Valentino Zocca, Gianmario Spacagna,
Daniel Slater, Peter Roelants

Deep Learning

Uczenie głębokie z językiem Python

Sztuczna inteligencja
i sieci neuronowe

Packt>

Tytuł oryginału: Python Deep Learning

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-4173-9

Copyright © Packt Publishing 2017.

First published in the English language under the title ‘Python Deep Learning
- (9781786464453)’

Polish edition copyright © 2018 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 Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach 9

O recenzencie 11

Przedmowa 13

Co zawiera książka? 13
Co jest potrzebne podczas lektury tej książki? 14
Dla kogo jest ta książka? 15
Konwencje 15
Pobieranie przykładowego kodu 16
Pobieranie kolorowych ilustracji do tej książki 16

Rozdział 1. Uczenie maszynowe — wprowadzenie 17

Czym jest uczenie maszynowe? 18
Różne podejścia do uczenia maszynowego 19
Uczenie nadzorowane 19
Uczenie nienadzorowane 22
Uczenie przez wzmacnianie 23
Fazy systemów uczenia maszynowego 24
Krótki opis popularnych technik (algorytmów) 28
Zastosowania praktyczne 40
Popularny pakiet open source 42
Podsumowanie 48

Rozdział 2. Sieci neuronowe 49

Dlaczego sieci neuronowe? 50
Podstawy 51
Neurony i warstwy 52
Różne rodzaje funkcji aktywacji 56
Algorytm propagacji wstecznej 61
Zastosowania praktyczne 68
Przykład kodu sieci neuronowej dla funkcji XOR 70
Podsumowanie 75

Kup ksi k   Pole ksi k

Rozdział 3. Podstawy uczenia głębokiego 77

Czym jest uczenie głębokie? 78
Podstawowe pojęcia 80
Uczenie się cech 81
Algorytmy uczenia głębokiego 88
Zastosowania uczenia głębokiego 89
Rozpoznawanie mowy 90
Rozpoznawanie i klasyfikacja obiektów 91
GPU kontra CPU 94
Popularne biblioteki open source — wprowadzenie 96
Theano 96
TensorFlow 97
Keras 97
Przykład implementacji głębokiej sieci neuronowej za pomocą biblioteki Keras 98
Podsumowanie 102

Rozdział 4. Nienadzorowane uczenie cech 105

Autoenkodery 107
Projekt sieci 110
Metody regularyzacji dla autoenkoderów 113
Autoenkodery — podsumowanie 117
Ograniczone maszyny Boltzmanna 119
Sieci Hopfielda a maszyny Boltzmanna 121
Maszyna Boltzmanna 123
Ograniczona maszyna Boltzmanna 125
Implementacja za pomocą biblioteki TensorFlow 126
Sieci DBN 130
Podsumowanie 132

Rozdział 5. Rozpoznawanie obrazów 135

Podobieństwa pomiędzy modelami sztucznymi a biologicznymi 136
Intuicja i uzasadnianie 137
Warstwy konwolucyjne 138
Parametry krok i wypełnienie w warstwach konwolucyjnych 144
Warstwy pooling 145
Dropout 147
Warstwy konwolucyjne w uczeniu głębokim 147
Warstwy konwolucyjne w bibliotece Theano 148
Przykład zastosowania warstwy konwolucyjnej do rozpoznawania cyfr za pomocą biblioteki Keras 150
Przykład zastosowania warstwy konwolucyjnej za pomocą biblioteki Keras dla zbioru danych CIFAR10 153
Szkolenie wstępne 155
Podsumowanie 156

Kup ksi k

Pole ksi k

Rozdział 6. Rekurencyjne sieci neuronowe i modele języka 159

Rekurencyjne sieci neuronowe 160
RNN — jak implementować i trenować? 162
Długa pamięć krótkotrwała 168
Modelowanie języka 171
Modele na bazie słów 171
Modele bazujące na znakach 176
Rozpoznawanie mowy 183
Potok rozpoznawania mowy 183
Mowa jako dane wejściowe 184
Przetwarzanie wstępne 185
Model akustyczny 186
Dekodowanie 189
Modele od końca do końca 190
Podsumowanie 190
Bibliografia 190

Rozdział 7. Uczenie głębokie w grach planszowych 195

Pierwsze systemy AI grające w gry 197
Wykorzystanie algorytmu min-max do oceny stanów gry 198
Implementacja gry w kółko i krzyżyk w Pythonie 201
Uczenie funkcji wartości 209
Trenowanie systemu AI do uzyskania mistrzostwa w grze w Go 210
Zastosowanie górnych granic zaufania do drzew 213
Uczenie głębokie w algorytmie przeszukiwania drzewa Monte Carlo 220
Krótkie przypomnienie technik uczenia przez wzmacnianie 222
Metoda policy gradients w funkcjach strategii uczenia 222
Metoda policy gradients w AlphaGo 230
Podsumowanie 232

Rozdział 8. Uczenie głębokie w grach komputerowych 235

Techniki uczenia nadzorowanego w odniesieniu do gier 235
Zastosowanie algorytmów genetycznych do grania w gry komputerowe 237
Q-learning 238
Funkcja Q 240
Q-learning w akcji 241
Gry dynamiczne 246
Odtwarzanie doświadczeń 250
Epsilon zachłanny 253
Breakout na Atari 254
Losowy test gry w Breakout na Atari 255
Wstępne przetwarzanie ekranu 257
Tworzenie głębokiej sieci konwolucyjnej 259
Problemy zbieżności w technikach Q-learning 263
Technika policy gradients kontra Q-learning 265

Kup ksi k Pole ksi k

Metody aktor-krytyk 266
Metoda baseline do redukcji wariancji 267
Uogólniony estymator korzyści 267
Metody asynchroniczne 268
Podejścia bazujące na modelach 269
Podsumowanie 272

Rozdział 9. Wykrywanie anomalii 273

Co to jest wykrywanie anomalii i wykrywanie elementów odstających? 274
Rzeczywiste zastosowania mechanizmów wykrywania anomalii 277
Popularne płytkie techniki uczenia maszynowego 278
Modelowanie danych 279
Modelowanie wykrywania 279
Wykrywanie anomalii z wykorzystaniem głębokich autoenkoderów 281
H2O 283
Wprowadzenie do pracy z H2O 285
Przykłady 285
Rozpoznawanie anomalii wykrywania cyfr z wykorzystaniem zestawu danych MNIST 286
Podsumowanie 298

Rozdział 10. Budowanie gotowego do produkcji systemu wykrywania włamań 301

Czym jest produkt danych? 302
Trening 304
Inicjalizacja wag 304
Współbieżny algorytm SGD z wykorzystaniem techniki HOGWILD! 306
Uczenie adaptacyjne 308
Uczenie rozproszone z wykorzystaniem mechanizmu MapReduce 314
Sparkling Water 317
Testowanie 320
Walidacja modelu 326
Dostrajanie hiperparametrów 335
Ocena od końca do końca 338
Podsumowanie zagadnień związanych z testowaniem 342
Wdrażanie 343
Eksport modelu do formatu POJO 344
Interfejsy API oceny anomalii 347
Podsumowanie wdrażania 349
Podsumowanie 350

Skorowidz 351

Kup ksi k

Pole ksi k

Sieci neuronowe

W poprzednim rozdziale opisaliśmy kilka algorytmów uczenia maszynowego i zaprezentowaliśmy różne techniki analizy danych stosowane w celu dokonywania prognoz. Pokazaliśmy na przykład, w jaki sposób maszyny mogą skorzystać z informacji o cenach sprzedaży domów w celu prognozowania cen nowych nieruchomości. Opisaliśmy, w jaki sposób techniki uczenia maszynowego są stosowane w dużych firmach, takich jak Netflix, w celu sugerowania użytkownikom nowych filmów na podstawie tych, które podobały im się wcześniej. Użyto do tego metody powszechnie stosowanej w branży e-commerce przez takich gigantów jak Amazon czy Walmart. Jednak większość z tych metod do prognozowania nowych danych wymagała danych oznakowanych etykietami, a żeby zwiększyć ich skuteczność, ludzie musieli opisać dane za pomocą sensownych cech.

Ludzie są zdolni do szybkiej ekstrapolacji trendów i wnioskowania reguł bez konieczności specjalnego oczyszczania i przygotowania dla nich danych. Byłoby dobrze, gdyby maszyny mogły nauczyć się postępować w taki sam sposób. Jak wspominaliśmy, Frank Rosenblatt wynalazł perceptron w 1957 roku, czyli ponad sześćdziesiąt lat temu. W porównaniu z nowoczesnymi głębokimi sieciami neuronowymi perceptron przypomina organizm jednokomórkowy w zestawieniu ze złożonymi wielokomórkowymi formami życia. Pomimo tego zrozumienie sposobu działania sztucznego neuronu i zapoznanie się z nim jest bardzo ważne. Jest także niezbędne dla lepszego zrozumienia i docenienia złożoności, jaką można uzyskać przez pogrupowanie wielu neuronów w wielu warstwach w celu stworzenia głębokich sieci neuronowych.

Sieci neuronowe to próba zasymulowania pracy ludzkiego mózgu i jego zdolności do wnioskowania nowych reguł w wyniku prostych obserwacji. Choć jesteśmy jeszcze daleko od zrozumienia sposobu, w jaki ludzki mózg porządkuje i przetwarza informacje, to dość dobrze rozumiemy już, jak działają pojedyncze ludzkie neurony.

Sztuczne sieci neuronowe starają się naśladować tę samą funkcjonalność, choć zastąpiono w nich komunikaty chemiczne i elektryczne wartościami i funkcjami liczbowymi. Znaczny postęp osiągnięto w ostatnim dziesięcioleciu, po tym, jak co najmniej dwadzieścia lat wcześniej sieci

Kup ksi k
Pole ksi k

neuronowe zyskały bardzo dużą popularność, a następnie o nich zapomniano. Ponowne zainteresowanie wynika po części z dostępności komputerów, które stają się coraz szybsze, korzystania z procesorów graficznych (*Graphical Processing Unit* — **GPU**) zamiast bardziej tradycyjnych układów **CPU** (*Computing Processing Unit*), lepszych algorytmów i doskonalszego projektu sieci neuronowych, a także coraz bardziej obszernych zbiorów danych. O tym wszystkim opowiemy w niniejszej książce.

Ten rozdział będzie formalnym wprowadzeniem w tematykę sieci neuronowych. Dokładnie opiszemy, jak działa neuron, i zaprezentujemy sposób zestawienia wielu warstw w celu utworzenia i wykorzystania głębokich sieci neuronowych bez sprzężenia zwrotnego.

Dlaczego sieci neuronowe?

Sieci neuronowe są znane od wielu lat. W ich historii można wyróżnić kilka okresów, podczas których popadały w niełaskę i na nowo się odradzały. Jednak w ostatnich latach stopniowo zyskiwały coraz większe znaczenie i przewagę nad wieloma konkurencyjnymi algorytmami uczenia maszynowego. Powodem tego stanu jest to, że architektura zaawansowanych sieci neuronowych udowodniła przydatność do wielu zadań oraz znacznie lepszą dokładność w porównaniu z innymi algorytmami.

Na przykład w dziedzinie rozpoznawania obrazów dokładność może być mierzona na podstawie porównania z bazą danych szesnastu milionów zdjęć, znaną pod nazwą ImageNet.

Przed wprowadzeniem głębokich sieci neuronowych dokładność poprawiała się powoli, a po ich wprowadzeniu stopa błędu spadła z 40% w 2010 roku do mniej niż 7% w roku 2014 i nadal spada. Stopa błędu przy rozpoznawaniu obrazów przez człowieka jest nadal niższa, ale tylko o mniej więcej 5%. Ze względu na sukces głębokich sieci neuronowych wszyscy uczestnicy współzawodnictwa ImageNet w 2013 roku zastosowali jakąś formę głębokiej sieci neuronowej.

Ponadto głębokie sieci neuronowe „uczą się" reprezentacji danych — tzn. nie tylko uczą się rozpoznawania obiektów, ale także tego, jakie są istotne cechy, które w unikatowy sposób definiują zidentyfikowany obiekt. Dzięki nauczeniu się automatycznego identyfikowania cech, głębokie sieci neuronowe mogą być z powodzeniem wykorzystywane do uczenia nienadzorowanego. Za ich pomocą można w naturalny sposób klasyfikować obiekty o podobnych cechach bez konieczności żmudnego znakowania przez ludzi. Podobny postęp osiągnięto również w innych dziedzinach, takich jak przetwarzanie sygnałów. Uczenie głębokie i głębokie sieci neuronowe są obecnie używane powszechnie, na przykład w systemie Siri firmy Apple. Kiedy firma Google wprowadziła algorytm uczenia głębokiego w swoim systemie operacyjnym Android, osiągnięto 25-procentowy spadek błędów rozpoznawania słów. Do rozpoznawania obrazów jest wykorzystywany również zbiór danych MNIST, składający się z próbek cyfr pisanych różnymi charakterami pisma ręcznego. Korzystanie z głębokich sieci neuronowych do rozpoznawania cyfr pozwala obecnie osiągnąć dokładność 99,79%, która jest porównywalna z dokładnością człowieka. Ponadto algorytmy bazujące na głębokich sieciach neuronowych są najlepszym

Kup ksi k Pole ksi k

przykładem imitacji działania ludzkiego mózgu przez sztuczną inteligencję. Mimo że jak dotychczas sieci neuronowe są bardzo elementarnym i uproszczonym modelem naszego mózgu, to oferują więcej niż jakikolwiek inny algorytm — namiastkę ludzkiego intelektu. Pozostałą część tej książki poświęcono różnym sieciom neuronowym oraz kilku ich zastosowaniom.

