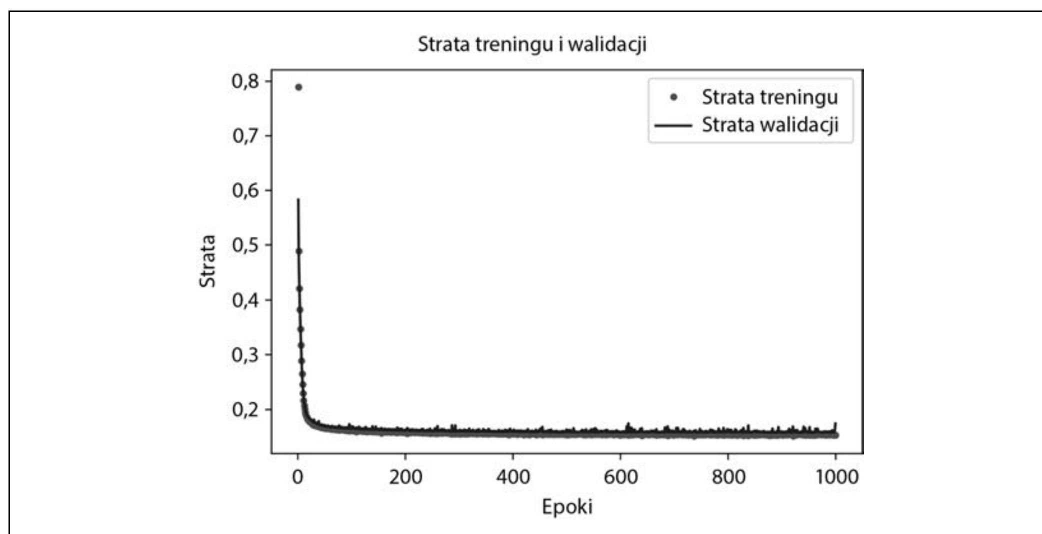
Wykres historii

Jak dotąd jest jasne, że nasz model nie prognozuje zbyt dobrze. Teraz musimy dowiedzieć się dlaczego. W tym celu użyjmy danych zebranych w naszym obiekcie `history_1`.

Następna komórka wyciąga straty treningu i walidacji z obiektu historii, a potem rysuje na ich podstawie wykres.

```
loss = history_1.history['loss']
val_loss = history_1.history['val_loss']
epochs = range(1, len(loss) + 1)

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

Obiekt `history_1` zawiera atrybut o nazwie `history_1.history`, który jest słownikiem wskaźników zebranych podczas treningu i walidacji. Wykorzystamy to do zebrania danych, które przedstawimy na wykresie. Na osi *x* umieścimy numery epok, a na osi *y* straty. Uruchom komórkę, a zobaczysz wykres widoczny na rysunku 4.13.



Rysunek 4.13. Wykres straty treningu i walidacji

Jak widzisz, wartość straty szybko spada przez pierwsze 50 epok, zanim zacznie się spłaszczać. Oznacza to, że model staje się coraz lepszy i prognozuje coraz dokładniej.

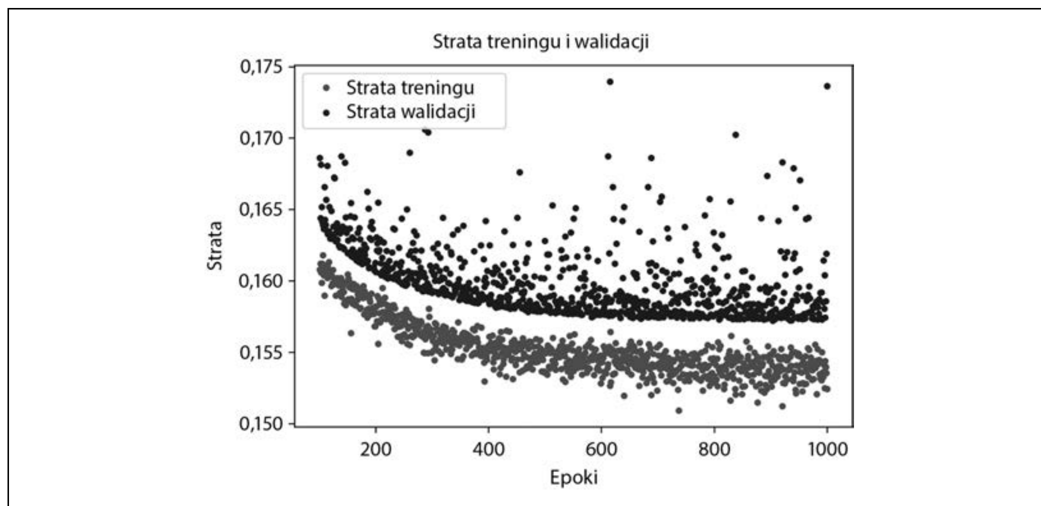
Naszym celem jest zatrzymanie treningu, gdy model nie udoskonala się już bardziej lub strata treningu jest mniejsza niż strata walidacji, co oznaczałoby, że model nauczył się przewidywać dane treningowe tak dobrze, że już nie potrafi uogólniać nowych danych.

Kup ksi k   Pole ksi k

Strata spada drastycznie w pierwszych kilku epokach, przez co reszta wykresu jest niemal nieczytelna. Pomińmy pierwsze 100 epok przez uruchomienie następnej komórki.

