O'REILLY®

Głębokie uczenie z TensorFlow

OD REGRESJI LINIOWEJ PO UCZENIE PRZEZ WZMACNIANIE

Bharath Ramsundar
Reza Bosagh Zadeh

Helion

Tytuł oryginału: TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning

Tłumaczenie: Leszek Sagalara

ISBN: 978-83-283-5705-1

Authorized Polish translation of the English edition of TensorFlow for Deep Learning
ISBN 9781491980453 © 2018 Reza Zadeh, Bharath Ramsundar

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 3.

Regresja liniowa i logistyczna z TensorFlow

W tym rozdziale dowiesz się, jak budować proste, ale nietrywialne przykłady systemów uczenia w TensorFlow. W pierwszej części tego rozdziału omówiono matematyczne podstawy budowania systemów uczenia, w szczególności funkcje, ciągłość i różniczkowalność. Przedstawimy ideę funkcji straty, a następnie wyjaśnimy, w jaki sposób uczenie maszynowe sprowadza się do zdolności do znalezienia minimalnych punktów skomplikowanych funkcji straty. Następnie omówimy metodę gradientu prostego i wyjaśnimy, jak można ją wykorzystać do minimalizacji funkcji straty. Pierwszy podrozdział zakończymy krótkim przedstawieniem algorytmicznej koncepcji automatycznego różniczkowania. Drugi podrozdział skupia się na wprowadzeniu koncepcji TensorFlow opartych na tych ideach matematycznych. Koncepcje te obejmują węzły zastępcze, zakresy, optymalizatory i TensorBoard oraz umożliwiają praktyczne konstruowanie i analizę systemów uczenia. Ostatni podrozdział zawiera studia przypadków dotyczące treningu modeli regresji liniowej i logistycznej w TensorFlow.

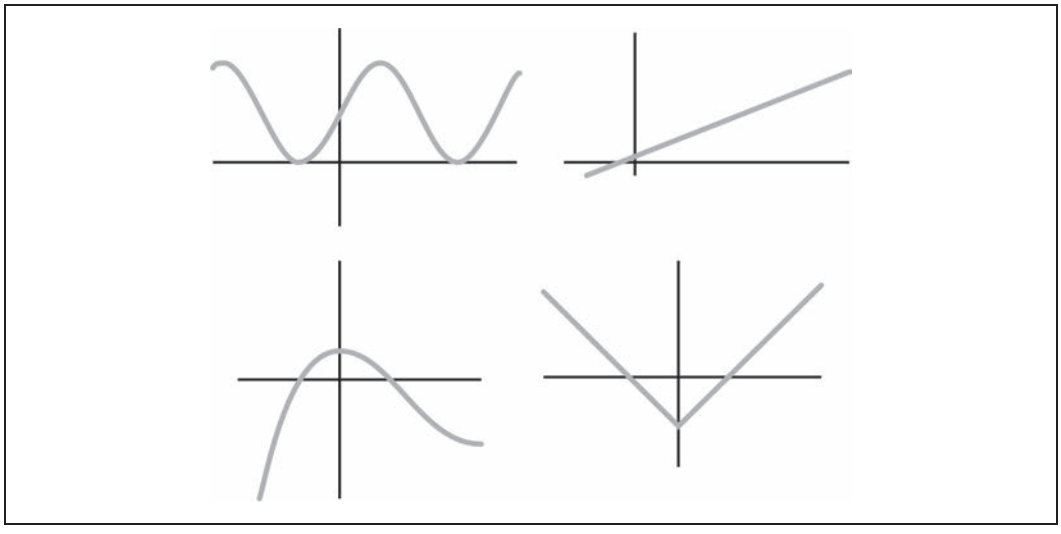
Ten rozdział jest długi i wprowadza wiele nowych koncepcji. Nic się nie stanie, jeśli nie zrozumiesz wszystkich niuansów podczas pierwszego czytania. Radzimy przejść dalej i wrócić tu później, aby odnieść się do tych koncepcji, gdy zajdzie taka konieczność. Będziemy wielokrotnie wykorzystywać te podstawy w pozostałej części książki, aby umożliwić ich stopniowe przyswajanie.

Przegląd matematyczny

Pierwszy podrozdział zawiera przegląd narzędzi matematycznych niezbędnych do koncepcyjnego zrozumienia uczenia maszynowego. Staramy się zminimalizować liczbę wymaganych greckich symboli, skupiając się raczej na kształtowaniu zrozumienia pojęć, a nie na manipulacjach technicznych.