Podstawy

W pierwszym rozdziale mówiliśmy o trzech różnych podejściach do uczenia maszynowego: uczeniu nadzorowanym, uczeniu nienadzorowanym oraz uczeniu przez wzmacnianie. Klasyczne sieci neuronowe są rodzajem uczenia maszynowego nadzorowanego, choć — jak przekonamy się później — popularność uczenia głębokiego wynika z faktu, że nowoczesne głębokie sieci neuronowe mogą być stosowane również w zadaniach uczenia nienadzorowanego. W następnym rozdziale zaprezentujemy podstawowe różnice pomiędzy klasycznymi płytkimi sieciami neuronowymi a sieciami głębokimi. Na razie jednak skupimy się głównie na klasycznych sieciach bez sprzężenia zwrotnego, które działają na zasadach algorytmu nadzorowanego. Nasze pierwsze pytanie brzmi: czym dokładnie jest sieć neuronowa?

Prawdopodobnie najlepszym sposobem interpretacji sieci neuronowej jest opisanie jej jako matematycznego modelu przetwarzania informacji. Takie określeniemoże wydawać się dość niejasne, stanie się jednak znacznie bardziej czytelne w kolejnych rozdziałach. Sieć neuronowa nie jest stałym programem. Jest to raczej model bądź system, który przetwarza informacje lub dane wejściowe podobnie, jak robią to jednostki biologiczne.

Możemy wyróżnić trzy główne cechy sieci neuronowych:

- **Architektura sieci neuronowej:** Opisuje sposób połączeń (ze sprzężeniem zwrotnym, bez sprzężenia zwrotnego, wielo- lub jednowarstwowe i tak dalej) pomiędzy neuronami, liczbę warstw i neuronów w każdej warstwie.
- **Uczenie:** Opisuje proces, który jest powszechnie definiowany jako trening. Może być przeprowadzany z wykorzystaniem różnych technik, takich jak propagacja wsteczna lub trening poziomów energetycznych. Identyfikuje sposób, w jaki określamy wagi pomiędzy neuronami.
- **Funkcja aktywacji:** Określa funkcję określoną na wartości aktywacji, która jest przekazywana do każdego neuronu, i wewnętrzny stan neuronu. Opisuje, jak neuron działa (stochastycznie, liniowo itp.) oraz w jakich warunkach zostanie uaktywniony (wyzwoli się), a także dane wyjściowe, jakie przekaże do sąsiednich neuronów.

Należy zwrócić uwagę, że niektórzy badacze uznają funkcję aktywacji za część architektury. Na początku łatwiej będzie jednak rozdzielić te dwa aspekty. Należy podkreślić, że sztuczne sieci neuronowe tylko w przybliżony sposób reprezentują działanie biologicznego mózgu. Biologiczna sieć neuronowa to model o wiele bardziej skomplikowany. Nie należy jednak się na tym

Kup ksi k Pole ksi k

koncentrować. Sztuczne sieci neuronowe są zdolne do wykonywania wielu przydatnych zadań (o czym przekonamy się wkrótce) — mogą na przykład aproksymować na dowolnym poziomie każdą funkcję przekształcającą dane wejściowe na wyjściowe.

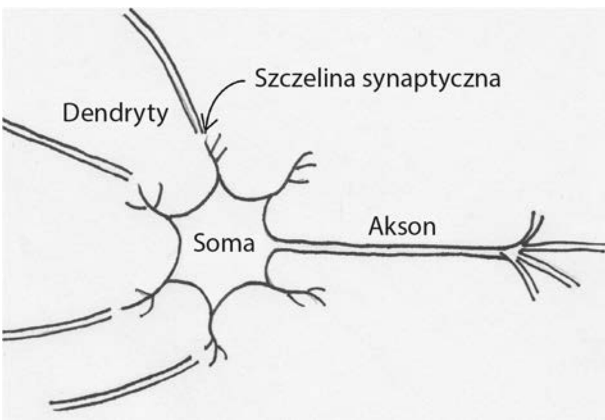
Rozwój sieci neuronowych bazuje na następujących założeniach:

- Przetwarzanie informacji odbywa się, w swojej najprostszej formie, za pośrednictwem prostych elementów zwanych neuronami.
- Neurony są połączone i wymieniają pomiędzy sobą sygnały za pośrednictwem łączy.
- Połączenia pomiędzy neuronami mogą być silniejsze lub słabsze i to decyduje o tym, w jaki sposób są przetwarzane informacje.
- Każdy neuron ma wewnętrzny stan, który jest określony przez wszystkie połączenia przychodzące od innych neuronów.
- Każdy neuron ma inną funkcję aktywacji, która jest obliczana na wewnętrznym stanie neuronu i określa jego sygnał wyjściowy.

W następnym punkcie zdefiniujemy szczegółowo sposób, w jaki działa neuron, oraz to, jak współdziała z innymi neuronami.

Neurony i warstwy

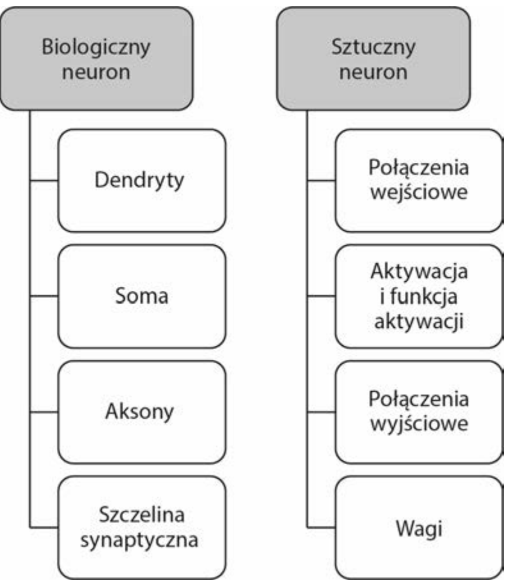
Co to jest neuron? Neuron reprezentuje jednostkę przetwarzania, która przyjmuje wartość wejściową i, zgodnie z określonymi regułami, przekazuje inną wartość na wyjściu.



W 1943 roku Warren McCulloch i Walter Pitts opublikowali artykuł [W.S. McCulloch i W. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, „The Bulletin of Mathematical Biophysics", 5 (4), s. 115–133, 1943], w którym opisali sposób działania pojedynczego biologicznego neuronu. Składnikami biologicznego neuronu są dendryty, soma (ciało komórki), aksony i szczelina synaptyczna. Pod innymi nazwami części te wchodzą również w skład sztucznego neuronu.

Kup ksi k Pole ksi k

Dendryty doprowadzają sygnał wejściowy z innych neuronów do somy — ciała neuronu. Soma to miejsce, w którym dane wejściowe są przetwarzane i sumowane. Jeśli sygnał wejściowy jest powyżej określonego progu, neuron „wyzwala się" i przekazuje pojedynczą daną wyjściową za pomocą sygnałów elektrycznych przez aksony. Pomiędzy aksonami neuronu nadawczego a dendrytami neuronów odbierających znajduje się szczelina synaptyczna, która dopasowuje chemicznie impulsy, dostosowując ich częstotliwości. W sztucznej sieci neuronowej częstotliwość modeluje się za pomocą liczbowej wagi: im wyższa częstotliwość, tym silniejszy impuls, a tym samym wyższa waga. Możemy zatem utworzyć tabele równoważności pomiędzy neuronami biologicznymi i sztucznymi (jest to bardzo uproszczony opis, ale wystarczający do naszych celów):



Schemat porównawczy neuronu biologicznego i sztucznego

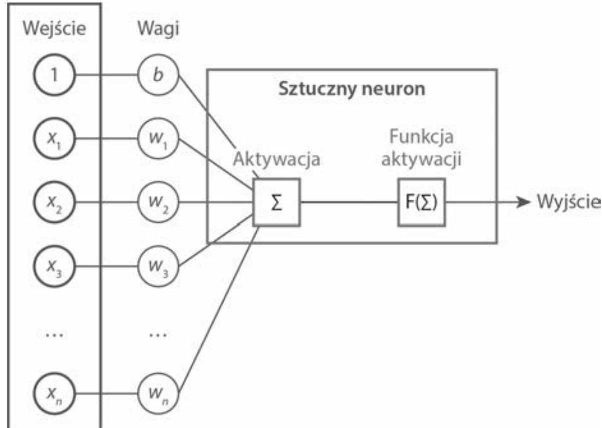
Dlatego możemy schematycznie opisać sztuczny neuron w następujący sposób (pierwszy rysunek na kolejnej stronie).

Prostą wartość aktywacji neuronu można wyrazić wzorem $a(x) = \Sigma_i w_i x_i$, gdzie x_i oznacza wartość każdego neuronu wejściowego, natomiast w_i to wartość połączenia pomiędzy neuronem i a wyjściem. W pierwszym rozdziale, we wprowadzeniu do sieci neuronowych, zaprezentowaliśmy pojęcie przesunięcia (ang. *bias*). Jeśli uwzględnimy przesunięcie i chcemy, aby jego obecność była jawna, możemy przepisać poprzednie równanie do postaci $a(x) = \Sigma_i w_i x_i + b$.

Efektem przesunięcia jest przekształcenie hiperpłaszczyzny zdefiniowanej przez wagi. Dzięki temu nie musi ona przechodzić przez początek układu współrzędnych.

Wartość aktywacji należy interpretować jako wartość wewnętrznego stanu neuronu.

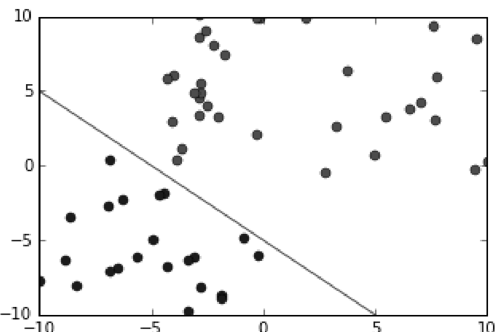
Kup ksi k Pole ksi k



Na środku tej ilustracji pokazano neuron, czyli somę. Otrzymuje ona wejście (aktywację) i ustawia wewnętrzny stan neuronu, który wyzwala wyjście (funkcja aktywacji). Sygnał wejściowy jest przekazywany od innych neuronów, a jego intensywność jest dopasowywana za pomocą wag (szczelina synaptyczna)

Jak wspomniano w poprzednim rozdziale, zdefiniowaną wcześniej wartość aktywacji można zinterpretować jako iloczyn skalarny pomiędzy wektorem w a wektorem x. Wektor x jest prostopadły do wektora wagi w, jeśli $<w,x> = 0$, dlatego wszystkie wektory x takie, że $<w,x> = 0$ definiują hiperpłaszczyznę w $\mathbf{R}^n$ (gdzie n oznacza wymiar wektora x).

Stąd każdy wektor x spełniający warunek $<w,x> >0$ jest wektorem po prawej stronie hiperpłaszczyzny zdefiniowanej przez w. Zatem neuron jest liniowym klasyfikatorem, który zgodnie z tą zasadą uaktywni się, gdy sygnał wejściowy osiągnie wartość powyżej określonego progu lub — geometrycznie — jeśli wejście znajduje się z jednej strony hiperpłaszczyzny zdefiniowanej przez wektor wag.