```
# Pominięcie pierwszych epok, by wykres był czytelniejszy
SKIP = 100

plt.plot(epochs[SKIP:], loss[SKIP:], 'g.', label='Strata treningu')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Strata walidacji')
plt.title('Strata treningu i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()
plt.show()
```

Rysunek 4.14 przedstawia wykres powstały w wyniku działania tej komórki.



Rysunek 4.14. Wykres straty treningu i walidacji z pominięciem pierwszych 100 epok

Po przybliżeniu widać, że strata się zmniejsza aż do około 600. epoki i od tego momentu jest w miarę stabilna. Oznacza to, że prawdopodobnie nie ma sensu trenować naszej sieci tak długo.

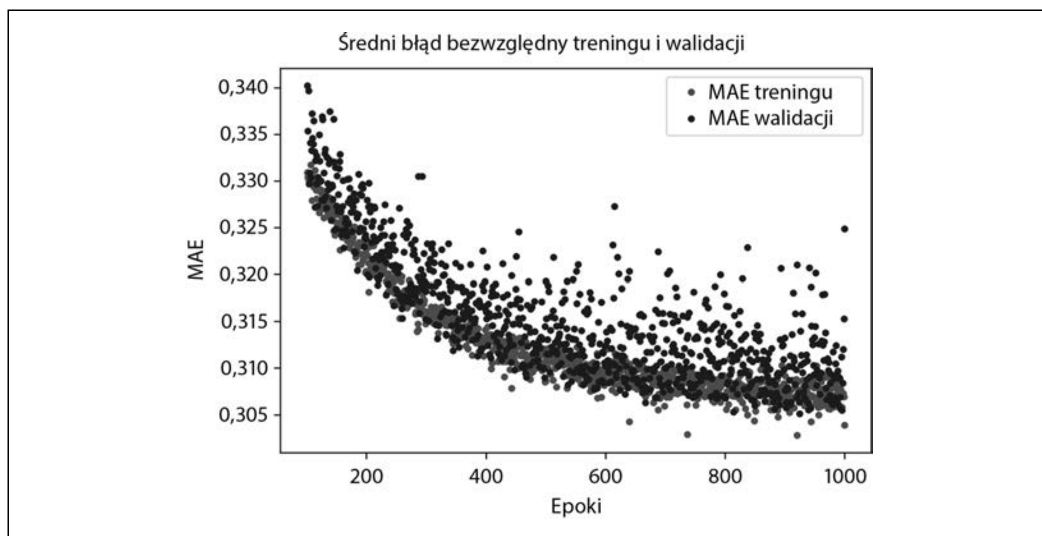
Jednak możesz zauważyć również, że najniższa strata wciąż wynosi około 0,15. Ta wartość wydaje się stosunkowo wysoka. Co więcej, wartości straty walidacji konsekwentnie rosną.

Aby dowiedzieć się więcej o działaniu naszego modelu, możemy przygotować wykres innych danych. Tym razem na wykresie umieścimy średni błąd bezwzględny. W tym celu uruchom następną komórkę.

```
# Rysowanie wykresu średniego błędu bezwzględnego, co jest kolejnym sposobem mierzenia błędów w prognozach
mae = history_1.history['mae']
val_mae = history_1.history['val_mae']

plt.plot(epochs[SKIP:], mae[SKIP:], 'g.', label='MAE treningu')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='MAE walidacji')
plt.title('Średni błąd bezwględny treningu i walidacji')
plt.xlabel('Epoki')
plt.ylabel('MAE')
plt.legend()
plt.show()
```


Kup ksi k Pole ksi k

Rysunek 4.15 przedstawia otrzymany wykres.



Rysunek 4.15. Wykres średniego błędu bezwzględnego podczas treningu i walidacji

Wykres średniego błędu bezwzględnego dostarcza nam kolejnych informacji. Widzimy, że średnio dane treningowe pokazują mniejszy błąd niż dane walidacyjne, co może oznaczać, że sieć się przetrenowała lub nauczyła danych tak sztywno, że nie jest w stanie trafnie prognozować na podstawie nowych danych.

Ponadto wartości średniego błędu bezwzględnego są dość wysokie, około 0,31, czyli niektóre z prognoz modelu odbiegają od poprawnego wyniku o 0,31. Jako że oczekujemy wartości z zakresu od –1 do 1, błąd rzędu 0,31 oznacza, że jesteśmy bardzo daleko od dokładnego modelowania sinusoidy.

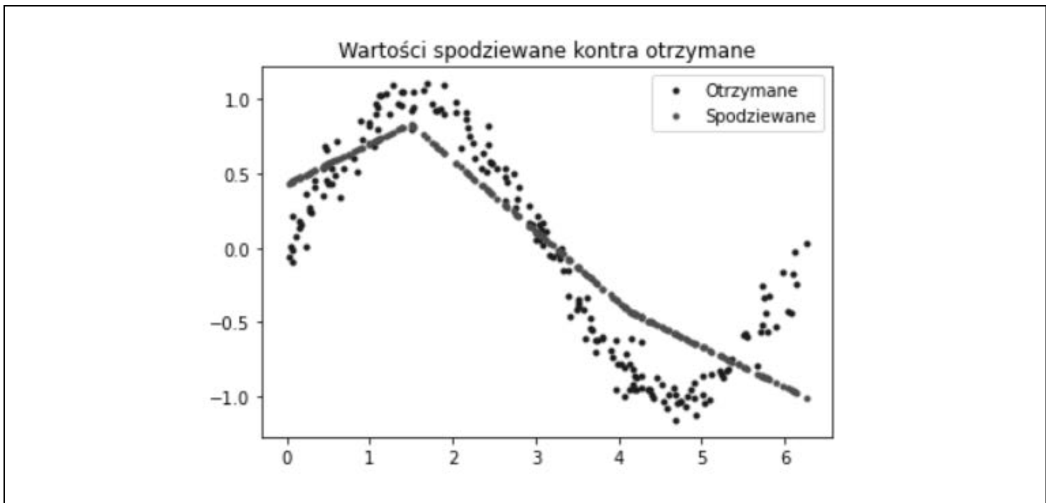
Aby rzucić więcej światła na to, co się dzieje, możemy narysować wykres prognozy naszej sieci dla danych treningowych w zestawieniu z oczekiwanymi wartościami.

Wykres jest sporządzany w następującej komórce:

```
# Użycie modelu do przewidywania na podstawie naszych danych treningowych
predictions = model_1.predict(x_train)