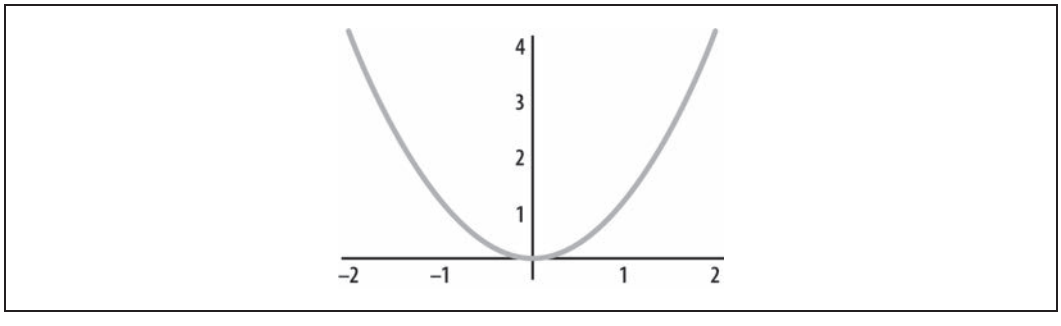
Funkcje i różniczkowalność

W tym podrozdziale przedstawiamy krótki przegląd koncepcji funkcji i różniczkowalności. Funkcja *f* jest regułą, która przenosi dane wejściowe na wyjście. Funkcje istnieją we wszystkich językach programowania komputerowego, a matematyczna definicja funkcji nie różni się zbytnio. Funkcje matematyczne wykorzystywane powszechnie w fizyce i technice mają jednak inne ważne właściwości, takie jak ciągłość i różniczkowalność. Ogólnie mówiąc, funkcja ciągła to taka, którą można narysować bez odrywania ołówka od papieru, jak pokazano na rysunku 3.1 (nie jest to oczywiście ścisła definicja, ale odzwierciedla ona ducha warunku ciągłości).



Rysunek 3.1. Przykłady funkcji ciągłych

Różniczkowalność to rodzaj stanu gładkości funkcji. W takiej funkcji niedopuszczalne są ostre narożniki ani załamania (rysunek 3.2).



Rysunek 3.2. Funkcja różniczkowalna

Główną zaletą funkcji różniczkowalnych jest to, że możemy wykorzystać nachylenie funkcji w danym punkcie jako wskazówkę do znalezienia miejsc, w których funkcja przyjmuje wartości większe lub mniejsze od wartości funkcji w aktualnej pozycji. Ułatwia to znajdowanie **minimów** funkcji. **Pochodna** funkcji różniczkowalnej *f*, oznaczana *f'*, jest kolejną funkcją, która dostarcza nachylenia pierwotnej funkcji we wszystkich punktach. W tej koncepcji chodzi o to, że pochodna funkcji w danym punkcie stanowi drogowskaz wskazujący kierunki, w których funkcja przyjmuje wartości większe lub mniejsze od jej aktualnej wartości. Algorytm optymalizacji może podążać za tym drogowskazem w kierunku minimum *f*. W samych minimach pochodna funkcji jest równa zeru.

Siła optymalizacji opartej na pochodnych nie jest zupełnie oczywista. Pokolenia adeptów analizy matematycznej cierpiały z powodu żmudnych ćwiczeń, szukając minimów stosunkowo nieskomplikowanych funkcji na papierze. Takie ćwiczenia mają na ogół niewielką wartość praktyczną ponieważ analityczne znalezienie minimów funkcji z niewielką liczbą parametrów wejściowych jest możliwe

tylko w najprostszych przypadkach, które dzisiaj i tak najwygodniej przeprowadzić graficznie. Siła optymalizacji opartej na pochodnych staje się oczywista dopiero wtedy, gdy istnieją setki, tysiące, miliony lub miliardy zmiennych. Przy takiej skali złożoności analityczne przedstawienie funkcji jest prawie niemożliwe, a wszystkie wizualizacje są na tyle ułomne, że nie oddają najistotniejszych właściwości funkcji. W przypadku tak dużej złożoności funkcji **gradient** funkcji ∇f, uogólnienie f do funkcji wielu zmiennych, jest prawdopodobnie najpotężniejszym narzędziem matematycznym do badania takich funkcji . Gradienty będziemy omawiać dokładniej w dalszej części tego rozdziału (koncepcyjnie; nie będziemy zajmować się technicznymi szczegółami obliczania gradientów).

Na bardzo wysokim poziomie uczenie maszynowe jest po prostu aktem wyznaczania minimów funkcji: algorytmy uczenia to jedynie mechanizm wyszukiwania minimów dla odpowiednio zdefiniowanych funkcji. Zaletą tej definicji jest matematyczna prostota. Ale czym są te specjalne funkcje różniczkowalne, które kodują w swoich minimach użyteczne rozwiązania, i jak możemy je znaleźć?

Funkcje straty

Aby rozwiązać określony problem dotyczący uczenia maszynowego, analityk danych musi znaleźć sposób na skonstruowanie funkcji, której minima kodują rozwiązania danego problemu w realnym świecie. Na szczęście dla naszego nieszczęsnego analityka, literatura z zakresu uczenia maszynowego zgromadziła bogatą historię **funkcji straty** (ang. *loss functions*), które wykonują takie kodowania. W praktyce uczenie maszynowe sprowadza się do zrozumienia różnych rodzajów dostępnych funkcji straty oraz wiedzy na temat tego, która z nich powinna być zastosowana dla danego problemu. Innymi słowy, funkcja straty jest mechanizmem, za pomocą którego projekt z zakresu przetwarzania danych jest przekształcany w matematykę. Całe uczenie maszynowe i wiele ze sztucznej inteligencji sprowadza się do stworzenia właściwej funkcji straty w celu rozwiązania problemu. Omówimy teraz kilka typowych rodzin funkcji straty.