Pojedynczy neuron jest liniowym klasyfikatorem

Sieć neuronowa może zawierać nieskończoną liczbę neuronów, ale niezależnie od tego w klasycznej sieci wszystkie neurony są ułożone w warstwy. Warstwa wejściowa reprezentuje zbiór danych — warunki początkowe. Na przykład, jeśli wejściem jest obraz w skali szarości, to warstwa wejściowa jest reprezentowana dla każdego piksela przez neuron wejściowy z wewnętrzną

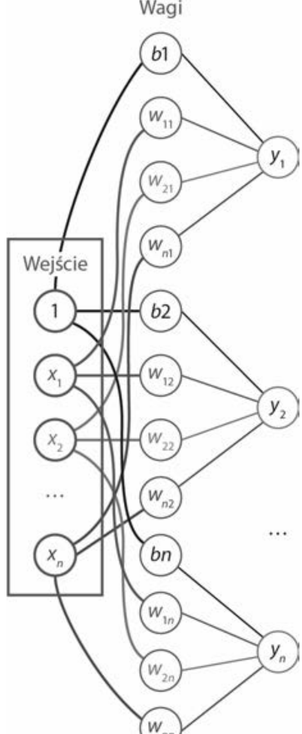
Kup ksi k Pole ksi k

wartością określającą intensywność piksela. Należy jednak zwrócić uwagę, że neurony w warstwie wejściowej nie są takie same jak inne, ponieważ ich wyjście jest stałe i równe wartości ich stanu wewnętrznego. Z tego powodu warstwa wejściowa, ogólnie rzecz biorąc, nie jest brana pod uwagę. Zatem jednowarstwowa sieć neuronowa jest prostą siecią złożoną z tylko jednej (oprócz wejściowej) warstwy — wyjścia. Od każdego neuronu wejściowego rysujemy linię łączącą go ze wszystkimi neuronami wyjściowymi. Wartość neuronu jest dostosowywana przez sztuczną szczelinę synaptyczną, czyli wagę $w_{i,j}$ łączącą wejściowy neuron x_i z neuronem wyjściowym y_j.

Zwykle każdy neuron wyjściowy reprezentuje klasę, na przykład w przypadku zbioru danych MNIST każdy neuron reprezentuje cyfrę.

Dlatego jednowarstwową sieć neuronową można wykorzystać do dokonywania takich prognoz, jak rozpoznawanie, jaką cyfrę przedstawia obraz wejściowy. W istocie zbiór wartości wyjściowych może być traktowany jako miara prawdopodobieństwa tego, że obraz reprezentuje określoną klasę, i dlatego neuron wyjściowy z największą wartością reprezentuje prognozę sieci neuronowej.

Należy zauważyć, że neurony w tej samej warstwie nigdy nie są ze sobą połączone (patrz rysunek poniżej). Wszystkie neurony z jednej warstwy są połączone z neuronami w warstwie następnej i tak dalej:

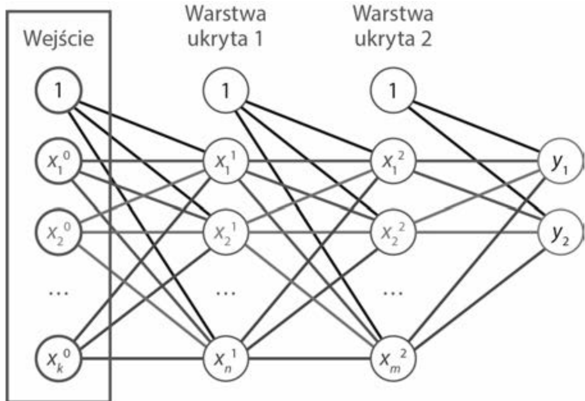


Przykład jednowarstwowej sieci neuronowej: neurony po lewej stronie reprezentują wejście z przesunięciem b, środkowa kolumna przedstawia wagi dla każdego połączenia, natomiast neurony po prawej reprezentują wyjście z uwzględnieniem wag w

Kup książkę Poleć książkę

Jest to jeden z podstawowych i głównych warunków dla klasycznych sieci neuronowych. Nie ma połączeń wewnątrz warstw, natomiast każdy neuron połączony jest z każdym neuronem warstw sąsiednich. Na poprzednim rysunku jawnie pokazaliśmy wagi dla każdego połączenia pomiędzy neuronami, ale zazwyczaj krawędzie łączące neurony pośrednio reprezentują wagi. **1** reprezentuje blok przesunięcia, czyli neuron o wartości *1* z wagą połączenia równą przesunięciu, które wprowadziliśmy wcześniej.

Jak już wspomniano wiele razy, jednowarstwowe sieci neuronowe są zdolne do klasyfikowania tylko tych klas, które dają się rozdzielić liniowo. Nic jednak nie stoi na przeszkodzie, aby pomiędzy wejściem a wyjściem wprowadzić więcej warstw. Te dodatkowe warstwy są nazywane warstwami ukrytymi.



Trójwarstwowa sieć neuronowa z dwoma warstwami ukrytymi. Warstwa wejściowa ma k neuronów wejściowych, pierwsza warstwa ukryta ma n ukrytych neuronów, natomiast druga zawiera m ukrytych neuronów. W zasadzie nie ma ograniczeń co do liczby ukrytych warstw. Wyjście w tym przykładzie to dwie klasy, y_1 i y_2. Na górze jest zawsze włączony neuron przesunięcia 1. Każde połączenie ma własną wagę w (nie pokazano jej dla uproszczenia)

Różne rodzaje funkcji aktywacji

Naukowcy zajmujący się neuronami biologicznymi odkryli setki, a może nawet ponad tysiąc różnych ich typów (patrz książka Gary'ego Marcusa i Jeremy'ego Freemana *The Future of the Brain*), dlatego musimy umieć zamodelować co najmniej kilka różnych typów sztucznych neuronów. Można to zrobić poprzez wykorzystanie różnych typów funkcji aktywacji, czyli funkcji określonych na wewnętrznym stanie neuronu reprezentowanym przez aktywację $a(x) = \Sigma_i w_i x_i$ obliczoną na podstawie wejść wszystkich neuronów wejściowych.

Funkcja aktywacji to funkcja określona jako $a(x)$, która definiuje wyjście neuronu. Najczęściej używane funkcje aktywacji to:

- $f(a) = a$: Funkcja ta przekazuje wartość aktywacji. Jest nazywana funkcją tożsamości.

Kup ksi k Pole ksi k

- $f(a)=\begin{cases}1 if\ a\geq 0\\ 0 if\ a<0\end{cases}$: Ta funkcja aktywuje neuron, jeśli wartość aktywacji jest większa od podanej wartości. Określa się ją jako progową funkcję aktywacji.
- $f(a)=\frac{1}{1+\exp(-a)}$: Ta funkcja jest jedną z najczęściej używanych. Przekazuje wartość w przedziale od 0 do 1 i można ją interpretować stochastycznie jako prawdopodobieństwo uaktywnienia neuronu. Jest powszechnie nazywana funkcją logistyczną lub logistycznym sigmoidem.
- $f(a)=\frac{2}{1+\exp(-a)}-1=\frac{1-\exp(-a)}{1+\exp(-a)}$: Tę funkcję aktywacji określa się jako sigmoid bipolarny. Jest to po prostu przeskalowany sigmoid logistyczny o wartościach w przedziale (–1, 1).
- $f(a)=\frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}=\frac{1-\exp(-2a)}{1+\exp(-2a)}$: Tę funkcję aktywacji określa się jako tangens hiperboliczny.
- $f(a)=\begin{cases}a\, if\ a\geq 0\\ 0 if\ a<0\end{cases}$: Ta funkcja aktywacji najbardziej przypomina odpowiednik dla neuronu biologicznego. Jest to połączenie funkcji tożsamości z funkcją progową. Określa się ją rektyfikatorem albo funkcją **ReLU** (*Rectified Linear Unit*).

Jakie są podstawowe różnice między tymi funkcjami aktywacji? Często różne funkcje aktywacji sprawdzają się lepiej dla różnych problemów. Ogólnie rzecz biorąc, tożsamościowa funkcja aktywacji lub funkcja progowa, choć jest powszechnie używana do tworzenia sieci neuronowych z takimi implementacjami, jak perceptron lub Adaline (*Adaptive Linear Neuron* — dosł. adaptacyjny neuron liniowy), niedawno straciła popularność na rzecz sigmoida logistycznego, tangensa hiperbolicznego lub funkcji ReLU. O ile funkcja tożsamości i funkcja progowa są znacznie prostsze i dlatego były preferowane w czasach, kiedy komputery nie miały wystarczającej mocy obliczeniowej, o tyle często lepiej używać funkcji nieliniowych, takich jak funkcje sigmoidalne albo ReLU.

Należy również zauważyć, że gdybyśmy używali tylko liniowych funkcji aktywacji, to wprowadzanie dodatkowych ukrytych warstw nie miałoby sensu, ponieważ złożenie funkcji liniowych nadal jest funkcją liniową. Ostatnie trzy funkcje aktywności różnią się następująco:

- Mają inną przeciwdziedzinę.
- Wraz ze wzrostem x ich pochodna maleje do zera.

Fakt, że pochodna funkcji aktywacji zmniejsza się do zera wraz ze wzrostem x oraz dlaczego to jest ważne, wyjaśnimy później. Na razie wystarczy, jeśli zaznaczymy, że gradient (np. pochodna) funkcji ma zasadnicze znaczenie dla trenowania sieci neuronowych. Mechanizm jest podobny do tego, który omawialiśmy na przykładzie regresji liniowej w pierwszym rozdziale, gdy staraliśmy się minimalizować funkcję, podążając w kierunku przeciwnym do kierunku wskazywanego przez jej gradient.

Kup ksi k   Pole ksi k

Zakres wartości przyjmowanych przez funkcję logistyczną to (0, 1), co jest jednym z powodów, dla których jest to preferowana opcja dla sieci stochastycznych, czyli sieci z neuronami, które mogą się uaktywnić na podstawie funkcji probabilistycznej.

Funkcja hiperboliczna jest bardzo podobna do logistycznej, ale jej zakres wynosi (–1, 1). Dla odróżnienia funkcja ReLU ma zakres (0, ∞), zatem może zwracać bardzo dużą wartość.

Ważniejsza jest jednak pochodna każdej z tych trzech funkcji. Dla funkcji logistycznej *f* pochodna ma postać *f* (1–f)*, a jeśli *f* jest funkcją tangens hiperboliczny, to jej pochodna to *(1+f)*(1–f)*.

Dla funkcji ReLU pochodna jest znacznie prostsza — jej wartość to po prostu $\begin{cases} 1 \, if \; a \geq 0 \\ 0 \, if \; a < 0 \end{cases}$.

Zobaczmy, jak szybko można obliczyć pochodną logistycznej funkcji sigmoidalnej. Zauważmy, że pochodną względem *a* z funkcji $\frac{1}{1+\exp(-a)}$ można wyrazić wzorem:

$$\frac{\exp(-a)}{(1+\exp(-a))\cdot(1+\exp(-a))} = \frac{1}{(1+\exp(-a))}\cdot\frac{(1+\exp(-a))-1}{(1+\exp(-a))}$$

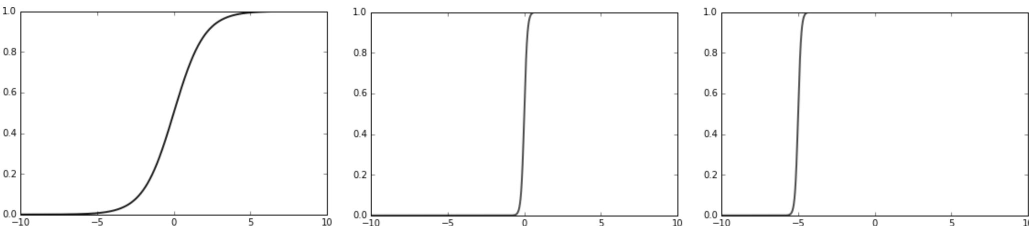
$$\frac{1}{(1+\exp(-a))}\cdot\left(\frac{(1+\exp(-a))}{(1+\exp(-a))}-\frac{1}{(1+\exp(-a))}\right) = f\cdot(1-f)$$

Gdy będziemy mówić o propagacji wstecznej, zobaczymy, że jednym z problemów sieci głębokich jest **znikający gradient** (jak wspomniano wcześniej), a zaleta funkcji aktywacji ReLU polega na tym, że jej pochodna jest stała i nie dąży do 0 w miarę wzrostu wartości funkcji.

Zwykle wszystkie neurony należące do tej samej warstwy mają taką samą funkcję aktywacji, natomiast w różnych warstwach funkcje aktywacji mogą być różne. Dlaczego jednak sieci neuronowe o głębokości przekraczającej jedną warstwę (złożone z dwóch lub większej liczby warstw) są tak ważne? Jak widzieliśmy, siła sieci neuronowych polega na ich zdolności do prognozowania — tzn. zdolności do aproksymacji funkcji zdefiniowanej na wejściu dla wymaganego wyjścia. Istnieje twierdzenie, zwane twierdzeniem o uniwersalnej aproksymacji, które głosi, że dowolną ciągłą funkcję na ograniczonych podzbiorach R^n można aproksymować za pomocą sieci neuronowej z co najmniej jedną warstwą ukrytą. Ponieważ formalny dowód tego twierdzenia jest zbyt złożony, aby go tutaj zaprezentować, podejmiemy próbę podania intuicyjnego wyjaśnienia, stosując kilka podstawowych reguł matematycznych. W tym celu w roli funkcji aktywacji użyjemy logistycznej funkcji sigmoidalnej.