# Rysowanie wykresu danych wyjściowych w zestawieniu z oczekiwanymi danymi
plt.clf()
plt.title('Wartości spodziewane kontra otrzymane')
plt.plot(x_test, y_test, 'b.', label='Otrzymane')
plt.plot(x_train, predictions, 'r.', label='Spodziewane')
plt.legend()
plt.show()
```

Przez wywołanie metody `model_1.predict(x_train)` uruchamiamy proces wnioskowania na podstawie wszystkich wartości *x* z zestawu treningowego. Metoda zwraca tablicę prognoz. Nanieśmy jej elementy na wykres wraz z oczekiwanymi wartościami *y* z naszego zestawu treningowego. Uruchom kod z komórki, by zobaczyć wykres z rysunku 4.16.

Kup ksi k Pole ksi k



Rysunek 4.16. Wykres wartości prognozowanych w porównaniu z wartościami oczekiwanymi z naszego zestawu treningowego

Ojejku! Z wykresu jasno wynika, że sieć nauczyła się odwzorowywać naszą funkcję sinus w bardzo ograniczony sposób. Prognozy są wysoce liniowe i tylko w niewielu miejscach pasują do oczekiwanych danych.

Sztywność tego dopasowania świadczy o tym, że model nie ma wystarczającej zdolności do nauczenia się pełnej złożoności funkcji sinus i jest w stanie prognozować wartości tylko w nazbyt uproszczony sposób. Po powiększeniu naszego modelu jego skuteczność powinna się poprawić.

Ulepszenie naszego modelu

Mając tę cenną wiedzę, że nasz pierwotny model był za mały, by nauczyć się złożoności naszych danych, możemy spróbować go ulepszyć. Jest to standardowy etap procesu uczenia maszynowego: projekt modelu, ocena skuteczności i wprowadzanie zmian z nadzieją na uzyskanie lepszych wyników.

Prostym sposobem na powiększenie sieci jest dodanie kolejnej warstwy neuronów. Każda warstwa przetwarza dane wejściowe, by otrzymać dane wyjściowe jak najbardziej zbliżone do oczekiwanych wyników. Im więcej warstw sieci, tym bardziej złożone może być przekształcanie danych.

Uruchom następującą komórkę, by zredefiniować nasz model w ten sam sposób co wcześniej, ale z dodatkową warstwą 16 neuronów pośrodku.

```
model_2 = tf.keras.Sequential()

# Pierwsza warstwa pobiera na wejściu skalar i przekazuje go do 16 neuronów.
# Neurony decydują, czy się aktywować, z wykorzystaniem funkcji aktywacji 'relu'.
model_2.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Druga, nowa warstwa może pomóc sieci nauczyć się bardziej złożonych reprezentacji danych.
model_2.add(layers.Dense(16, activation='relu'))
```


Kup ksi k
Pole ksi k

```
# Ostatnia warstwa składa się z pojedynczego neuronu, gdyż na wyjściu chcemy otrzymać jedną liczbę.
model_2.add(layers.Dense(1))

# Kompilacja modelu za pomocą standardowego optymalizatora i funkcji straty
model_2.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Wyświetlenie podsumowania dla modelu
model_2.summary()
```

Jak widzisz, kod jest w zasadzie taki sam jak w naszym pierwszym modelu, ale z dodatkową gęstą warstwą (`Dense`). Uruchommy komórkę, by zobaczyć podsumowanie.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 16)                32
_________________________________________________________________
dense_3 (Dense)              (None, 16)                272
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
_________________________________________________________________
```

Mając dwie warstwy z 16 neuronami, nasz model jest o wiele większy. Ma $(1 \cdot 16) + (16 \cdot 16) + (16 \cdot 1) = 288$ wag i dodatkowo $16 + 16 + 1 = 33$ przesunięcia, co daje w sumie $288 + 33 = 321$ parametrów. Nasz pierwotny model miał tylko 49 parametrów, więc nowy model zwiększył się o 555%. Miejmy nadzieję, że ta dodatkowa pojemność pomoże przedstawić złożoność naszych danych.

Następna komórka przeprowadzi trening naszego nowego modelu. Ze względu na to, że nasz pierwszy model przestał ulepszać się tak szybko, skróćmy trening do 600 epok. Uruchom następującą komórkę, by rozpocząć trening:

```
history_2 = model_2.fit(x_train, y_train, epochs=600, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

Po zakończeniu treningu możemy spojrzeć na informacje z ostatniej epoki, aby zobaczyć, czy model stał się lepszy.

```
Epoch 600/600
600/600 [==============================] - 0s 150us/sample - loss: 0.0115 - mae: 0.0859 -
↪val_loss: 0.0104 - val_mae: 0.0806
```

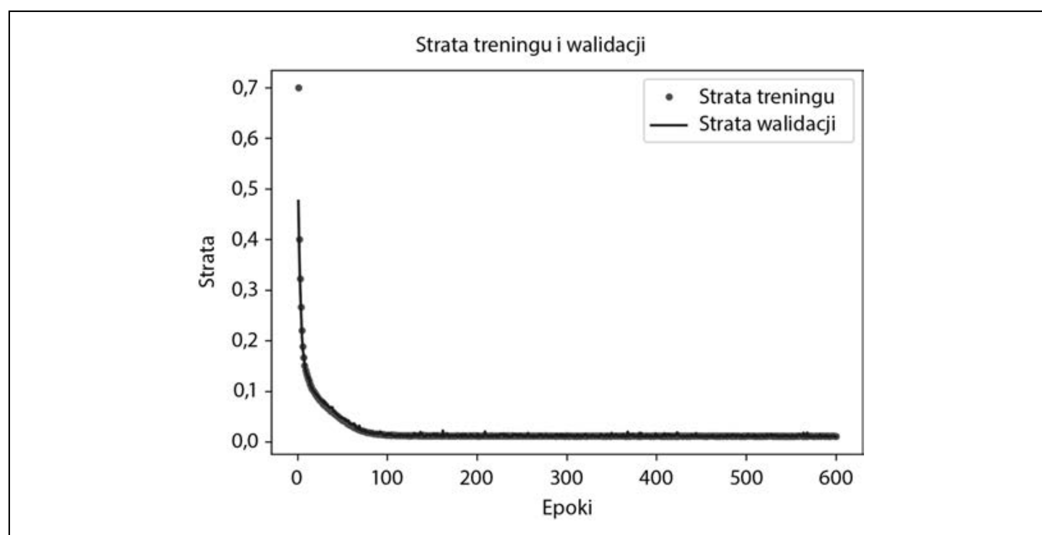
O! Możesz łatwo zobaczyć, że zrobiliśmy ogromny postęp — strata walidacji spadła z 0,17 do 0,01, a średni błąd bezwzględny walidacji z 0,32 do 0,08. Wygląda to bardzo obiecująco.

By zobaczyć, jak działa model, uruchommy następną komórkę, by stworzyć te same wykresy co wcześniej. Najpierw narysujemy wykres straty.

Kup ksi k Pole ksi k

```
# Rysowanie wykresu straty, która jest odległością wartości przewidzianej od oczekiwanej podczas treningu i walidacji
loss = history_2.history['loss']
val_loss = history_2.history['val_loss']

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

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

Rysunek 4.17 przedstawia wykres, który jest wynikiem działania kodu tej komórki.