Zaczniemy od stwierdzenia, że funkcja straty L musi posiadać pewne właściwości matematyczne, aby była użyteczna. Po pierwsze, L musi używać zarówno punktów danych x, jak i etykiet y. Oznaczamy to, zapisując funkcję straty jako $L(x, y)$. Używając naszego języka z poprzedniego rozdziału, zarówno x, jak i y są tensorami, a L jest funkcją przekształcającą pary tensorów na skalary. Jak taka funkcja straty powinna wyglądać? Powszechnie przyjmuje się założenie, że funkcje straty są addytywne. Załóżmy, że (x_i, y_i) są danymi dostępnymi dla naszego przykładu oraz że istnieje ogółem N przykładów. Wówczas funkcję straty można rozłożyć jako

$$L(x, y) = \sum_{i=1}^{N} L_i(x_i, y_i)$$

(w praktyce L_i jest taka sama dla każdego punktu danych). Ten addytywny rozkład zapewnia wiele korzyści. Pierwszą z nich jest obliczanie pochodnej jako sumy pochodnych poszczególnych składników, więc obliczenie gradientu całkowitej straty można sprowadzić do następującej formuły:

$$\nabla L(x, y) = \sum_{i=1}^{N} \nabla L_i(x_i, y_i).$$

Ta matematyczna sztuczka oznacza, że dopóki funkcje składowe L_i są różniczkowalne, to samo będzie dotyczyć również całkowitej funkcji straty. Wynika z tego, że problem projektowania funkcji straty przekłada się na problem projektowania mniejszych funkcji $L_i(x_i, y_i)$. Zanim zabierzemy się za projektowanie L_i, dobrze będzie zrobić małą dygresję, która wyjaśni różnicę między problemami klasyfikacji i regresji.

Klasyfikacja i regresja

Algorytmy uczenia maszynowego można ogólnie sklasyfikować jako problemy nadzorowane lub nienadzorowane. Problemy nadzorowane to te, dla których dostępne są zarówno punkty danych *x*, jak i etykiety *y*, a problemy nienadzorowane mają tylko punkty danych *x* bez etykiet *y*. Ogólnie rzecz biorąc, nienadzorowane uczenie maszynowe jest o wiele trudniejsze i słabiej określone (co to znaczy „rozumieć" punkty danych *x*?). W tym momencie nie będziemy zagłębiać się w nienadzorowane funkcje straty, ponieważ w praktyce większość nienadzorowanych strat to sprytnie przetworzone straty nadzorowane.

Nadzorowane uczenie maszynowe można podzielić na dwa podproblemy: klasyfikacji i regresji. Problem klasyfikacji to ten, w którym staramy się zaprojektować system uczenia mechanicznego, przypisujący danemu punktowi danych dyskretną etykietę, powiedzmy 0/1 (lub bardziej ogólnie $0, \cdots, n$). Regresja to problem projektowania systemu uczenia maszynowego, który do danego punktu danych przypisuje etykietę o wartości rzeczywistej (w $\mathbb{R}$).

Na wysokim poziomie problemy te mogą wydawać się dość odmienne. Obiekty dyskretne i ciągłe są zazwyczaj inaczej postrzegane przez matematykę i zdrowy rozsądek. Jednak częścią podstępu stosowanego w uczeniu maszynowym jest wykorzystanie ciągłych, różniczkowalnych funkcji straty do kodowania zarówno problemów klasyfikacji, jak i regresji. Jak już wcześniej wspominaliśmy, wiele z uczenia mechanicznego to po prostu sztuka przekształcania skomplikowanych systemów świata rzeczywistego w odpowiednio proste funkcje różniczkowalne.

W kolejnych punktach przedstawimy parę funkcji matematycznych, które okażą się bardzo przydatne w przekształcaniu zadań klasyfikacji i regresji w odpowiednie funkcje straty.

Funkcja straty L^2

Funkcja straty L^2 (wymawiana jako *el-dwa*) jest powszechnie stosowana do problemów regresji. Funkcja straty L^2 (określana powszechnie jako norma L^2) jest miarą długości wektora:

$$\|a\|_2 = \sqrt{\sum_{i=1}^{N} a_i^2}.$$

Tutaj przyjmuje się, że *a* jest wektorem o *N* składowych. Norma L^2 jest powszechnie stosowana do określenia odległości pomiędzy dwoma wektorami:

$$\|a - b\|_2 = \sqrt{\sum_{i=1}^{N} (a_i - b_i)^2}.$$

Idea L^2 jako pomiaru odległości jest bardzo przydatna do rozwiązywania problemów regresji w nadzorowanym uczeniu maszynowym. Załóżmy, że *x* jest zbiorem danych, a *y* to związane z nimi

etykiety. Niech *f* będzie jakąś funkcją różniczkowalną, która koduje nasz model uczenia maszynowego. Następnie, aby skłonić *f* do przewidywania *y*, tworzymy funkcję straty L^2:

$$L(x, y) = \| f(x) - y \|_2 .$$

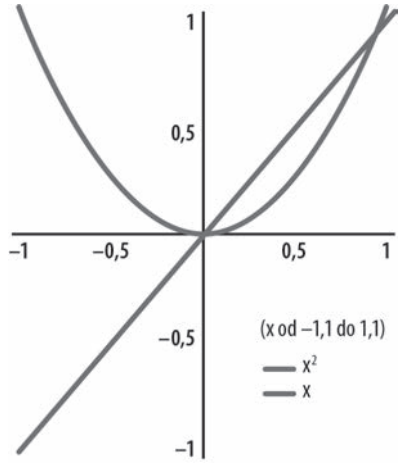
Należy zauważyć, że w praktyce często zamiast bezpośredniego wykorzystywania funkcji straty L^2 stosuje się raczej jej kwadrat:

$$\|a - b\|_2^2 = \sum_{i=1}^{N} (a_i - b_i)^2 ,$$

aby uniknąć stosowania czynnika typu $1/\sqrt{(x)}$ w gradiencie. W pozostałej części tego rozdziału i książki będziemy wielokrotnie używać kwadratu funkcji straty L^2.