Logistyczną funkcję sigmoidalną można zdefiniować jako $\frac{1}{1+\exp(-a)}$, gdzie $\alpha(x) = \Sigma_i w_i x_i + b$.

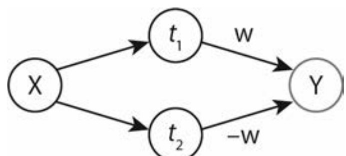
Załóżmy, że mamy tylko jeden neuron $x = x_i$:



Na wykresie z lewej strony zamieszczono standardową funkcję sigmoidalną z wagą 1 i przesunięciem 0. W środku jest wykres funkcji sigmoidalnej z wagą 10, natomiast po prawej przedstawiono funkcję sigmoidalną z wagą 10 i przesunięciem 50

W tym przypadku możemy łatwo wykazać, że jeśli w będzie bardzo duże, to funkcja logistyczna będzie bliska funkcji schodkowej (ang. *step function*). Im większa wartość w, tym funkcja bardziej przypomina funkcję schodkową, która w punkcie 0 ma wysokość 1. Z drugiej strony wartość b przesuwa wykres funkcji, a przesunięcie jest równe ujemnej wartości współczynnika b/w. Oznaczmy tę wartość $t = -b/w$.

Przyjmując te założenia, rozważmy prostą sieć neuronową z jednym neuronem wejściowym i jedną warstwą ukrytą złożoną z dwóch neuronów i jednego neuronu wyjściowego w warstwie wyjściowej:



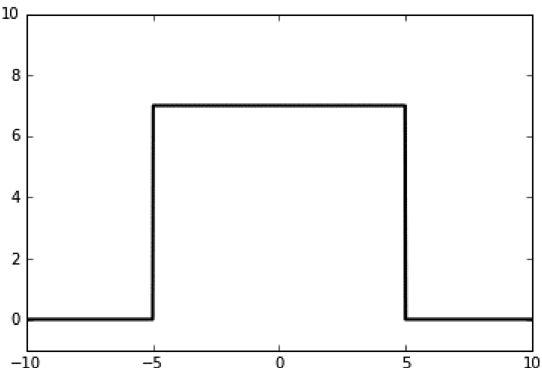
X jest odwzorowywany na dwa ukryte neurony z takimi wagami i przesunięciem, że na górnym ukrytym neuronie współczynnik $-b/w$ wynosi t_1, natomiast na ukrytym neuronie dolnym ten współczynnik wynosi t_2. Oba ukryte neurony korzystają z sigmoidalnej logistycznej funkcji aktywacji

Sygnał wejściowy x jest mapowany na dwa neurony — jeden z taką wagą i przesunięciem, że ich współczynnik wynosi t_1, i drugi, dla którego ten współczynnik wynosi t_2. Następnie dwa ukryte neurony mogą być zmapowane na neuron wyjściowy odpowiednio z wagami w i $-w$. Jeśli zastosujemy logistyczną, sigmoidalną funkcję aktywacji dla każdego ukrytego neuronu oraz funkcję tożsamościową dla neuronu wyjściowego (bez przesunięcia), to otrzymamy funkcję schodkową od t_1 do t_2 z wysokością w podobną do pokazanej na poniższym rysunku.

Ponieważ za pomocą ciągu funkcji schodkowych, takich jak ta, którą pokazano na rysunku, można zaproksymować dowolną funkcję ciągłą na kompaktowym podzbiorze R, możemy uzyskać obraz pokazujący, dlaczego twierdzenie o uniwersalnej aproksymacji jest prawdziwe (w uproszczonej formie jest to treść tzw. twierdzenia o aproksymacji funkcjami prostymi).

Kup ksi k Pole ksi k

Nieznacznie większym wysiłkiem można uogólnić to twierdzenie dla R^n.



Oto kod, który generuje wykres zamieszczony na poniższym rysunku:

```
#Użytkownik może zmieniać wartości wagi w,
#a także biasValue1 i biasValue2, aby obserwować,
#jak zmienia się wykres dla różnych funkcji schodkowych

import numpy
import matplotlib.pyplot as plt
weightValue = 1000
#do modyfikacji w celu zmiany początku funkcji schodkowej
biasValue1 = 5000
#do modyfikacji w celu zmiany końca funkcji schodkowej
biasValue2 = -5000

plt.axis([-10, 10, -1, 10])

print ("Funkcja schodkowa rozpoczyna się w punkcie {0} i kończy w punkcie {1}"
         .format(-biasValue1/weightValue,
         -biasValue2/weightValue))

y1 = 1.0/(1.0 + numpy.exp(-weightValue*x - biasValue1))
y2 = 1.0/(1.0 + numpy.exp(-weightValue*x - biasValue2))
#do modyfikacji w celu zmiany wysokości funkcji schodkowej
w = 7
y = y1*w-y2*w
plt.plot(x, y, lw=2, color='black')
plt.show()
```


Kup ksi k   Pole ksi k

Algorytm propagacji wstecznej

Wcześniej pokazaliśmy, że za pomocą sieci neuronowych można zmapować wejścia na zdeterminowane wyjścia w zależności od stałych wag. Po zdefiniowaniu **architektury** sieci neuronowych (typ bez sprzężenia zwrotnego, liczba ukrytych warstw, liczba neuronów w warstwie) i po wybraniu funkcji aktywacji dla każdego neuronu należy ustawić wagi, które z kolei zdefiniują wewnętrzne stany wszystkich neuronów w sieci.

Zobaczymy, jak to zrobić dla sieci jednowarstwowej, a następnie, w jaki sposób rozszerzyć to pojęcie dla głębokiej sieci bez sprzężenia zwrotnego. W przypadku głębokiej sieci neuronowej algorytm ustawiania wag określa się jako algorytm odwróconej dystrybucji. Na opisanie i objaśnienia dla tego algorytmu poświęcimy większą część niniejszego podrozdziału, ponieważ jest to jedno z najważniejszych pojęć wielowarstwowych sieci neuronowych bez sprzężenia zwrotnego. Najpierw jednak pokrótce omówimy ten algorytm dla jednowarstwowych sieci neuronowych.

Ogólne pojęcie, które należy zrozumieć, jest następujące: każda sieć neuronowa jest przybliżeniem funkcji, dlatego każda sieć neuronowa nie jest równa żądanej funkcji, ale różni się od niej o pewną wartość. Ta wartość to błąd, a naszym celem powinno być dążenie do zminimalizowania go.

Ponieważ w sieci neuronowej błąd jest funkcją wag, to chcemy minimalizować błędy w stosunku do wag.

Funkcja błędu jest funkcją wielu wag. Jest to zatem funkcja wielu zmiennych. W ujęciu matematycznym zbiór punktów, w których ta funkcja ma wartość 0, reprezentuje hiperpłaszczyznę, a żeby znaleźć minimum dla tej powierzchni, należy wybrać punkt, a następnie podążać po krzywej w kierunku minimum.

Regresja liniowa

Regresję liniową wprowadziliśmy już w pierwszym rozdziale, ale ponieważ teraz mamy do czynienia z wieloma zmiennymi, to w celu uproszczenia zastosujemy notację macierzową. Niech x będzie sygnałem wejściowym. Możemy interpretować x jako wektor. W przypadku regresji liniowej będziemy rozważać pojedynczy, wyjściowy neuron y. W związku z tym zbiór wag w jest wektorem o takim samym rozmiarze, co wektor x.

Załóżmy, że dla każdej wartości wejściowej x chcemy mieć na wyjściu docelową wartość t, natomiast dla każdego x sieć neuronowa zwraca wartość y zdefiniowaną przez wybraną funkcję aktywacji. W tym przypadku wartość bezwzględna różnicy $(y\text{-}t)$ stanowi różnicę pomiędzy wartością prognozowaną a wartością rzeczywistą dla konkretnej wejściowej próbki x.

Jeśli mamy m wartości wejściowych x^i, to każdej z nich odpowiada wartość docelowa t^i. W tym przypadku obliczamy błąd średniokwadratowy $\Sigma_i(y^i\text{-}t^i)^2$, gdzie każda wartość y^i jest funkcją wagi w.

Kup ksi k Pole ksi k

Zatem błąd jest funkcją wagi w i zwykle jest oznaczany jako $J(w)$.

Jak wspomniano wcześniej, błąd reprezentuje hiperpłaszczyznę o wymiarze równym wymiarowi w (niejawnie bierzemy pod uwagę także przesunięcie). Dla każdej wartości w_j trzeba znaleźć krzywą, która prowadzi do wartości minimalnej na tej płaszczyźnie. Kierunek, w którym krzywa wzrasta, jest określony przez jej pochodną. Można ją określić wzorem:

$$\vec{d} = \frac{\partial \sum_i \left(y^i - t^i\right)^2}{\partial w_j}$$

Aby poruszać się w kierunku minimum, trzeba dla każdej wartości w_j podążać w przeciwnym kierunku niż ten określonyprzez $\vec{d}$.

Obliczamy następujące wyrażenie:

$$\vec{d} = \frac{\partial \sum_i \left(y^i - t^i\right)^2}{\partial w_j} = \sum_i \frac{\partial \left(y^i - t^i\right)^2}{\partial w_j} = 2 \cdot \sum_i \frac{\partial y^i}{\partial w_j} \left(y^i - t^i\right)$$

Jeśli $y^i = <x^i, w>$, to $\frac{\partial y^i}{\partial w_j} = x_j^i$ i dlatego:

$$\vec{d} = \frac{\partial \sum_i \left(y^i - t^i\right)^2}{\partial w_j} = 2 \cdot \sum_i x_j^i \left(y^i - t^i\right)$$

> Notacja czasami może być myląca, zwłaszcza gdy widzi się ją po raz pierwszy. Wejścia są oznaczane za pomocą wektorów x^i, przy czym indeks górny oznacza i-tą próbkę. Ponieważ x i w są wektorami, to indeks dolny wskazuje na j-tą współrzędną wektora. Oznaczenie y^i reprezentuje wyjście sieci neuronowej w odpowiedzi na wejście x^i, natomiast t^i reprezentuje cel — tzn. pożądaną wartość odpowiadającą wejściu x^i.

Aby podążać w kierunku minimum, trzeba zmieniać każdą wagę zgodnie z kierunkiem jej pochodnej o niewielką wartość l. Jest to tzw. **szybkość uczenia się** (ang. *learning rate*), która zazwyczaj jest znacznie mniejsza niż 1 (zwykle 0,1 lub mniej). Możemy zatem zmodyfikować wzór, uwzględniając szybkość uczenia się. Otrzymujemy:

$$w_j \to w_j - \lambda \sum_i x_j^i \left(y^i - t^i\right)$$

Możemy też, bardziej ogólnie, zapisać zaktualizowany wzór w postaci macierzowej w następujący sposób:

$$w \to w - \lambda \nabla \left(\sum_i \left(y^i - t^i\right)^2 \right) = w - \lambda \nabla \left(J(w)\right)$$

Kup ksi k

Pole ksi k

W powyższym wzorze ∇ (tzw. nabla) reprezentuje wektor pochodnych cząstkowych. Proces opisany powyższym wzorem nazywany jest **metodą gradientu prostego** (ang. *gradient descent*).

> $\nabla = \left(\frac{\partial}{\partial w_i}, \cdots, \frac{\partial}{\partial w_n} \right)$ to wektor pochodnych cząstkowych. Zamiast zapisywania reguły aktualizacji wektora w oddzielnie dla każdego z jego składników w_j możemy to zrobić w postaci macierzowej, gdzie zamiast oznaczania cząstkowej pochodnej dla każdego j używamy ∇, co oznacza pochodne cząstkowe dla każdego j.

Należy dodatkowo zwrócić uwagę, że aktualizację można przeprowadzić po obliczeniu wszystkich wektorów wejściowych, jednak w niektórych przypadkach wagi mogą być aktualizowane po każdej próbce lub po pewnej ustalonej liczbie próbek.

Regresja logistyczna

W regresji logistycznej wyjście nie jest ciągłe. Jest raczej zdefiniowane jako zbiór klas. W tym przypadku funkcja aktywacji nie będzie, tak jak wcześniej, funkcją tożsamości. Zamiast niej użyjemy logistycznej funkcji sigmoidalnej. Logistyczna funkcja sigmoidalna, zgodnie z tym, co powiedzieliśmy wcześniej, ma wartość rzeczywistą z przedziału (0, 1) i dlatego może być interpretowana jako funkcja prawdopodobieństwa. Z tego względu dobrze się nadaje do rozwiązywania problemów polegających na rozdzieleniu dwóch klas. W tym przypadku celem jest wskazanie jednej z dwch klas, tej do której należy dany obiekt, a wyjście reprezentuje prawdopodobieństwo, że dany obiekt należy do jednej z tych dwóch klas (na przykład $t = 1$).