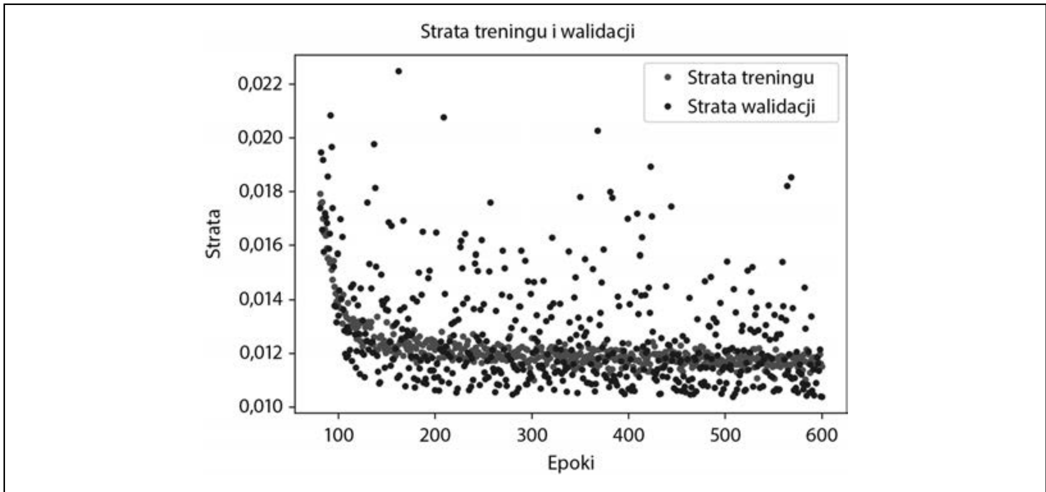
Rysunek 4.17. Wykres wartości straty treningu i walidacji

Następnie narysujemy ten sam wykres strat z pominięciem pierwszych 100 epok, by lepiej widzieć szczegóły.

```
# Pominięcie pierwszych epok, by wykres był czytelniejszy
SKIP = 100

plt.clf()
plt.plot(epochs[SKIP:], loss[SKIP:], 'g.', label=' Strata treningu')
plt.plot(epochs[SKIP:], val_loss[SKIP:], 'b.', label='Strata walidacji')
plt.title('Strata treningu i walidacji')
plt.xlabel('Epoki')
plt.ylabel('Strata')
plt.legend()
plt.show()
```

Rysunek 4.18 przedstawia otrzymany wykres.

Kup ksi k
Pole ksi k

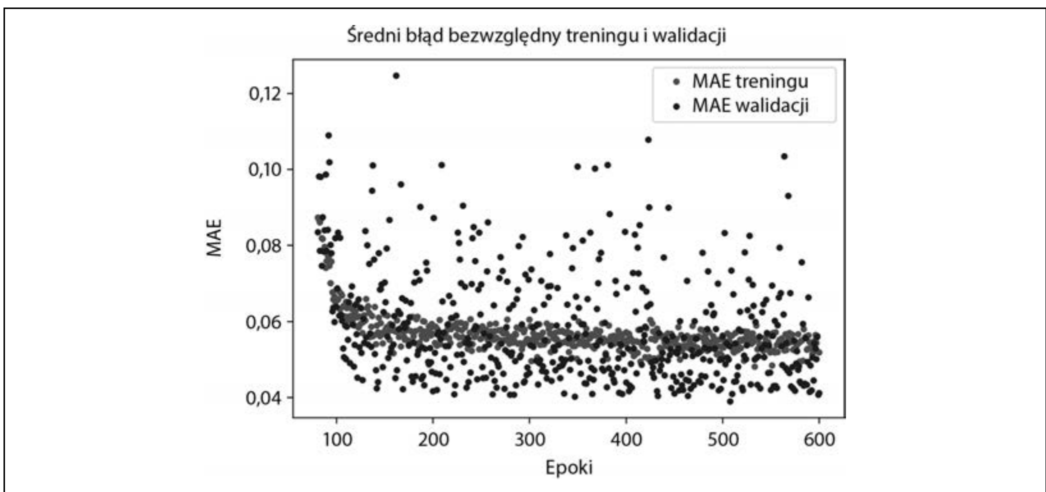


Rysunek 4.18. Wykres wartości straty treningu i walidacji z pominięciem pierwszych 100 epok

W końcu rysujemy wykres średniego błędu bezwzględnego dla tego samego zakresu epok.

```
plt.clf()

# Rysowanie wykresu średniego błędu bezwzględnego, co jest kolejnym sposobem mierzenia błędów w prognozach
mae = history_2.history['mae']
val_mae = history_2.history['val_mae']

plt.plot(epochs[SKIP:], mae[SKIP:], 'g.', label='MAE treningu')
plt.plot(epochs[SKIP:], val_mae[SKIP:], 'b.', label='MAE walidacji')
plt.title('Średni błąd bezwględny treningu i walidacji')
plt.xlabel('Epoki')
plt.ylabel('MAE')
plt.legend()
plt.show()
```

Rysunek 4.19 przedstawia wykres.



Rysunek 4.19. Wykres średniego błędu bezwzględnego podczas treningu i walidacji

Kup ksi k Pole ksi k

Świetne wyniki! Możemy z tych wykresów wyczytać dwie fascynujące informacje:

- Ogólnie wskaźniki są znacznie lepsze dla walidacji niż dla treningu, co oznacza, że sieć się nie przetrenowuje.
- W sumie wartości straty i średniego błędu bezwzględnego są mniejsze.

Możesz się zastanawiać, dlaczego wskaźniki dla walidacji są, ogólnie rzecz biorąc, lepsze niż te dla treningu, a nie takie same. Wynika to z faktu, że wskaźniki dla walidacji są obliczane na końcu każdej epoki, podczas gdy wskaźniki dla treningu są obliczane w jego trakcie. Oznacza to, że walidacja następuje na modelu, który był trenowany nieco dłużej.

Nasz model wydaje się działać bardzo dobrze na podstawie naszych danych walidacyjnych. Jednak aby się upewnić, musimy przeprowadzić ostatni test.