Słabości funkcji straty L^2

Funkcja straty L^2 ostro karze duże odchylenia od prawdziwych etykiet, ale nie sprawdza się dobrze w nagradzaniu dokładnych dopasowań dla etykiet o rzeczywistej wartości. Rozbieżność tę możemy zrozumieć matematycznie, badając zachowanie funkcji x^2 i x w pobliżu początku układu współrzędnych (rysunek 3.3).



Rysunek 3.3. Porównanie funkcji kwadratowej i tożsamościowej w pobliżu początku

Zauważ, jak x^2 zmniejsza się gwałtownie do 0 dla małych wartości *x*. W efekcie małe odchylenia nie są mocno karane przez funkcję straty L^2. W regresji małowymiarowej nie stanowi to poważnego problemu, ale w regresji wielowymiarowej L^2 staje się słabą funkcją straty, ponieważ może istnieć wiele małych odchyleń, które łącznie sprawią, że wydajność regresji będzie słaba. Przykładowo, w predykcji obrazu strata L^2 tworzy rozmyte obrazy, które nie są atrakcyjne wizualnie. Ostatnie postępy w nauce maszynowej doprowadziły do opracowania sposobów uczenia się funkcji strat. Te wyuczone funkcje strat, powszechnie nazywane generatywnymi sieciami kontradyktoryjnymi (GAN — ang. *Generative Adversarial Networks*), są o wiele bardziej odpowiednie dla regresji wielowymiarowej i potrafią generować obrazy pozbawione rozmyć.

Rozkłady prawdopodobieństwa

Przed wprowadzeniem funkcji straty dla problemów klasyfikacji warto poświęcić chwilę na zapoznanie się z rozkładami prawdopodobieństwa. Zacznijmy od tego, czym jest rozkład prawdopodobieństwa i dlaczego powinniśmy się nim przejmować w uczeniu maszynowym. Prawdopodobieństwo jest skomplikowanym zagadnieniem, więc zagłębimy się w nie tylko na tyle, abyś mógł je zrozumieć na minimalnym wymaganym poziomie. Na wysokim poziomie rozkłady prawdopodobieństwa oferują matematyczny trik, który pozwala na rozluźnienie dyskretnego zbioru wyborów w kontinuum. Załóżmy na przykład, że musisz zaprojektować system uczenia maszynowego, który przewiduje, czy przy rzucie monetą wypadnie orzeł, czy reszka. Nie wydaje się, aby wybór orzeł/reszka można było zakodować jako funkcję ciągłą, a tym bardziej różniczkowalną. W jaki sposób możesz więc wykorzystać maszynerię obliczeniową lub TensorFlow do rozwiązywania problemów związanych z dyskretnymi wyborami?

Wprowadź rozkład prawdopodobieństwa. Spraw, aby zamiast dokonywania sztywnych wyborów klasyfikator przewidział prawdopodobieństwo tego, że wypadnie orzeł lub reszka. Dla przykładu, klasyfikator może nauczyć się przewidywać, że prawdopodobieństwo wyrzucenia reszki wynosi 0,75, a orła 0,25. Zauważ, że prawdopodobieństwo zmienia się w sposób ciągły! W związku z tym, pracując z prawdopodobieństwami zdarzeń dyskretnych, a nie z samymi zdarzeniami, możesz zgrabnie odsunąć na bok kwestię, że w rzeczywistości wyliczenia nie sprawdzają się dla zdarzeń dyskretnych.

Rozkład prawdopodobieństwa p jest po prostu listą prawdopodobieństw dla możliwych do wystąpienia zdarzeń dyskretnych. W tym przypadku $p = (0{,}75, 0{,}25)$. Możesz też spotkać się z zapisem $p{:}\{0, 1\} \rightarrow \mathbb{R}$ jako oznaczeniem funkcji przekształcającej zestaw dwóch elementów na liczby rzeczywiste. Taki sposób zapisu może być czasami przydatny.

Przypomnijmy pokrótce, że techniczna definicja rozkładu prawdopodobieństwa jest bardziej skomplikowana. Możliwe jest przypisywanie rozkładów prawdopodobieństwa do zdarzeń o wartościach rzeczywistych. Rozkłady takie omówimy w dalszej części rozdziału.