> Notacja znów może być myląca. Cel jest oznaczony przez t. Może on mieć w tym przykładzie dwie wartości. Są one często określane jako klasa 0 i klasa 1. Nie należy mylić tych wartości 0 i 1 z wartościami logistycznej funkcji sigmoidalnej, która jest funkcją ciągłą o wartościach rzeczywistych z przedziału od 0 do 1. Wartość rzeczywista funkcji sigmoidalnej reprezentuje prawdopodobieństwo, że wyjście będzie klasy 0 lub 1.

Jeśli a jest wartością aktywacji neuronu — tak, jak zdefiniowano wcześniej — to $\sigma(a)$ oznacza logistyczną funkcję sigmoidalną. Dlatego dla każdej próbki x prawdopodobieństwo, że wyjście będzie klasy y, z uwzględnieniem wag w, wynosi:

$$P(t|x,w) = \begin{cases} \sigma(a) & if\ t = 1 \\ 1 - \sigma(a) & if\ t = 0 \end{cases}$$

Powyższe równanie możemy zapisać krócej w następujący sposób:

$$P(t|x,w) = \sigma(a)^t (1 - \sigma(a))^{1-t}$$

A ponieważ dla każdej próbki x^i prawdopodobieństwa $P(t^i|x^i, w)$ są niezależne, to globalne prawdopodobieństwo można opisać w następujący sposób:

$$P(t|x,w) = \Pi_i P\left(t^i \middle| x^i, w\right) = \Pi_i \sigma\left(a^i\right)^{t^i} \left(1 - \sigma\left(a^i\right)\right)^{\left(1-t^i\right)}$$

Kup ksi k Pole ksi k

Jeśli obliczymy logarytm naturalny z poprzedniego równania (aby zamienić iloczyny w sumy), otrzymamy następujące wyrażenie:

$$\log\left(P(t|x,w)\right) = \log\left(\Pi_i \sigma(a)^{t^i}(1-\sigma(a))^{(1-t^i)}\right)$$
$$= \sum_i t^i \log\left(\sigma\left(a^i\right)\right) + \left(1-t^i\right)\log\left(1-\sigma\left(a^i\right)\right)$$

Celem jest teraz maksymalizacja tego logarytmu w celu uzyskania największego prawdopodobieństwa prognozowania prawidłowego wyniku. Zazwyczaj można to osiągnąć tak, jak w poprzednim przypadku, za pomocą metody gradientów prostych w celu minimalizacji funkcji kosztów $J(w)$ zdefiniowanej za pomocą wzoru $J(w) = -log(P(y | x,w))$.

Tak jak poprzednio, obliczamy pochodną funkcji kosztów względem wag w_j. Otrzymujemy:

$$\frac{\partial}{\partial w_j}\sum_i t^i \log\left(\sigma\left(a^i\right)\right) + \left(1-t^i\right)\log\left(1-\sigma\left(a^i\right)\right) = \sum_i \frac{\partial \sum_i t^i \log\left(\sigma\left(a^i\right)\right) + \left(1-t^i\right)\log\left(1-\sigma\left(a^i\right)\right)}{\partial w_j}$$

$$\sum_i t^i \frac{\partial \log\left(\sigma\left(a^i\right)\right)}{\partial w_j} + \left(1-t^i\right)\frac{\partial \log\left(1-\sigma\left(a^i\right)\right)}{\partial w_j} = \sum_i t^i\left(1-\sigma\left(a^i\right)\right)x_j^i + \left(1-t^i\right)\sigma\left(a^i\right)x_j^i$$

Aby zrozumieć ostatnie równanie, przypomnijmy następujące fakty:

$$\frac{\partial \sigma\left(a^i\right)}{\partial a^i} = \sigma\left(a^i\right)\left(1-\sigma\left(a^i\right)\right)$$

$$\frac{\partial \sigma\left(a^i\right)}{\partial a_j} = 0$$

$$\frac{\partial a^i}{\partial w_j} = \frac{\partial \sum_k w_k x_k^i + b}{\partial w_j} = x_j^i$$

W konsekwencji, zgodnie z regułą łańcuchową:

$$\sum_i \frac{\partial \log\left(\sigma\left(a^i\right)\right)}{\partial w_j} = \sum_i \frac{1}{\sigma\left(a^i\right)}\sigma\left(a^i\right)\left(1-\sigma\left(a^i\right)\right)x_j^i = \left(1-\sigma\left(a^i\right)\right)x_j^i$$

Podobnie:

$$\sum_i \frac{\partial \log\left(1-\sigma\left(a^i\right)\right)}{\partial w_j} = \sigma\left(a^i\right)x_j^i$$

Kup ksi k

Pole ksi k

Ogólnie rzecz biorąc, w przypadku wieloklasowego wyjścia t, gdzie t jest wektorem $(t_1, \ldots, t_n)$, możemy uogólnić to równanie, korzystając ze wzoru $J(w) = -log(P(y|x,w)) = \Sigma_{ij} t^i_j log(\sigma(a^i))$. W ten sposób otrzymujemy równanie aktualizacji wag:

$$w_j \rightarrow w_j - \lambda \sum_i x^i_j \left(\sigma\left(a^i\right) - t^i\right)$$

Przypomina ono regułę aktualizacji, z którą zetknęliśmy się podczas omawiania regresji liniowej.

Propagacja wsteczna

W przypadku sieci jednowarstwowej dostrajanie wag jest łatwe, ponieważ można skorzystać z regresji liniowej lub regresji logistycznej i równocześnie dostosować wagi tak, aby uzyskać mniejszy błąd (minimalizując funkcję kosztów). Dla wielowarstwowych sieci neuronowych możemy użyć podobnego rozumowania w przypadku wag, które są używane do połączenia ostatniej warstwy ukrytej z warstwą wyjścia, ponieważ wiemy, jaką postać ma mieć warstwa wyjściowa. Nie możemy jednak postąpić tak samo dla warstw ukrytych, ponieważ nie wiemy wcześniej, jakie powinny być w nich wartości neuronów. Zamiast tego obliczamy błąd w ostatniej ukrytej warstwie i szacujemy, jaki mógłby on być w warstwie poprzedniej. Błąd jest propagowany z ostatniej do pierwszej warstwy, stąd nazwa propagacja wsteczna.

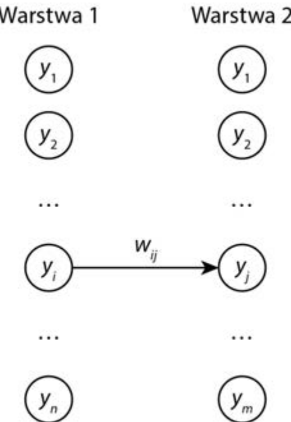
Propagacja wsteczna jest jednym z najbardziej skomplikowanych algorytmów. Aby jednak go zrozumieć, wystarczy znać podstawy rachunku różniczkowego, w tym zasadę różniczkowania funkcji złożonej. Najpierw wprowadzimy niektóre oznaczenia. Literą J oznaczamy koszt (błąd) dla funkcji aktywacji y zdefiniowanej na wartości aktywacji a (y może być na przykład logistyczną funkcją sigmoidalną), która jest funkcją wag w i wejścia x. Zdefiniujemy także $w_{i,j}$ — wagę pomiędzy i-tą wartością wejściową a j-tym wyjściem. Wejście i wyjście w opisie propagacji wstecznej rozumiemy bardziej ogólnie niż dla sieci jednowarstwowej: jeśli $w_{i,j}$ łączy parę sąsiednich warstw, to jako wejściowe oznaczamy neurony w pierwszej z dwóch kolejnych warstw, a jako wyjściowe neurony w drugiej z nich.

Aby zbytnio nie rozbudowywać notacji, a jednocześnie uwzględnić konieczność oznaczenia warstwy, w której jest każdy neuron, zakładamy, że i-te wejście y_i jest zawsze w warstwie poprzedzającej j-te wyjście y_j.

> Należy zwrócić uwagę, że litera y jest używana zarówno do oznaczenia wejścia, jak i wyjścia funkcji aktywacji. y_j jest wyjściem z funkcji aktywacji, ale jest również wejściem do warstwy następnej. Z tego względu możemy interpretować wartości y_j jako funkcje wartości y_j.

Używamy również indeksów dolnych i oraz j wszędzie, gdzie mamy element z indeksem dolnym i należącym do warstwy poprzedzającej warstwę zawierającą element oznaczony indeksem dolnym j.

Kup ksi k Pole ksi k



W tym przykładzie warstwa 1 reprezentuje wejście, a warstwa 2 wyjście, zatem $w_{i,j}$ to liczba łącząca wartość y_j w pierwszej warstwie z wartością y_j w następnej warstwie

Używając tej notacji oraz reguły różniczkowania funkcji złożonej, możemy zapisać dla ostatniej warstwy naszej sieci neuronowej następujący wzór:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

Ponieważ wiemy, że $\frac{\partial a_j}{\partial w_{i,j}} = y_j$, to otrzymujemy:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j} y_i$$

Jeśli y jest logistyczną funkcją sigmoidalną zdefiniowaną wcześniej, to otrzymamy taki sam wynik jak na końcu poprzedniego punktu, ponieważ znamy funkcję kosztów i możemy obliczyć wszystkie pochodne.

Dla wcześniejszych warstw zachodzi taka sama reguła:

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial w_{i,j}}$$

W rzeczywistości a_j jest funkcją aktywacji, która — jak wiadomo — jest funkcją wag. Wartość y_j jest funkcją aktywacji neuronu w drugiej warstwie, a funkcja kosztów zależy od wybranej funkcji aktywacji.

> Choć sieć składa się z kilku warstw, to zawsze koncentrujemy się na parach warstw sąsiednich i dlatego, być może nadużywając nieco notacji, zawsze mamy warstwę pierwszą i drugą, tak jak pokazano na rysunku 10. Są to odpowiednio warstwa wejścia i warstwa wyjścia.

Kup ksi k
Pole ksi k

Ponieważ wiemy, że $\frac{\partial a_j}{\partial w_{i,j}} = y_i$ oraz że $\frac{\partial y_j}{\partial a_j}$ jest pochodną funkcji aktywacji, którą możemy obliczyć, to musimy tylko obliczyć pochodną $\frac{\partial J}{\partial y_j}$.

Zwróćmy uwagę, że jest to pochodna błędu w odniesieniu do funkcji aktywacji drugiej warstwy. Jeśli możemy obliczyć tę pochodną dla ostatniej warstwy i mamy wzór, który pozwala obliczyć pochodną dla jednej warstwy przy założeniu, że możemy obliczyć pochodną dla warstwy następnej, to możemy obliczyć wszystkie pochodne, począwszy od ostatniej warstwy w kierunku pierwszej.

Należy pamiętać, że zgodnie z definicją wartości y_j są to wartości aktywacji dla neuronów drugiej warstwy, ale są one również funkcjami aktywacji, a zatem wartościami aktywacji warstwy pierwszej. Dlatego stosując regułę różniczkowania funkcji złożonej, otrzymujemy:

$$\frac{\partial J}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial y_i}$$

Tak jak wcześniej, możemy obliczyć zarówno $\frac{\partial y_j}{\partial a_j}$, jak i $\frac{\partial a_j}{\partial y_i} = w_{i,j}$, zatem kiedy znamy $\frac{\partial J}{\partial y_j}$, to możemy obliczyć $\frac{\partial J}{\partial y_i}$, a ponieważ możemy obliczyć $\frac{\partial J}{\partial y_j}$ dla ostatniej warstwy, to możemy poruszać się wstecz i obliczyć dla dowolnej warstwy $\frac{\partial J}{\partial y_i}$, a tym samym $\frac{\partial J}{\partial w_{i,j}}$.

Podsumowując, jeśli mamy sekwencję warstw, gdzie:

$$y_i \rightarrow y_j \rightarrow y_k$$

to prawdziwe są poniższe dwa podstawowe równania, gdzie sumę w drugim równaniu należy czytać jako sumę po wszystkich wychodzących połączeniach od y_j do dowolnego neuronu y_k w kolejnej warstwie.

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial w_{i,j}}$$

$$\frac{\partial J}{\partial y_i} = \sum_k \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial y_j}$$

Korzystając z tych dwóch równań, można obliczyć pochodne kosztów w odniesieniu do każdej warstwy.