Test

Wcześniej odłożyliśmy 20% naszych danych na potrzeby testu. Jak już mówiliśmy, ważne jest, aby oddzielić dane walidacyjne od testowych. Jako że ulepszaliśmy naszą sieć na podstawie jej działania na danych walidacyjnych, istnieje ryzyko, że przypadkowo przetrenowaliśmy zestaw walidacyjny i model nie będzie w stanie skutecznie przewidywać na podstawie nowych danych. Przez zachowanie świeżych danych i użycie ich do ostatniego testu naszego modelu możemy się upewnić, że tak się nie stało.

Po wykorzystaniu naszych danych testowych musimy powstrzymać chęć ponownego ich użycia do trenowania i ulepszania modelu. Jeśli dokonalibyśmy zmian, które miałyby na celu ulepszenie działania modelu na danych testowych, moglibyśmy je przetrenować. W takim przypadku nie bylibyśmy tego świadomi, gdyż nie mielibyśmy już żadnych niewykorzystanych danych, z którymi moglibyśmy przeprowadzić test.

Oznacza to, że jeśli nasz model działa źle na naszych danych testowych, musimy wrócić do etapu projektowania. Musimy przestać ulepszać model i zastosować całkiem nową architekturę.

Mając to na uwadze, następująca komórka oceni nasz model na podstawie danych testowych:

```
# Obliczenie i wyświetlenie straty naszych danych testowych
loss = model_2.evaluate(x_test, y_test)

# Prognozowanie na podstawie naszych danych testowych
predictions = model_2.predict(x_test)

# Wykres prognoz w porównaniu z rzeczywistymi wartościami
plt.clf()
plt.title('Porównanie prognozy z rzeczywistością')
plt.plot(x_test, y_test, 'b.', label='Rzeczywistość')
plt.plot(x_test, predictions, 'r.', label='Prognoza')
plt.legend()
plt.show()
```

Najpierw wywołujemy metodę `evaluate()` z danymi testowymi, która obliczy wskaźnik straty i średniego błędu bezwzględnego. Dzięki temu będziemy wiedzieć, jak bardzo przewidziane przez model wartości odbiegają od rzeczywistości. Następnie przygotowujemy zestaw prognoz i umieszczamy je na wykresie w zestawieniu z rzeczywistymi wartościami.

Kup ksi k Pole ksi k

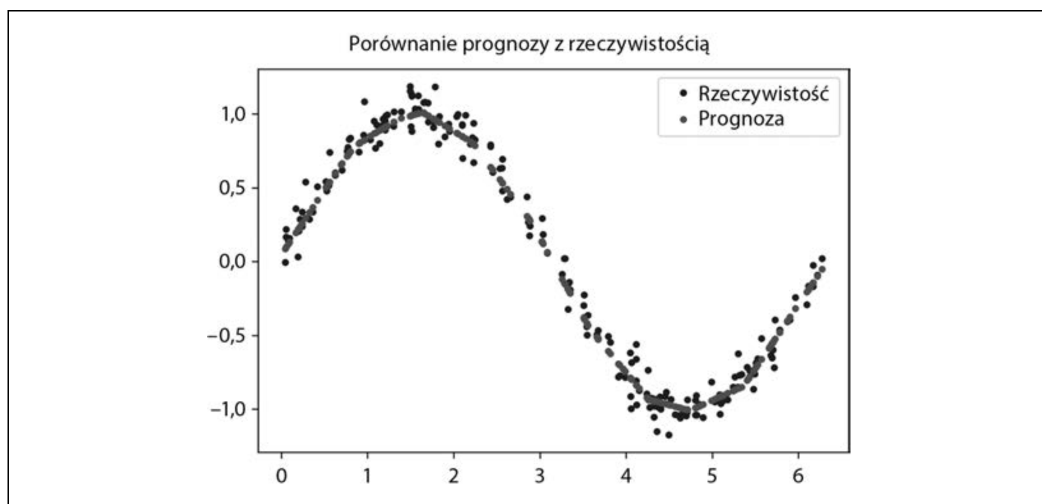
Teraz możemy uruchomić komórkę, by zobaczyć, jak skutecznie nasz model działa! Najpierw sprawdźmy wyniki metody `evaluate()`.

```
200/200 [==============================] - 0s 71us/sample - loss: 0.0103 - mae: 0.0718
```

Widzimy, że zostało ocenionych 200 grup danych, czyli cały zestaw testowy. Każda prognoza zajęła modelowi 71 mikrosekund. Strata wynosiła 0,0103, co jest bardzo dobrym wynikiem i wartością zbliżoną do straty walidacji, wynoszącej 0,0104. Nasz średni błąd bezwzględny, 0,0718, również jest bardzo mały i całkiem zbliżony do swojego odpowiednika uzyskanego na danych walidacyjnych — 0,0806.

Oznacza to, że nasz model działa całkiem sprawnie i się nie przetrenowuje! Jeśli nasz model przetrenowałby dane walidacyjne, moglibyśmy się spodziewać, że wskaźniki naszego zestawu testowego byłyby znacząco gorsze od tych wynikających z walidacji.

Wykres naszych prognoz w porównaniu z wartościami rzeczywistymi, widoczny na rysunku 4.20, wyraźnie pokazuje, jak dobrze nasz model działa.



Rysunek 4.20. Wykres przewidzianych wartości w zestawieniu z rzeczywistymi wartościami dla naszych danych testowych

Widzisz, że w większości kropki przedstawiające *prognozy* tworzą krzywą bez zakłóceń wzdłuż krzywej utworzonej z *rzeczywistych* wartości. Nasza sieć nauczyła się przewidywać sinusoidę, nawet pomimo tego, że dane nie były uporządkowane!

Jeśli jednak przyjrzysz się bliżej, zauważysz pewne niedoskonałości. Szczyt i dołek naszej przewidzianej sinusoidy nie są idealnie gładkie, jak prawdziwa fala sinusoidalna. Nasz model nauczył się zmian w naszych danych treningowych, które są rozłożone losowo. Jest to łagodne przetrenowanie, gdyż zamiast nauczyć się gładkiej funkcji sinus, model nauczył się powielać dokładny kształt danych.

W kontekście naszych potrzeb przetrenowanie nie jest głównym problemem. Naszym celem jest, by model delikatnie rozświetlał oraz wygaszał diodę LED, i nie musi być idealnie płynny, by to osiągnąć.