Entropia krzyżowa

Entropia krzyżowa (ang. *cross-entropy*) jest matematyczną metodą pomiaru odległości pomiędzy dwoma rozkładami prawdopodobieństwa:

$$H(p,q) = -\sum_x p(x)\log\, q(x)$$

Tutaj p i q są dwoma rozkładami prawdopodobieństwa. Zapis $p(x)$ oznacza prawdopodobieństwo p względem zdarzenia x. Ta definicja jest warta dokładnego omówienia. Podobnie jak norma L^2, H zapewnia pojęcie odległości. Zauważ, że w przypadku gdy $p = q$,

$$H(p,p) = -\sum_x p(x)\log\, p(x)$$

Wielkość ta jest entropią p i zazwyczaj jest zapisywana po prostu jako $H(p)$. Jest to miara tego, jak nieuporządkowany jest rozkład; entropia jest zmaksymalizowana, gdy wszystkie zdarzenia są równie prawdopodobne. Wartość $H(p)$ jest zawsze mniejsza lub równa $H(p, q)$. W rzeczywistości im rozkład q

jest „dalej" od p, tym większa jest entropia krzyżowa. Nie będziemy zagłębiać się dokładnie w znaczenia tych stwierdzeń, ale warto pamiętać o tym, że entropia krzyżowa jest miarą odległości.

Na marginesie należy zauważyć, że w przeciwieństwie do normy L^2, H jest asymetryczne! Oznacza to, że $H(p, q) \neq H(q, p)$. Z tego powodu rozważania z wykorzystaniem entropii krzyżowej mogą być nieco skomplikowane i powinny być przeprowadzane z pewną ostrożnością.

Wracając do konkretów, załóżmy teraz, że $p = (y, 1–y)$ jest prawdziwym rozkładem danych dla systemu dyskretnego z dwoma wynikami, a $q = (y_{pred}, 1–y_{pred})$ to przewidywania systemu uczenia maszynowego. Wtedy strata entropii krzyżowej to

$$H(p,q) = y \log y_{pred} + (1-y)\log(1-y_{pred}).$$

Ta forma straty jest szeroko stosowana w systemach uczenia maszynowego do szkolenia klasyfikatorów. Empirycznie minimalizowanie $H(p, q)$ wydaje się konstruować klasyfikatory, które dobrze odtwarzają dostarczone etykiety szkoleniowe.

Metoda gradientu prostego

Do tej pory w tym rozdziale dowiedziałeś się o pojęciu wyszukiwania minimów funkcji jako mechanizmu pośredniczącego dla uczenia maszynowego. Dla przypomnienia, minimalizowanie odpowiedniej funkcji jest często wystarczające, aby nauczyć maszynę, jak rozwiązać określone zadanie. Aby skorzystać z tego mechanizmu, należy użyć odpowiedniej funkcji straty, takiej jak np. L^2, lub entropii krzyżowej $H(p, q)$ w celu przekształcenia problemów klasyfikacji i regresji w odpowiednie funkcje straty.

Wagi — parametry w procesie uczenia

Do tej pory w tym rozdziale wyjaśniliśmy, że uczenie maszynowe jest aktem minimalizowania odpowiednio zdefiniowanej funkcji straty $L(x, y)$. Oznacza to, że próbujemy znaleźć takie argumenty dla funkcji straty L, które ją minimalizują. Jednak uważni czytelnicy przypomną sobie, że (x, y) są stałymi wielkościami, których nie można zmienić. Jakie argumenty dla L zmieniamy więc podczas uczenia?

Wprowadzamy parametry uczenia W. Załóżmy, że $f(x)$ jest funkcją różniczkowalną, którą chcielibyśmy dopasować do naszego modelu uczenia maszynowego. Zdecydujemy, że f będzie *dopasowywane do danych* przez wybór W. Oznacza to, że nasza funkcja ma właściwie dwa argumenty $f(W, x)$. Ustalenie wartości W da nam funkcję, która zależy wyłącznie od punktów danych x. Te właśnie parametry są wielkościami faktycznie wybieranymi w procesie minimalizacji funkcji straty. W dalszej części rozdziału zobaczymy, jak można wykorzystać TensorFlow do kodowania wag przy użyciu zmiennej `tf.Variable`.

Załóżmy teraz, że zakodowaliśmy już nasz problem za pomocą odpowiedniej funkcji straty. Jak w praktyce możemy znaleźć jej minima? Kluczową sztuczką, którą zastosujemy, jest minimalizacja metodą gradientu prostego (ang. *gradient descent*). Załóżmy, że f jest funkcją, która zależy od pewnych wag W. Wówczas ∇W oznacza kierunek zmiany w W, który maksymalnie zwiększałby f. Wynika z tego, że wykonanie kroku w przeciwnym kierunku przybliżyłoby nas do minimum f.

Zapis gradientów

Gradient dla wag W zapisaliśmy jako ∇W. Czasami jednak wygodnie będzie stosować dla gradientu następujący zapis:

$$\nabla W = \frac{\partial L}{\partial W}.$$

To równanie można odczytać jako stwierdzenie, że gradient ∇W koduje kierunek, który maksymalnie zmienia stratę L.

Ideą metody gradientu prostego jest znalezienie minimów funkcji poprzez wielokrotne podążanie w kierunku przeciwnym niż ten, który wskazuje gradient. Algorytmicznie ta reguła aktualizacji wartości wag może być wyrażona jako

$$W = W - \alpha \nabla W,$$

gdzie α jest **wielkością kroku** i decyduje o tym, na ile istotny jest gradient ∇W. Idea polega na tym, aby wykonać wiele małych kroków zmierzających w kierunku przeciwnym niż ten wskazywany przez ∇W. Zwróć uwagę, że samo ∇W jest funkcją W, więc rzeczywisty krok zmienia się przy każdej iteracji. Każdy krok wykonuje małą aktualizację macierzy wag W. Iteracyjny proces wykonywania aktualizacji jest zazwyczaj nazywany **uczeniem** macierzy wag W.