Kup ksi k

Pole ksi k

Jeśli podstawimy $\delta_j = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}$, to δ_j reprezentuje odchylenie kosztu względem wartości aktywacji. Wartość δ_j możemy interpretować jako błąd na poziomie neuronu y_j. Możemy zatem przepisać powyższe równanie:

$$\frac{\partial J}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}\frac{\partial a_j}{\partial y_i} = \sum_j \delta_j w_{i,j}$$

Z tego wynika, że $\delta_i = \left(\sum_j \delta_j w_{i,j}\right)\frac{\partial y_i}{\partial a_i}$. Powyższe dwa równania pozwalają patrzeć na propagację wsteczną w inny sposób — jako odchylenie kosztu w stosunku do wartości aktywacji — i dostarczają wzoru do obliczenia tego odchylenia dla każdej warstwy, jeśli dysponujemy wartością odchylenia dla warstwy następnej:

$$\delta_j = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial a_j}$$

$$\delta_i = \left(\sum_j \delta_j w_{i,j}\right)\frac{\partial y_i}{\partial a_i}$$

Możemy również połączyć te równania i wykazać, że:

$$\frac{\partial J}{\partial w_{i,j}} = \delta_j \frac{\partial a_j}{\partial w_{i,j}} = \delta_j y_i$$

Algorytm propagacji wstecznej dla aktualizacji wag można określić dla każdej warstwy następującym wzorem:

$$w_{i,j} \to w_{i,j} - \lambda \delta_j y_i$$

W ostatnim punkcie tego podrozdziału zaprezentujemy przykładowy kod, który pomoże zrozumieć i zastosować przedstawione pojęcia i wzory.

Zastosowania praktyczne

Kilka przykładów zastosowań uczenia maszynowego zaprezentowaliśmy w poprzednim rozdziale. Sieci neuronowe mają wiele podobnych zastosowań.

W tym punkcie dokonamy przeglądu wybranych zastosowań, w których użyto sieci neuronowych w czasie, gdy stały się one popularne w końcu lat osiemdziesiątych i na początku lat dziewięćdziesiątych. Wtedy to odkryto propagację wsteczną i powstały możliwości trenowania sieci neuronowych o większej głębokości.

Kup ksi k Pole ksi k

Przetwarzanie sygnałów

Jest wiele zastosowań sieci neuronowych w dziedzinie przetwarzania sygnałów. Jednym z pierwszych było tłumienie echa na liniach telefonicznych, zwłaszcza dla połączeń międzykontynentalnych. Bernard Widrow i Marcian Hoff rozpoczęli pracę nad ich rozwojem w 1957 roku. W systemie Adaline do trenowania wykorzystano w roli funkcji aktywacji funkcję tożsamościową. Dążono do minimalizacji średniokwadratowego błędu pomiędzy wartością aktywacji a wartością docelową. System Adaline był trenowany pod kątem usunięcia echa z sygnału na linii telefonicznej poprzez podanie sygnału wejściowego zarówno na układ Adaline, jak i na linię telefoniczną. Różnica pomiędzy wyjściem z linii telefonicznej a wyjściem z systemu Adaline to błąd, który jest używany do uczenia sieci i usuwania zakłóceń (echa) z sygnału.

Medycyna

System **Instant physician** (dosł. błyskawiczny lekarz) został zaprojektowany przez Jamesa Andersona w 1986 roku. Głównym założeniem systemu było przechowywanie dużej ilości dokumentacji medycznej zawierającej informacje na temat objawów, diagnostyki i leczenia dla każdego przypadku. Uczenie sieci było ukierunkowane na dokonywanie jak najlepszych diagnoz i leczenia dla podanych różnych objawów.

Znacznie później, wykorzystując głębokie sieci neuronowe, firma IBM opracowała sieć neuronową zdolną do przewidywania na podstawie notatek lekarzy — podobnie jak robią to doświadczeni kardiolodzy — możliwych schorzeń serca.

Autonomiczne samochody

Derrick Nguyen i Bernard Widrow w 1989 roku oraz Thomas Miller, Richard Sutton i Paul Werbos w roku 1990 opracowali sieć neuronową zdolną do dostarczania wskazówek dotyczących sterowania dużym samochodem ciężarowym z przyczepą cofającym do strefy załadunku. Sieć neuronowa w tym zastosowaniu składa się z dwóch modułów: pierwszy potrafi obliczać nowe pozycje za pomocą sieci neuronowej złożonej z wielu warstw, przez naukę reakcji pojazdu na różne sygnały i jest nazywany emulatorem. Drugi moduł zwany kontrolerem uczy się wydawać odpowiednie polecenia za pomocą emulatora, aby poznać pozycję pojazdu. W ostatnich latach autonomiczne samochody odniosły ogromny sukces i stały się rzeczywistością, choć zastosowano w nich znacznie bardziej skomplikowane głębokie sieci neuronowe w połączeniu z sygnałami wejściowymi z kamer, odbiorników GPS, lidarów i sonarów.

Biznes

W 1988 roku Edward Collins, Sushmito Ghosh i Christopher Scofield opracowali sieć neuronową, którą można wykorzystać do oszacowania, czy należy udzielić kredytu hipotecznego. Wykorzystując dane pochodzące od rzeczoznawców kredytów hipotecznych, szkolono sieci neuronowe pod kątem ustalania, czy wnioskodawcom może być przyznany kredyt. Wejściem było kilka cech takich, jak liczba lat zatrudnienia wnioskodawcy, poziom dochodów, liczba osób na utrzymaniu, szacowana wartość nieruchomości i tak dalej.

Kup ksi k Pole ksi k

Rozpoznawanie wzorców

Ten problem omawialiśmy wiele razy. Jednym z obszarów, w których zastosowano sieci neuronowe, jest rozpoznawanie znaków. Mechanizm ten może na przykład być stosowany do rozpoznawania cyfr, a także pisanych ręcznie kodów pocztowych.

Generowanie mowy

W 1986 roku Terrence Sejnowski i Charles Rosenberg opracowali powszechnie znany przykład biblioteki NETtalk, która generuje wypowiadane słowa przez czytanie tekstu pisanego. Wymaga ona do działania zbioru próbek napisanych słów wraz z wymową. Wejście zawiera zarówno wymawianą literę, jak i litery przed nią i za nią (zazwyczaj trzy), a trening odbywa się z wykorzystaniem najczęściej wypowiadanych słów wraz z ich transkrypcją fonetyczną. W implementacji biblioteki sieć neuronowa najpierw uczy się rozróżniania samogłosek od spółgłosek, a następnie rozpoznawania początków i zakończeń słów. Zanim wypowiadane słowa staną się zrozumiałe, zwykle potrzeba kilku przebiegów, a postępy uczenia się czasami przypominają nauczanie dzieci sposobu wymawiania słów.

Przykład kodu sieci neuronowej dla funkcji XOR

Powszechnie wiadomo — wspominaliśmy o tym już wcześniej — że jednowarstwowe sieci neuronowe nie potrafią przewidzieć wyniku funkcji `XOR`. Potrafią klasyfikować tylko zbiory dające się rozdzielić liniowo. Jednak, jak widzieliśmy, zgodnie z twierdzeniem o uniwersalnej aproksymacji, każdą funkcję można aproksymować za pomocą sieci dwuwarstwowej o odpowiednio złożonej architekturze. W prezentowanym przykładzie stworzymy sieć neuronową z dwoma neuronami w ukrytej warstwie i pokażemy, że za jej pomocą można zamodelować funkcję `XOR`.

Kod napiszemy jednak w taki sposób, aby Czytelnik mógł go zmodyfikować dla dowolnej liczby warstw i neuronów w każdej warstwie. Dzięki temu możliwe jest zasymulowanie różnych scenariuszy. Jako funkcję aktywacji dla tej sieci wykorzystamy funkcję tangensa hiperbolicznego. Na potrzeby trenowania sieci zaimplementujemy opisany wcześniej algorytm propagacji wstecznej.

Będziemy musieli zaimportować tylko jedną bibliotekę — `numpy`, chociaż jeśli Czytelnik chce zwizualizować wyniki, polecamy również zaimportowanie biblioteki `matplotlib`. Dlatego pierwsze linijki kodu mają następującą postać:

```
import numpy
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
```

Następnie definiujemy naszą funkcję aktywacji i jej pochodne (w tym przykładzie używamy funkcji `tanh(x)`):

Kup ksi k Pole ksi k

```
def tanh(x):
    return (1.0 - numpy.exp(-2*x))/(1.0 + numpy.exp(-2*x))

def tanh_derivative(x):
    return (1 + tanh(x))*(1 - tanh(x))
```

Następnie zdefiniujemy klasę `NeuralNetwork`:

```
class NeuralNetwork:
```

Zgodnie z regułami składni Pythona dla całego kodu wewnątrz klasy `NeuralNetwork` trzeba zastosować wcięcie. Definiujemy konstruktor klasy `NeuralNetwork`, tzn. jej zmienne, które w tym przypadku będą tworzyć architekturę sieci neuronowej, czyli ile ma warstw i ile neuronów w każdej warstwie. Zainicjujemy również losowo wagi tak, by ich wartości mieściły się w zakresie pomiędzy –1 a 1. `net_arch` jest jednowymiarową tablicą zawierającą liczby neuronów w każdej warstwie: na przykład `[2,4,1]` oznacza warstwę wejścia z dwoma neuronami, ukrytą warstwę z czterema neuronami i warstwę wyjścia z jednym neuronem.

Ponieważ badamy funkcję `XOR`, to w warstwie wejścia musimy mieć dwa neurony, natomiast w warstwie wyjścia tylko jeden:

```
#net_arch składa się z listy liczb całkowitych, które oznaczają
#liczbę neuronów w każdej warstwie,
# tzn. architekturę sieci
def init (self, net_arch):
    self.activity = tanh
    self.activity_derivative = tanh_derivative
    self.layers = len(net_arch)
    self.steps_per_epoch = 1000
    self.arch = net_arch

    self.weights = []
    #zakres wartości wag (–1,1)
    for layer in range(self.layers - 1):
        w = 2*numpy.random.rand(net_arch[layer] + 1,
                                net_arch[layer+1]) - 1
        self.weights.append(w)
```

W tym kodzie określiliśmy funkcję aktywacji jako tangens hiperboliczny i zdefiniowaliśmy jej pochodną. Określiliśmy także liczbę kroków treningowych przypadających na epokę. Na koniec zainicjowaliśmy wagi, dbając o to, aby zrobić to także dla przesunięć, które dodamy później. Dalej musimy zdefiniować funkcję `fit`, która będzie trenować naszą sieć. W ostatnim wierszu `nn` reprezentuje klasę `NeuralNetwork`, natomiast `predict` to metoda tej klasy, którą zdefiniujemy później:

```
#data to zbiór wszystkich możliwych par wartości logicznych
#True lub False określonych przez liczby 1 lub 0
#labels zawiera wynik logicznej operacji 'xor'
#na każdej z tych par wejściowych
```


Kup ksi k Pole ksi k

```
    def fit(self, data, labels, learning_rate=0.1, epochs=100):
        #Dodanie przesunięć do warstwy wejścia
        ones = numpy.ones((1, data.shape[0]))
        Z = numpy.concatenate((ones.T, data), axis=1)
        training = epochs*self.steps_per_epoch
        for k in range(training):
            if k % self.steps_per_epoch == 0:
                print('epochs: {}'.format(k/self.steps_per_epoch))
                for s in data:
                    print(s, nn.predict(s))
```

Działanie powyższego kodu sprowadza się do dodania "1" do danych wejściowych (zawsze włączony neuron przesunięcia) i skonfigurowania go w taki sposób, aby w celu śledzenia postępów wyświetlać wynik na koniec każdej epoki. Teraz wykonamy kolejny krok — skonfigurujemy propagację w przód:

```
            sample = numpy.random.randint(data.shape[0])
            y = [Z[sample]]
            for i in range(len(self.weights)-1):
                activation = numpy.dot(y[i], self.weights[i])
                activity = self.activity(activation)
                #dodaj przesunięcie do następnej warstwy
                activity = numpy.concatenate((numpy.ones(1),
                           numpy.array(activity)))
                y.append(activity)

            #ostatnia warstwa
            activation = numpy.dot(y[-1], self.weights[-1])
            activity = self.activity(activation)
            y.append(activity)
```

Po każdym kroku chcemy aktualizować wagi, zatem losowo wybieramy jeden z wejściowych punktów danych, następnie konfigurujemy propagację w przód poprzez ustawienie aktywacji dla każdego neuronu, a potem stosujemy funkcję `tanh(x)` w odniesieniu do wartości aktywacji. Ponieważ mamy przesunięcie, dodajemy je do naszej macierzy `y`, która zawiera wartości wyjściowe dla każdego neuronu.