Kup ksi k Pole ksi k

Jeżeli poziom przetrenowania byłby dla nas problemem, moglibyśmy go rozwiązać przez techniki regularyzacji lub przez uzyskanie większej liczby danych treningowych.

Teraz, gdy jesteśmy już zadowoleni z naszego modelu, przygotujmy go do uruchomienia na docelowym urządzeniu!

Konwertowanie modelu na potrzeby TensorFlow Lite

Na początku tego rozdziału wspomnieliśmy o TensorFlow Lite, który jest zestawem narzędzi do uruchamiania modeli TensorFlow na małych urządzeniach, od telefonów komórkowych po płytki z mikrokontrolerami.

Rozdział 13. szczegółowo omawia TensorFlow Lite dla mikrokontrolerów. Teraz jest ważne, by wiedzieć, że ten program ma dwie główne części:

Konwerter TensorFlow Lite

Zamienia modele TensorFlow na specjalny format zajmujący bardzo mało miejsca w pamięci, by można było go używać na urządzeniach, w których ten zasób jest ograniczony. Co więcej, program ten może zastosować optymalizacje, które jeszcze bardziej zmniejszą rozmiar modelu i sprawią, że będzie działał szybciej na małych urządzeniach.

Interpreter TensorFlow Lite

Uruchamia odpowiednio przekonwertowany model TensorFlow Lite za pomocą najbardziej wydajnych dla danego urządzenia operacji.

Zanim uruchomimy nasz model za pomocą TensorFlow Lite, musimy go odpowiednio przetworzyć. W tym celu użyjemy API konwertera TensorFlow Lite w Pythonie, który bierze nasz model stworzony za pomocą Keras i zapisuje go na dysku w specjalnym formacie *FlatBuffer*, zaprojektowanym w ten sposób, by w wykorzystaniu pamięci był tak wydajny, jak to tylko możliwe. Ponieważ nasz model będziemy wdrażać na urządzeniu z ograniczoną pamięcią, taki format będzie bardzo pomocny! W rozdziale 12. przyjrzymy się bliżej formatowi FlatBuffer.

Poza przekonwertowaniem modelu na wspomniany właśnie format konwerter TensorFlow Lite może również zoptymalizować model, dzięki czemu ten będzie jeszcze mniejszy lub będzie szybciej działał, albo jedno i drugie. Może to się wiązać z mniejszą dokładnością, ale zazwyczaj różnica jest tak mała, że nadal warto. Więcej o optymalizacji możesz przeczytać w rozdziale 13.

Jedną z najbardziej przydatnych technik optymalizacji jest **kwantyzacja**. W modelu wagi i przesunięcia są domyślnie zapisywane jako 32-bitowe liczby zmiennoprzecinkowe, tak by podczas treningu możliwe były obliczenia wymagające dużej dokładności. Kwantyzacja pozwala na zmniejszenie dokładności tych liczb i ich zamianę na 8-bitowe liczby całkowite — czterokrotne zmniejszenie rozmiaru. Ponadto, ponieważ dla procesora obliczenia na liczbach całkowitych są łatwiejsze niż na liczbach zmiennoprzecinkowych, model po kwantyzacji będzie działał szybciej.

Największą zaletą kwantyzacji jest minimalna strata na dokładności. Innymi słowy, gdy chcemy użyć naszego modelu na urządzeniach z małą pamięcią, niemal zawsze warto ją zastosować.

Kup ksi k   Pole ksi k

W następnej komórce używamy konwertera, by stworzyć i zapisać dwie nowe wersje naszego modelu: pierwszy przekonwertowany na format FlatBuffer programu TensorFlow Lite, przed optymalizacjami, a drugi po zastosowaniu kwantyzacji.

Uruchom następującą komórkę, aby przekonwertować model na te dwie wersje:

```
# Zamiana modelu na format TensorFlow Lite bez kwantyzacji
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
tflite_model = converter.convert()

# Zapisanie modelu na dysku
open("sine_model.tflite", "wb").write(tflite_model)

# Zamiana modelu na format TensorFlow Lite z kwantyzacją
converter = tf.lite.TFLiteConverter.from_keras_model(model_2)
# Optymalizacja z zastosowaniem domyślnych optymalizacji, które zawierają kwantyzację
converter.optimizations = [tf.lite.Optimize.DEFAULT]
# Definicja funkcji generującej, która dostarcza wartości x naszych danych testowych jako reprezentacyjnego zestawu danych,
# i informacja dla konwertera, że ma ich użyć
def representative_dataset_generator():
  for value in x_test:
    # Każda wartość skalarna musi być umieszczona w tablicy 2D, która jest spakowana w listę.
    yield [np.array(value, dtype=np.float32, ndmin=2)]
converter.representative_dataset = representative_dataset_generator
# Konwersja modelu
tflite_model = converter.convert()

# Zapisanie modelu na dysku
open("sine_model_quantized.tflite", "wb").write(tflite_model)
```

W celu zastosowania kwantyzacji dla modelu, by działał tak efektywnie, jak to możliwe, musimy zapewnić *reprezentacyjny zestaw danych* — zestaw liczb z pełnego zakresu wartości wejściowych, z którymi model był trenowany.

W poprzedniej komórce możemy jako danych reprezentacyjnych użyć naszych wartości *x* z danych testowych. Definiujemy funkcję `representative_dataset_generator()`, która używa operatora `yield`, by zwracać je jedna po drugiej.

Aby udowodnić, że po konwersji i kwantyzacji modele nadal są dokładne, zastosujemy te obie techniki do prognozowania, a następnie porównamy otrzymane rezultaty z wynikami testowymi. Ze względu na to, że są to modele w formacie TensorFlow Lite, musimy uruchomić model za pomocą interpretera TensorFlow Lite.

Ponieważ wspomniany właśnie interpreter został zaprojektowany głównie ze względu na wydajność, jest trochę bardziej skomplikowany w użyciu niż API Keras. Prognozowanie z naszym modelem Keras wymagało jedynie wywołania metody `predict()`, do której przekazywaliśmy tablicę danych wejściowych. W przypadku TensorFlow Lite musimy wykonać następujące kroki:

1. Utworzenie instancji obiektu `Interpreter`.
2. Wywołanie kilku metod, które przydzielają pamięć dla modelu.
3. Zapisanie danych wejściowych w tensorze wejściowym.
4. Wywołanie metody.
5. Odczytanie danych wyjściowych z tensora otrzymanego na wyjściu.