Efektywne obliczanie gradientów za pomocą minigrup

Jednym z problemów jest to, że obliczanie ∇W może być bardzo powolne. Z założenia ∇W zależy od funkcji straty L. Ponieważ L zależy od całego zbioru danych, przetwarzanie ∇W przy dużych zbiorach danych może stać się bardzo wolne. W praktyce zazwyczaj szacuje się ∇W na podstawie fragmentu zbioru danych zwanego **minigrupą** (ang. *minibatch*). Każda minigrupa ma zazwyczaj rozmiar 50 – 100. Rozmiar minigrupy jest **hiperparametrem** w algorytmie uczenia głębokiego. Kolejnym hiperparametrem jest wielkość kroku dla każdego kroku α. Algorytmy uczenia głębokiego mają zazwyczaj klastry hiperparametrów, które same nie są uczone poprzez stochastyczny spadek wzdłuż gradientu (ang. *stochastic gradient descent*).

Obecność parametrów uczenia i hiperparametrów jest z jednej strony słabością, a z drugiej strony atutem głębokich architektur. Hiperparametry zapewniają wiele miejsca na wykorzystanie intuicji eksperta, a dzięki parametrom uczenia dane mówią same za siebie. Jednak ta elastyczność sama w sobie szybko staje się słabością, ponieważ zrozumienie zachowania hiperparametrów jest czymś w rodzaju czarnej magii, która blokuje początkujących przed szerszym wdrażaniem uczenia głębokiego. W dalszej części książki poświęcimy sporo miejsca na omówienie optymalizacji hiperparametrów.

Zakończymy ten punkt, wprowadzając pojęcie **epoki**. Epoka jest pełnym przejściem algorytmu gradientu prostego przez dane x. W szczególności epoka składa się z tylu kroków gradientu prostego, ile jest wymaganych do przejrzenia wszystkich danych zawartych w minigrupie danego rozmiaru. Załóżmy na przykład, że zbiór zawiera 1000 punktów danych, a szkolenie wykorzystuje minigrupę o rozmiarze 50. Wówczas epoka będzie składała się z 20 aktualizacji metodą gradientu

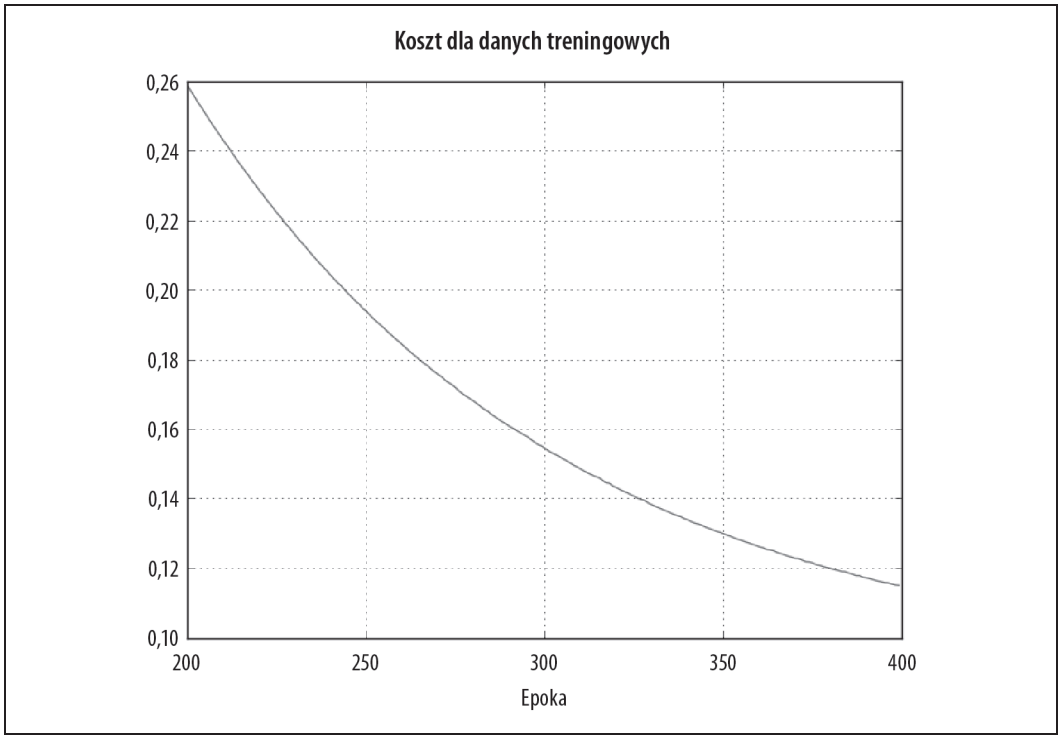
prostego. Każda epoka szkolenia zwiększa ilość użytecznej wiedzy, którą nabył model. Matematycznie będzie to odpowiadało zmniejszeniu wartości funkcji straty na zestawie treningowym.