Następnie, w celu korekty wag, wykonujemy algorytm propagacji wstecznej dla błędu:

```
            #błąd dla warstwy wyjścia
            error = labels[sample] - y[-1]
            delta_vec = [error * self.activity_derivative(y[-1])]
            #należy zacząć od tyłu
            #od przedostatniej warstwy
            for i in range(self.layers-2, 0, -1):
                error = delta_vec[- 1].dot(self.weights[i][1:].T)
                error =
                error*self.activity_derivative(y[i][1:])
                delta_vec.append(error)
            #Teraz trzeba ustawić wartości od tyłu do przodu
            delta_vec.reverse()
```


Kup ksi k Pole ksi k

```
        #Na koniec korygujemy wagi,
        #używając reguł propagacji wstecznej
        for i in range(len(self.weights)):
            layer = y[i].reshape(1, nn.arch[i]+1)
            delta = delta_vec[i].reshape(1, nn.arch[i+1])
            self.weights[i]
            +=learning_rate*layer.T.dot(delta)
```

Na tym kończymy implementację algorytmu propagacji wstecznej. Pozostało napisanie funkcji `predict`, która sprawdza wyniki:

```
def predict(self, x):
    val = numpy.concatenate((numpy.ones(1).T, numpy.array(x)))
    for i in range(0, len(self.weights)):
        val = self.activity(numpy.dot(val, self.weights[i]))
        val = numpy.concatenate((numpy.ones(1).T,
                                 numpy.array(val)))
    return val[1]
```

W tym momencie trzeba jeszcze napisać funkcję `main`, która ma następującą postać:

```
if __name__ == '__main__':
    numpy.random.seed(0)
    #Zainicjowanie obiektu klasy NeuralNetwork z
    #2 neuronami wejściowymi
    #2 neuronami ukrytymi
    #1 neuronem wyjściowym
    nn = NeuralNetwork([2,2,1])
    X = numpy.array([[0, 0],
                     [0, 1],
                     [1, 0],
                     [1, 1]])
    #Ustawienie etykiet — prawidłowych wyników operacji XOR
    y = numpy.array([0, 1, 1, 0])
    #Wywołanie funkcji fit i trenowanie sieci
    #dla wybranej liczby epok
    nn.fit(X, y, epochs=10)
    print "Ostateczna prognoza"
    for s in X:
        print(s, nn.predict(s))
```

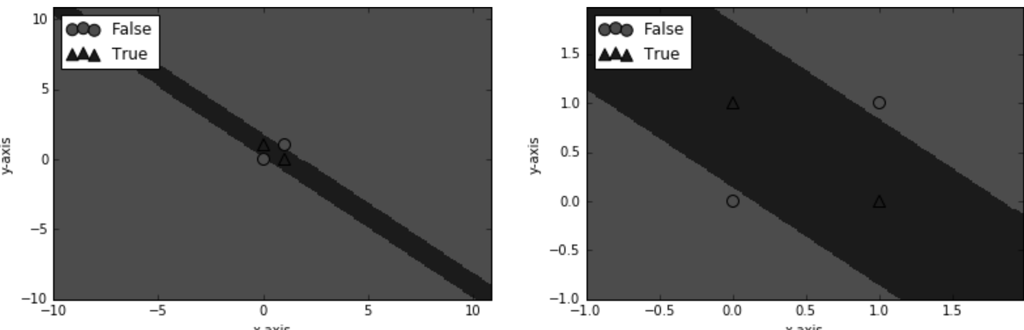
Zwróćmy uwagę na użycie wywołania `numpy.random.seed(0)`. Wykorzystano je, aby zadbać o spójność inicjalizacji wag w kolejnych przebiegach, tak aby można było porównać wyniki, ale nie jest ono konieczne dla implementacji sieci neuronowej.

Na tym kończy się kod. W wyniku jego działania powinniśmy uzyskać czterowymiarową tablicę, na przykład: `(0.003032173692499, 0.9963860761357, 0.9959034563937, 0.0006386449217567)`, co pokazuje, że sieć uczy się, że prawidłowy wynik to `(0,1,1,0)`.

Czytelnik może nieco zmodyfikować kod wykorzystanej wcześniej w tej książce funkcji `plot_decision_regions`, aby zobaczyć, w jaki sposób różne sieci neuronowe oddzielają różne obszary w zależności od wybranej architektury.

Kup ksi k Pole ksi k

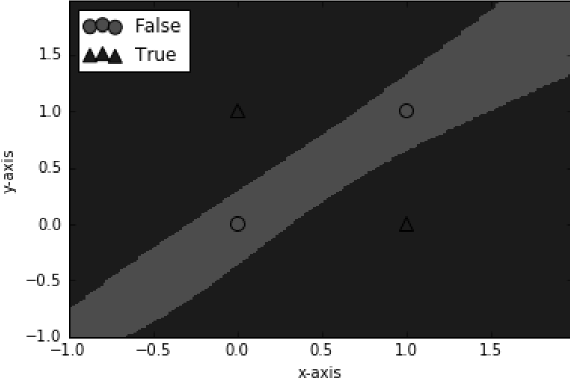
Obraz wyjściowy będzie wyglądał tak, jak pokazano na poniższych ilustracjach. Kółka reprezentują wejścia (`True, True`) i (`False, False`), a trójkąty wejścia (`True, False`) i (`False, True`) dla funkcji `XOR`.



Ten sam rysunek, po lewej zmniejszony, a po prawej powiększony dla wybranych wejść. Sieć neuronowa uczy się oddzielać te punkty, tworząc pasmo zawierające dwie wartości wyjściowe True

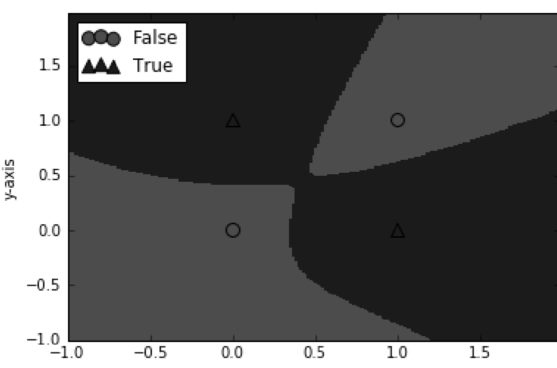
Różne architektury sieci neuronowych (na przykład implementacja sieci z różną liczbą neuronów w warstwie ukrytej lub z więcej niż jedną warstwą ukrytą) mogą generować różne obszary rozdzielające. Aby zmienić architekturę sieci, wystarczy zmodyfikować w kodzie linijkę `nn = NeuralNetwork([2,2,1])`. O ile pierwszy argument 2 powinien być zachowany (wejście się nie zmienia), drugą 2 można zmodyfikować w celu zaznaczenia występowania innej liczby neuronów w warstwie ukrytej. Dodanie kolejnej liczby całkowitej wprowadza nową warstwę ukrytą z liczbą neuronów zgodną z dodaną wartością. Ostatniej 1 nie można zmodyfikować. Na przykład (`[2,4,3,1]`) będzie reprezentować trójwarstwową sieć neuronową z czterema neuronami w pierwszej warstwie ukrytej i trzema w drugiej.

Czytelnik zaobserwowałby wtedy, że choć rozwiązanie jest zawsze takie samo, to krzywe rozdzielające regiony będą różne w zależności od wybranej architektury. Ustawienie `nn = Neural` ↪`Network([2,4,3,1])` daje następujący obraz:



Kup książkę Poleć książkę

Z kolei ustawienie `nn = NeuralNetwork([2,4,1])` generuje następujący wynik:



Zatem architektura definiuje sposób, w jaki sieć neuronowa podchodzi do rozwiązania problemu, a różne architektury zapewniają różne podejścia (choć wszystkie mogą dawać ten sam wynik). Na podobnej zasadzie ludzkie procesy myślowe mogą podążać różnymi drogami, aby dojść do tego samego wniosku. Teraz jesteśmy gotowi, aby przyjrzeć się uważniej, czym są głębokie sieci neuronowe i jakie są ich zastosowania.

Podsumowanie

W tym rozdziale zaprezentowaliśmy szczegółowe informacje na temat sieci neuronowych i wspomnieliśmy o ich zaletach w porównaniu z innymi konkurencyjnymi algorytmami. Sieci neuronowe składają się z jednostek, czyli neuronów, które należą do sieci, lub ich połączeń, czyli wag, charakteryzujących siłę komunikacji pomiędzy poszczególnymi neuronami, oraz z funkcji aktywacji opisujących sposób, w jaki neurony przetwarzają informacje.

W rozdziale opowiedzieliśmy, jak można tworzyć różne architektury oraz że sieć neuronowa może składać się z wielu warstw i dlaczego wewnętrzne (ukryte) warstwy są ważne. Opowiedzieliśmy, że informacje płyną od wejścia do wyjścia, przechodząc z każdej warstwy do następnej, w zależności od wag i zdefiniowanej funkcji aktywacji. Na koniec pokazaliśmy, że możemy zdefiniować metodę — tzw. wsteczną propagację — w celu dostrojenia wag, po to, by poprawić pożądany poziom dokładności. Wymieniliśmy również wiele obszarów, w których sieci neuronowe były i są stosowane.

W następnym rozdziale będziemy kontynuować omawianie głębokich sieci neuronowych, a w szczególności wyjaśnimy sens słowa „głębokie”, podobnie jak w głębokim uczeniu, tłumacząc, że określenie to odnosi się nie tylko do liczby ukrytych warstw w sieci, ale też — co jest znacznie ważniejsze — do jakości uczenia sieci neuronowej. W tym celu pokażemy, jak

Kup ksi k Pole ksi k

sieci neuronowe uczą się rozpoznawać cechy i łączyć je ze sobą jako reprezentacje rozpoznawanych obiektów. To otworzy nam drogę do wykorzystania sieci neuronowych w dziedzinie uczenia nienadzorowanego.

Ponadto opiszemy kilka ważnych bibliotek uczenia głębokiego, a na koniec zaprezentujemy konkretny przykład zastosowania sieci neuronowych do rozpoznawania cyfr.

Kup ksi k

Pole ksi k

Skorowidz

A

Adadelta, 312
Adagrad, 311
AI, Artificial Intelligence, 18, 195
akceleracja Nesterova, 310
algorytm
    adadelta, 151
    Bayesa, 28, 33, 34
    drzewa decyzyjnego, 30, 31
    HOGWILD!, 307
    maszyny wektorów podpierających, 34
    min-max, 198, 206
    MLPClassifier, 47
    propagacji wstecznej, 61, 68
    przeszukiwania drzewa Monte Carlo, 220
    Q-learning, 263
    SGD, 151, 302, 306, 308, 350
    UCB1, 215
    Viterbiego, 189
algorytmy
    generatywne, 80
    genetyczne, 237
    grupowania, 28, 31
    przewidywania, 80
    regresji, 28, 42
    uczenia głębokiego, 88
AlphaGo, 230
    metoda policy gradients, 230
analiza głównych składowych, PCA, 108
analiza regresji, 28
anomalia, 273
    kontekstowa, 279
    punktowa, 279
    zbiorowa, 279
Apache Spark, 302, 318
API REST, 347
aproksymacja wielomianem drugiego rzędu, 166
architektura uczenia głębokiego, 315
atak typu DOS, 339
AUC, Area Under the Curve, 330, 332
autoenkodery, 89, 107, 117, 299
    kompresujące, 114
    metody regularyzacji, 113
    odszumiające, 113
    rzadkie, 116
automatyczny wybór cech, 85

B

baseline, 267
biblioteka
    Keras, 97, 150, 153
    Sparkling Water, 350
    TensorFlow, 97, 126
    Theano, 96, 148
    H2O, 283
biblioteki open source, 96
big data, 18
blok przesunięcia, 38
błąd
    rekonstrukcji, 283, 292
    średniokwadratowy, 111, 282
BM, Boltzmann Machines, 89
BPTT, back-propagation through time, 176

C

CNN, Convolutional Neural Networks, 89
CPU, Computing Processing Unit, 17, 50, 94
CTC, Connectionist Temporal Classification, 187
czas do wykrycia, 339, 343
czułość, 325

D

dane
nieoznakowane, 331
oznakowane, 328
DBN, Deep Belief Networks, 23, 89, 132
DCT, Discrete Cosine Transform, 186
dekodowanie, 189
długa pamięć krótkotrwała, 168
dostrajanie hiperparametrów, 335
dropout, 147
drzewo
decyzyjne, 28–32
Monte Carlo, 211, 220
dyskretna transformata kosinusowa, DCT, 186

E

EBM, Energy-Based Model, 120
EM, Excess-Mass, 321–334
enkoder, 287
entropia krzyżowa, 36, 111
epsilon zachłanny, 253

F

fonem, 189
formalna elegancja, 323
format POJO, 344
framework H2O, 317, 337, 347
funkcja, 29, 249
fit, 99
MSE, 282
predict, 44
Q, 239, 240
ReLU, 48, 57, 101
schodkowa, 59
sigmoidalna, 59, 111
tanh, 48
XOR, 70
funkcje
aktywacji, 47, 51, 56, 99, 111
kosztów, 99
strat, 282
strategii uczenia, 222

G

generowanie modeli, 80
głębokie
autoenkodery, 110
sieci konwolucyjne, 259
sieci neuronowe, 77
górna granica przedziału ufności, 215
górny poziom zaufania dla drzew, 215
GPU, Graphical Processing Units, 17, 50, 94
gra
Breakout, 254
losowy test, 255
wstępne przetwarzanie ekranu, 257
typu labirynt, 196
w Go, 210
w kółko i krzyżyk, 198, 218
implementacja, 201
gradienty, 223
eksplodujące, 165
zanikające, 165
grupowanie, 22, 23
gry
dynamiczne, 246
komputerowe, 237
planszowe, 195