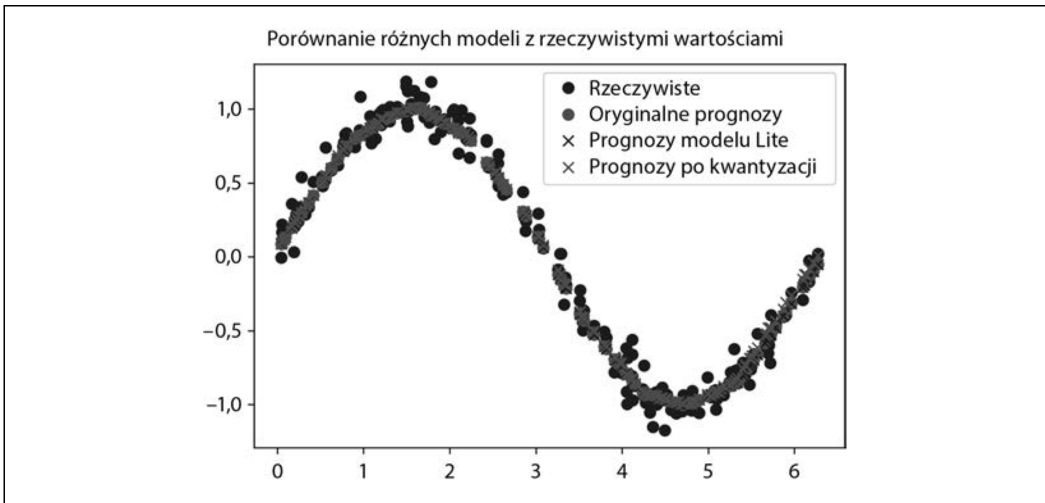
Wydaje się, że jest dużo do zrobienia, ale nie martw się tym za bardzo na tym etapie, omówimy ten proces w szczegółach w rozdziale 5. Teraz uruchomimy poniżej pokazaną komórkę, by oba modele przygotowały prognozy, które przedstawimy na wykresie w zestawieniu z wynikami z naszego pierwotnego, nieprzetworzonego modelu.

```
# Utworzenie instancji interpretera dla każdego modelu
sine_model = tf.lite.Interpreter('sine_model.tflite')
sine_model_quantized = tf.lite.Interpreter('sine_model_quantized.tflite')

# Przypisanie pamięci dla każdego modelu
sine_model.allocate_tensors()
sine_model_quantized.allocate_tensors()

# Pobranie indeksów tensorów wejściowych i wyjściowych
sine_model_input_index = sine_model.get_input_details()[0]["index"]
sine_model_output_index = sine_model.get_output_details()[0]["index"]
sine_model_quantized_input_index = sine_model_quantized.get_input_details()[0]["index"]
sine_model_quantized_output_index = \
  sine_model_quantized.get_output_details()[0]["index"]

# Utworzenie tablic do przechowywania wyników
sine_model_predictions = []
sine_model_quantized_predictions = []

# Uruchomienie interpretera każdego modelu dla każdej wartości i zapisanie wyników w tablicach
for x_value in x_test:
  # Utworzenie tensora 2D z bieżącą wartością x
  x_value_tensor = tf.convert_to_tensor([[x_value]], dtype=np.float32)
  # Zapisanie wartości w tensorze wejściowym
  sine_model.set_tensor(sine_model_input_index, x_value_tensor)
  # Uruchomienie procesu wnioskowania
  sine_model.invoke()
  # Odczyt prognozy z tensora wyjściowego
  sine_model_predictions.append(sine_model.get_tensor(sine_model_output_index)[0])
  # Te same kroki dla modelu po kwantyzacji
  sine_model_quantized.set_tensor(sine_model_quantized_input_index, x_value_tensor)
  sine_model_quantized.invoke()
  sine_model_quantized_predictions.append(\
      sine_model_quantized.get_tensor(sine_model_quantized_output_index)[0])

# Sprawdzenie, jak prognozy pasują do danych
plt.clf()
plt.title('Porównanie różnych modeli z rzeczywistymi wartościami')
plt.plot(x_test, y_test, 'bo', label='Rzeczywiste')
plt.plot(x_test, predictions, 'ro', label='Oryginalne prognozy')
plt.plot(x_test, sine_model_predictions, 'bx', label='Prognozy modelu Lite')
plt.plot(x_test, sine_model_quantized_predictions, 'gx', \
  label='Prognozy po kwantyzacji')
plt.legend()
plt.show()
```

W rezultacie uruchomienia tej komórki otrzymamy wykres pokazany na rysunku 4.21.

Z wykresu wynika, że prognozy oryginalnego modelu, przekonwertowanego modelu i modelu po kwantyzacji są wystarczająco blisko, by różnica między nimi była niedostrzegalna. Wygląda to dobrze!

Kup ksi k Pole ksi k



Rysunek 4.21. Wykres porównujący prognozy modelu z wartościami rzeczywistymi

Skoro kwantyzacja zmniejsza rozmiar modelu, porównajmy oba przekonwertowane modele, by zobaczyć różnicę w rozmiarze. Uruchom następującą komórkę, by obliczyć rozmiar obu modeli i je porównać.

```
import os
basic_model_size = os.path.getsize("sine_model.tflite")
print("Podstawowy model ma %d bajtów" % basic_model_size)
quantized_model_size = os.path.getsize("sine_model_quantized.tflite")
print("Po kwantyzacji model ma %d bajtów" % quantized_model_size)
difference = basic_model_size - quantized_model_size
print("Różnica wynosi %d bajtów" % difference)
```

Na wyjściu powinieneś zobaczyć następujące informacje:

```
Podstawowy model ma 2736 bajtów
Po kwantyzacji model ma 2512 bajtów
Różnica wynosi 224 bajtów
```

Model po kwantyzacji jest mniejszy o 224 bajty od podstawowego modelu TensorFlow Lite, co jest dobrą wiadomością, ale jest to tylko niewielka różnica w rozmiarze. Rozmiar 2,4 kB jest już na tyle mały, że wagi i przesunięcia są tylko ułamkiem całkowitego rozmiaru. Poza wagami model zawiera całą logikę, która tworzy architekturę naszej sieci uczenia głębokiego, znanej jako **graf obliczeniowy**. W przypadku naprawdę małych modeli to może wpływać głównie na rozmiar, a nie wagi modelu, tym samym kwantyzacja przynosi niewielki efekt.