Wczesne epoki spowodują znaczne spadki w funkcji straty. Proces ten jest często określany jako **uczenie aprioryczne** na zbiorze danych. Choć wydaje się, że model uczy się szybko, w rzeczywistości tylko dostosowuje się do przebywania w tej części przestrzeni parametrów, która jest istotna dla danego problemu. Późniejsze epoki będą odpowiadać znacznie mniejszym spadkom w funkcji straty, ale to właśnie w tych późniejszych epokach często następują znaczące postępy w uczeniu. Kilka epok to zazwyczaj zbyt mało czasu dla nietrywialnego modelu, aby mógł nauczyć się czegokolwiek użytecznego; modele są zwykle szkolone przez okres od 10 do 1000 epok lub do wystąpienia konwergencji. Może się wydawać, że to dużo, należy jednak zauważyć, że liczba wymaganych epok zazwyczaj nie skaluje się w zależności od wielkości zbioru danych. W konsekwencji gradient prosty skaluje się z wielkością danych liniowo, a nie kwadratowo! Jest to jedna z największych zalet metody stochastycznego spadku wzdłuż gradientu w porównaniu z innymi algorytmami uczenia. Bardziej skomplikowane algorytmy mogą wymagać tylko jednego przejścia przez zbiór danych, ale mogą używać programowania całkowitoliczbowego, które skaluje się kwadratowo z liczbą punktów danych. W dobie dużych zbiorów danych takie wydłużenie czasu wykonywania programów jest fatalną słabością.

Śledzenie spadku w funkcji straty jako funkcji liczby epok może być niezwykle użytecznym skrótem wizualnym, który ułatwia zrozumienie procesu uczenia. Wykresy te są często określane jako krzywe straty (patrz rysunek 3.4). Z czasem doświadczony operator może zdiagnozować typowe niepowodzenia w uczeniu za pomocą jednego szybkiego spojrzenia na krzywą straty. W ramach tej książki zwrócimy szczególną uwagę na krzywe straty dla różnych modeli uczenia głębokiego. W szczególności w dalszej części tego rozdziału przedstawimy TensorBoard, potężny pakiet wizualizacyjny, dzięki któremu TensorFlow umożliwia śledzenie takich wielkości jak funkcje straty.

Systemy automatycznego różniczkowania

Uczenie maszynowe jest sztuką definiowania funkcji straty dopasowanych do zbiorów danych, a następnie ich minimalizowania. Aby zminimalizować funkcje straty, musimy obliczyć ich gradienty i użyć algorytmu gradientu prostego w celu iteracyjnej redukcji straty. Nadal jednak musimy przedyskutować, w jaki sposób gradienty są rzeczywiście obliczane. Do niedawna odpowiedź brzmiała: „ręcznie”. Eksperci w dziedzinie uczenia maszynowego wyciągali długopisy i kartki papieru i ręcznie obliczali pochodne macierzy, aby opracować formuły analityczne dla wszystkich gradientów w systemie uczenia. Formuły te były następnie ręcznie kodowane w celu zaimplementowania algorytmu. Proces ten bywał notorycznie przyczyną pomyłek i niejednemu ekspertowi w dziedzinie uczenia maszynowego zdarzyły się w publikowanych pracach i systemach produkcyjnych przypadkowe błędy gradientowe, które nie zostały odkryte przez lata.



Rysunek 3.4. Przykład krzywej straty dla modelu. Zauważ, że krzywa straty pochodzi z modelu trenowanego z prawdziwym gradientem (tzn. nie jest to szacunkowa minigrupa) i w związku z tym jest gładsza niż inne krzywe strat, które napotkasz w dalszej części tej książki

Sytuacja ta uległa znacznej zmianie wraz z powszechną dostępnością silników automatycznego różniczkowania. Systemy takie jak TensorFlow są w stanie automatycznie obliczać gradienty dla niemal wszystkich funkcji strat. To automatyczne różniczkowanie jest jedną z największych zalet TensorFlow i podobnych systemów, ponieważ osoby zajmujące się uczeniem maszynowym nie muszą już być ekspertami w dziedzinie analizy macierzowej. Jednak na wysokim poziomie wciąż warto rozumieć, jak TensorFlow potrafi automatycznie obliczać pochodne złożonych funkcji. Czytelnikom, którzy przeżywali męki na zajęciach wstępnych z analizy matematycznej, przypominamy, że obliczanie pochodnych funkcji to zaskakująco mechaniczny proces. Istnieje szereg prostych reguł, które można zastosować w celu obliczenia pochodnych dla większości funkcji. Dla przykładu:

$$\frac{d}{dx} x^n = nx^{n-1}$$

,

$$\frac{d}{dx} e^x = e^x .$$

Reguły te można połączyć dzięki zasadzie dotyczącej różniczkowania funkcji złożonej:

$$\frac{d}{dx} f(g(x)) = f'(g(x))g'(x),$$

gdzie f oznacza pochodną f, a g pochodną g. Dzięki tym regułom łatwo jest sobie wyobrazić, w jaki sposób można zaprogramować automatyczny mechanizm różniczkujący dla analizy jednowymiarowej. Rzeczywiście, stworzenie takiego silnika różniczkującego jest częstym ćwiczeniem na pierwszym roku programowania w klasach opartych na Lispie. (Okazuje się, że poprawna analiza funkcji jest o wiele trudniejszym problemem niż obliczanie pochodnych. Składnia języka Lisp sprawia, że analizowanie formuł jest czynnością trywialną, podczas gdy w innych językach często łatwiej jest poczekać z przeprowadzeniem tego ćwiczenia do czasu rozpoczęcia zajęć z kompilatorem).