H

H2O, 283, 285
hiperparametry, 311, 322, 335, 338
hiperpłaszczyzna, 35
HMM, Hidden Markov Models, 186
HOGWILD!, 306

I

implementacja
głębokiej sieci neuronowej, 98
gry w kółko i krzyżyk, 201
inicjalizacja wag, 304, 308
interfejs API, 318, 347
intuicja, 137

J

jednolitość, 323
jednowarstwowe sieci neuronowe, 56

Kup ksi k

Pole ksi k

K

Keras, 97, 98
warstwy konwolucyjne, 150, 153
klasteryzacja, 22, 23, 31, 32
metodą k-średnich, 31, 32
klasyfikacja, 21, 22, 25
koszt szkody, 340
KPI, Key Performance Indicators, 326, 338, 340, 341
krok, 144
kryteria akceptacji dobrej teorii, 322
krzywa ROC, 329
k-średnie, 22, 28, 31

L

liść, 142
LSTM, Long Short Term Memory, 168

M

macierz wag, 100
maszyna
ograniczona Boltzmanna, RBM, 80, 89, 119, 125, 133
Boltzmanna, BM, 89, 121, 123
wektorów podpierających, 34
max-pooling, 145
mechanizm MapReduce, 314
metoda
aktor-krytyk, 266
baseline, 267
entropii krzyżowej, 36, 223
gradientu prostego, 63
MCTS, 211, 213
MCTS-UCT, 231
min_max, 207, 213
Newtona, 310, 313
policy gradients, 222, 224, 230
SGD, 306
wektorów podpierających, SVM, 42
metody
asynchroniczne, 268
optymalizacji, 167
MLP, Multilayer Perceptrons, 89
MNIST, 98
moc
objaśniania, 323
unifikacji, 323
modele
akustyczne, 186
biologiczne, 136
EBM, 120
GMM, 186
języka, 171
bazujące na słowach, 171
bazujące na uwadze, 188
bazujące na znakach, 176
LSTM, 179
neuronowe, 173
n-gramy, 171
od końca do końca, 190
sztuczne, 136
modelowanie
danych, 279
wykrywania, 279
mowa jako dane wejściowe, 184
MSE, Mean Squared Error, 111
MV, Mass-Volume, 321, 331, 331–334
MVP, Minimal Viable Product, 303

N

nadmierne dopasowanie, 26, 27
naiwny algorytm Bayesa, 33
network daydreaming, 121
neuron, 37–39, 52
neuronowe modele języka, 173
n-gramy, 171
nieliczność modelu, 324
nienadzorowane uczenie cech, 105
NLP, Natural Language Processing, 42

O

obcinanie gradientu, 166
ocena F, 330
anomalii, 321, 328, 331, 347
od końca do końca, 338, 350
stanów gry, 198
odczytywanie danych, 177
odtwarzanie doświadczeń, 250
ograniczone maszyny Boltzmanna, RBM, 80, 89, 119, 125
optymalizator sgd, 99
owocność, 323, 324

Kup ksi k Pole ksi k

P

pakiet scikit-learn, 42
pamięć LSTM, 168
PC, Principal Component, 109
PCA, Principal Component Analysis, 108
perceptron, 37, 38, 43
perceptrony wielowarstwowe, **MLP**, 89
pęd, 309, 310, 313, 343
plaster, 142
płytkie techniki uczenia maszynowego, 278
POC, proof of concepts, 285
podejścia bazujące na modelach, 269
POJO, Plain Old Java Object, 344
policy gradients, 222, 224, 265
poszukiwanie nowości, 253
potok rozpoznawania mowy, 183
PR, Precision-Recall, 321, 330, 331, 334
prawdopodobieństwo
  a posteriori, 186
  a priori, 34
precyzja, 325
produkt danych, 301–304, 350
prognozowanie prawdopodobieństwa, 231
propagacja wsteczna, 61, 65
  w czasie, BPTT, 163, 167, 176
prostota, 323
próbkowanie Monte Carlo, 124, 212
przedział ufności, 214
przesunięcie, bias, 53
przeszukiwanie drzewa Monte Carlo, 211, 220
przetwarzanie
  ekranu, 257
  języka naturalnego, NLP, 42
  wstępne dźwięku, 185

Q

Q-learning, 238, 241, 263, 265

R

RBM, Restricted Boltzmann Machines, 80, 89, 119, 125
redukcja wariancji, 267
regresja
  liniowa, 28, 61
  logistyczna, 63
rekurencyjne sieci neuronowe, RNN, 159–90
ReLU, Rectified Linear Unit, 57
RL, 231
RMS, Root Mean Square, 312
RNN, Recurrent Neural Networks, 159
ROC, Receiver Operating Characteristic, 321
rozkład błędów, 289
rozpoznawanie
  mowy, 183
    bazujące na uwadze, 188
    model akustyczny, 186
    potok, 183
    przetwarzanie wstępne, 185
  obrazów, 135
równomierna inicjalizacja adaptacyjna, 305

S

selekcja modelu, 321
SGD, stochastic gradient descent, 151, 307–313, 342, 350
sieci
  DBN, 23, 89, 130, 186
  Hopfielda, 121
  konwolucyjne, 259
  neuronowe, 17, 25–49
    architektura, 51
    funkcje aktywacji, 51, 56
    głębokie, 77
    jednowarstwowe, 55
    modelowanie funkcji XOR, 70
    neurony, 52
    rekurencyjne, 159–90
    splotowe, 89
    szkolenie wstępne, 155
    uczenie, 51
    warstwy, 52
    zastosowania, 68
sieć
  AlexNet, 95
  CTC, 188
  LSTM, 91, 170, 177
    próbkowanie, 181
    trening, 180, 182
  RL, 231
  SL, 231
  V, 231
siła unifikacyjna, 324
skuteczność wykrywania, 338
Sparkling Water, 302, 317, 350

Kup ksi k Pole ksi k

splotowe sieci neuronowe, CNN, 89
stabilność modelu, 339
strategia
  szybkiej symulacji, 231
  uczenia, 222
SVM, support vector machines, 42
symulacje przypadków awarii, 339
system
  AI
    gra w Go, 210
    grające w gry, 197
    trenowanie, 210
  AlphaGo, 230
  wykrywania włamań, 301
szkolenie wstępne, 155
sztuczka jądra, 35
sztuczna inteligencja, AI, 18, 195
szybkość uczenia się, 62

Ś

śmiałość, 323, 324

T

tablica pomyłek, 329
technika HOGWILD!, 306
techniki uczenia nadzorowanego, 235
tempo uczenia się, 29
TensorFlow, 97, 126
testowanie, 320, 342, 343
testy A/B, 340
Theano, 96
  warstwy konwolucyjne, 148
trening, 304, 306, 316, 317
twierdzenie o uniwersalnej aproksymacji, 25, 59
tworzenie głębokiej sieci konwolucyjnej, 259

U

UCB, Upper Confidence Bounds, 215
uczenie
  adaptacyjne, 308, 342
    Adadelta, 312
    Adagrad, 311
    metoda Newtona, 310
    pęd, 309
    wyżarzanie tempa, 309
  bazujące
    na modelach, 266
    na strategii, 266
    na wartościach, 266
  cech nienadzorowane, 81, 105
  funkcji wartości, 209
  głębokie, 28, 39–42, 77, 78
    algorytmy, 88
    H2O, 315
    przeszukiwanie drzewa Monte Carlo, 220
    w grach komputerowych, 235
    w grach planszowych, 195
    warstwy konwolucyjne, 147
    zastosowania, 89
  maszynowe, 17–19, 24, 41, 48, 278
    cel, 25
    dane szkoleniowe, 25
    efekt, 25
    przetwarzanie danych, 25
    reprezentacja, 25
    tworzenie przypadku testowego, 25
    uczeń, 25
    zbieranie danych, 25
  nadzorowane, 19, 23, 28, 231, 235
  nienadzorowane, 22, 28
  przez wzmacnianie, 19, 23, 28, 36, 222, 231, 246
  rozproszone, 314
ukryte modele Markova, 186
uogólniony estymator korzyści, 267

V

V, 231

W

wagi, 301, 304–308, 342
walidacja, 321, 334
  krzyżowa, 326
  modelu, 321, 326
  z wykorzystaniem krzywej ROC, 329
warstwy
  gęste, 99
  konwolucyjne, 78, 138
    krok, 144
    rozpoznawanie cyfr, 150
    w bibliotece Theano, 148
    w uczeniu głębokim, 147
    wypełnienie, 144
    zbiór danych CIFAR10, 153

Kup ksi k
Pole ksi k

warstwy
  pooling, 145
  środkowe, 111
  ukryte, 79, 111
    dobór liczby, 110
  wyjściowe, 110
wartość
  MSE, 291
  NaN, 165
wdrażanie, 343, 349
wektor wejściowy, 35
widmo Mela, 185
wolumen, 142
wskaźnik
  istotności, 339
  KPI, 338
współbieżny algorytm SGD, 306
współczynnik
  fałszywych alarmów, 329
  MFCC, 186
wstępne przetwarzanie, 177
wykrywanie
  anomalii, 273
    nadzorowane, 280
    nienadzorowane, 280
    seminadzorowane, 280
    stosowanie autoenkoderów, 281, 285
  cyfr, 286
  elementów odstających, 274
  włamań, 301
wynik ataku, 339
wypełnienie, 144
wyznaczanie gradientów, 223

Z

zadanie odwróconego wahadła, 246
zastosowania
  algorytmów genetycznych, 237
  uczenia głębokiego, 89
zastosowanie sieci neuronowej, 68
  autonomiczne samochody, 69
  generowanie mowy, 70
  modelowanie funkcji XOR, 70
  przetwarzanie sygnałów, 69
  rozpoznawanie
    i klasyfikacja obiektów, 91
    cyfr, 150
    mowy, 90
    wzorców, 70
  system Instant physician, 69
  wykrywania anomalii, 277
zbiór danych
  CIFAR10, 153
  MNIST, 22, 98
zestaw danych MNIST, 286
znikający gradient, 58

Kup ksi k
Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

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

Uczenie głębokie: zajrzyj w przyszłość programowania!

Na naszych oczach dokonuje się przełom: technologie wykorzystujące rozmaite formy sztucznej inteligencji zaczynają się pojawiać w różnych branżach. Niektórzy nawet nie zdają sobie sprawy, jak często i jak powszechnie stosuje się algorytmy uczenia głębokiego. Możliwości w tym zakresie stale rosną. Wzrasta też zapotrzebowanie na inżynierów, którzy swobodnie operują wiedzą o uczeniu głębokim i są w stanie zaimplementować potrzebne algorytmy w konkretnym oprogramowaniu. Uczenie głębokie jest jednak dość złożonym zagadnieniem, a przyswojenie potrzebnych umiejętności wymaga wysiłku.

Ta książka stanowi doskonałe wprowadzenie w temat uczenia głębokiego. Wyjaśniono tu najważniejsze pojęcia uczenia maszynowego. Pokazano, do czego mogą się przydać takie narzędzia jak pakiet scikit-learn, biblioteki Theano, Keras czy TensorFlow. Ten praktyczny przewodnik znakomicie ułatwi zrozumienie zagadnień rozpoznawania wzorców, dokładnego skalowania danych, pozwoli też na rzetelne zapoznanie się z algorytmami i technikami uczenia głębokiego. Autorzy zaproponowali wykorzystanie w powyższych celach języka Python — ulubionego narzędzia wielu badaczy i pasjonatów nauki.

W książce między innymi:

- solidne podstawy uczenia maszynowego i sieci neuronowych
- trening systemów sztucznej inteligencji w grach komputerowych
- rozpoznawanie obrazów
- rekurencyjne sieci neuronowe w modelowaniu języka
- budowa systemów wykrywania oszustw i włamań

Dr Valentino Zocca opracował wiele algorytmów matematycznych i modeli prognostycznych dla firmy Boeing. Obecnie jest konsultantem w branży finansowej.
Gianmario Spacagna pracuje w firmie Pirelli, gdzie buduje systemy maszynowego uczenia się i kompletne rozwiązania do produktów informacyjnych.
Daniel Slater tworzył oprogramowanie do oceny ryzyka dla branży finansowej. Obecnie zajmuje się systemami do przetwarzania dużych ilości danych i analizy zachowań użytkowników.
Peter Roelants specjalizuje się w stosowaniu technik uczenia głębokiego do badań spektralnych obrazów, rozpoznawania mowy czy ekstrakcji danych z dokumentów.

ISBN 978-83-283-4173-9
9 788328 341739
Cena: 67,00 zł