Bardziej złożone modele mają więcej wag, co oznacza, że kwantyzacja oszczędzi znacznie więcej miejsca. Bardziej złożone modele mogą być nawet cztery razy mniejsze.

Bez względu na rzeczywisty rozmiar nasze modele po kwantyzacji będą działać szybciej niż podstawowe modele, co jest istotne, gdy w grę wchodzą malutkie mikrokontrolery.

Kup ksi k   Pole ksi k

Konwertowanie na plik C

Ostatni krok w przygotowaniu naszego modelu do użycia z TensorFlow Lite dla mikrokontrolerów polega na przekonwertowaniu go na plik źródłowy C, który może być włączony do naszej aplikacji.

Do tej pory w tym rozdziale używaliśmy API TensorFlow Lite dla Pythona. Dzięki temu mogliśmy używać konstruktora klasy `Interpreter`, by wgrać z dysku pliki naszego modelu.

Jednak większość mikrokontrolerów nie ma systemu plików, a nawet jakby miały, dodatkowy kod wymagany do wgrania modelu z dysku byłby rozrzutnością, biorąc pod uwagę ograniczone miejsce w pamięci. W zamian stosujemy eleganckie rozwiązanie — dostarczamy model w postaci pliku źródłowego C, który może być bezpośrednio wgrany do pamięci.

W tym pliku model jest zdefiniowany jako tablica bajtów. Na szczęście istnieje wygodne narzędzie uniksowe o nazwie *xxd*, które jest w stanie przekonwertować dany plik na wymagany format.

Następująca komórka uruchamia *xxd* na naszym modelu po kwantyzacji, zapisuje plik wyjściowy pod nazwą *sine_model_quantized.cc* i wyświetla jego zawartość na ekranie.

```
# Instalacja narzędzia xxd, jeśli nie jest dostępne
!apt-get -qq install xxd
# Zapisanie modelu jako pliku źródłowego C
!xxd -i sine_model_quantized.tflite > sine_model_quantized.cc
# Wyświetlenie pliku źródłowego
!cat sine_model_quantized.cc
```

Plik wyjściowy jest bardzo długi, więc nie będziemy go tutaj przytaczać w całości, ale oto wycinek zawierający początek i koniec pliku:

```
unsigned char sine_model_quantized_tflite[] = {
  0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x00, 0x00, 0x12, 0x00,
  0x1c, 0x00, 0x04, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x10, 0x00, 0x14, 0x00,
  // ...
  0x00, 0x00, 0x08, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x09,
  0x04, 0x00, 0x00, 0x00
};
unsigned int sine_model_quantized_tflite_len = 2512;
```

Aby wykorzystać ten model w projekcie, mógłbyś albo skopiować i wkleić źródło, albo pobrać plik z notatnika.

Podsumowanie

I na tym zakończyliśmy budowę naszego modelu. Wytrenowaliśmy, oceniliśmy i przekonwertowaliśmy sieć uczenia głębokiego TensorFlow, która może pobierać liczby z zakresu od 0 do 2π i zapewniać wystarczająco dobre przybliżenie ich sinusa.

Było to nasze pierwsze użycie interfejsu Keras do wytrenowania niewielkiego modelu. W kolejnych projektach będziemy trenować modele, które są wciąż malutkie, ale są *o wiele bardziej* zaawansowane.

Na razie przejdźmy do rozdziału 5., gdzie napiszemy kod w celu uruchomienia naszego modelu na mikrokontrolerach.

Kup ksi k Pole ksi k

Kup ksi k

Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

Ograniczone zasoby? Poznaj TinyML!

Może się wydawać, że profesjonalne systemy uczenia maszynowego wymagają sporych zasobów mocy obliczeniowej i energii. Okazuje się, że niekoniecznie: można tworzyć zaawansowane, oparte na sieciach neuronowych aplikacje, które doskonale poradzą sobie bez potężnych procesorów. Owszem, praca na mikrokontrolerach podobnych do Arduino lub systemach wbudowanych wymaga pewnego przygotowania i odpowiedniego podejścia, jest to jednak fascynujący sposób na wykorzystanie niewielkich urządzeń o niskim zapotrzebowaniu na energię do tworzenia zdumiewających projektów.

Ta książka jest przystępnym wprowadzeniem do skomplikowanego świata, w którym za pomocą techniki TinyML wdraża się głębokie uczenie maszynowe w systemach wbudowanych. Nie musisz mieć żadnego doświadczenia z zakresu uczenia maszynowego czy pracy z mikrokontrolerami. W książce wyjaśniono, jak można trenować modele na tyle małe, by mogły działać w każdym środowisku — również Arduino. Dokładnie opisano sposoby użycia techniki TinyML w tworzeniu systemów wbudowanych opartych na zastosowaniu uczenia maszynowego. Zaprezentowano też kilka ciekawych projektów, na przykład dotyczący budowy urządzenia rozpoznającego mowę, magicznej różdżki reagującej na gesty, a także rozszerzenia możliwości kamery o wykrywanie ludzi.

W książce między innymi:

- praca z Arduino i innymi mikrokontrolerami o niskim poborze mocy
- podstawy uczenia maszynowego, budowy i treningu modeli
- TensorFlow Lite i zestaw narzędzi Google dla TinyML
- bezpieczeństwo i ochrona prywatności w aplikacji
- optymalizacja modelu
- tworzenie modeli do interpretacji różnego rodzaju danych

Pete Warden jest współzałożycielem zespołu do spraw TensorFlow. Obecnie zajmuje się platformą TensorFlow dla mobilnych systemów operacyjnych i systemów wbudowanych. Był założycielem firmy Jetpac, przejętej przez Google w 2014 roku.

Daniel Situnayake wspiera programistów TensorFlow w Google. Jest współzałożycielem firmy Tiny Farms, która jako pierwsza w Stanach Zjednoczonych zautomatyzowała proces uzyskiwania białka z owadów na skalę przemysłową.

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
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8362-3

9 788328 383623

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 99,00 zł