W jaki sposób można rozszerzyć te reguły na analizy wielowymiarowe? Uzyskanie właściwego wyniku jest trudniejsze, ponieważ istnieje więcej liczb do rozważenia. Przykładowo, jeśli mamy $X = AB$, gdzie X, A i B są macierzami, wzór będzie mieć postać

$$\nabla_A L = \frac{\partial L}{\partial A} = \frac{\partial L}{\partial X} B^T = (\nabla_X L) B^T$$

.

Tego rodzaju wzory można łączyć w celu uzyskania systemu symbolicznego różniczkowania dla obliczeń wektorowych i tensorowych.

Uczenie z TensorFlow

W dalszej części tego rozdziału omówimy koncepcje niezbędne do nauki podstawowych modeli uczenia maszynowego z TensorFlow. Zaczniemy od wprowadzenia koncepcji ćwiczebnych zbiorów danych i wyjaśnimy, jak je tworzyć przy użyciu popularnych bibliotek Pythona. Następnie omówimy nowe idee TensorFlow, takie jak węzły zastępcze, słowniki zasilające, zakresy nazw, optymalizatory i gradienty. W następnym punkcie pokażemy, jak używać tych koncepcji do trenowania prostych modeli regresji i klasyfikacji.

Tworzenie ćwiczebnych zbiorów danych

W tym punkcie omówimy, jak tworzyć proste, ale sensowne syntetyczne zestawy danych lub ćwiczebne zestawy danych, które będziemy wykorzystywać do szkolenia prostych nadzorowanych modeli klasyfikacji i regresji.

(Niezwykle) krótkie wprowadzenie do NumPy

Będziemy intensywnie wykorzystywać NumPy w celu definiowania użytecznych zestawów danych. NumPy jest pakietem Pythona, który umożliwia manipulowanie tensorami (określanymi w NumPy jako obiekty `ndarrays`). Listing 3.1 pokazuje niektóre podstawy.

Listing 3.1. Przykłady użycia NumPy

```
>>> import numpy as np
>>> np.zeros((2,2))
array([[0., 0.],
       [0., 0.]])
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Być może zauważyłeś, że manipulowanie obiektami `ndarray` w NumPy wygląda bardzo podobnie do manipulacji tensorami w TensorFlow. Podobieństwo to zostało zaprojektowane celowo przez architektów TensorFlow. Wiele kluczowych funkcji użytkowych TensorFlow ma podobne argumenty i formy jak analogiczne funkcje w NumPy. Dlatego też nie będziemy się starali szczegółowo omawiać NumPy i liczymy na to, że dzięki eksperymentom czytelnicy będą w stanie samodzielnie rozpracować sposób korzystania z tego pakietu. W internecie można znaleźć wiele zasobów, które zapewniają wprowadzenie do NumPy.

Dlaczego ćwiczebne zbiory danych są tak ważne?

W uczeniu maszynowym często krytyczne znaczenie ma nauka poprawnego korzystania z ćwiczebnych zbiorów danych. Jest to trudne zadanie, a jednym z błędów najczęściej popełnianych przez początkujących jest próba zbyt wczesnego uczenia nietrywialnych modeli na złożonych danych. Próby te często kończą się żałosną porażką, a niedoszli adepci uczenia maszynowego zniechęcają się, przekonani, że uczenie maszynowe jest nie dla nich.

Prawdziwym winowajcą nie jest oczywiście student, ale raczej fakt, że rzeczywiste zbiory danych posiadają wiele idiosynkrazji. Doświadczeni specjaliści w zakresie przetwarzania danych przekonali się, że rzeczywiste zbiory danych często wymagają oczyszczenia i wstępnego przetworzenia, zanim staną się zdatne do nauki. Głębokie uczenie potęguje ten problem, ponieważ większość modeli głębokiego uczenia wykazuje notoryczną wrażliwość na nieprawidłowości w danych. Kwestie takie jak szeroki zakres etykiet regresji lub silne wzorce szumu potrafią zakłócić metody oparte na gradiencie prostym, nawet jeśli inne algorytmy uczenia maszynowego (takie jak np. losowe lasy) nie miałyby problemów.

Na szczęście, prawie zawsze można sobie poradzić z tymi kwestiami, ale może to wymagać znacznego zaawansowania ze strony specjalisty ds. danych. Te kwestie wrażliwości są być może największą przeszkodą na drodze do standaryzacji uczenia maszynowego jako technologii. Będziemy wnikliwie analizować strategie oczyszczania danych, ale na razie zalecamy znacznie prostszą alternatywę: wykorzystanie ćwiczebnych zbiorów danych!

Ćwiczebne zbiory danych są kluczowe dla zrozumienia algorytmów uczenia. Przy bardzo prostych, syntetycznych zbiorach danych zbadanie, czy algorytm nauczył się prawidłowej reguły, jest trywialne. W przypadku bardziej złożonych zbiorów danych ocena ta może być bardzo trudna. W związku z tym w pozostałej części tego rozdziału będziemy korzystać tylko z ćwiczebnych zbiorów danych, ponieważ omawiamy podstawy uczenia w oparciu o metodę gradientu prostego za pomocą TensorFlow. W kolejnych rozdziałach zajmiemy się dogłębnie studiami przypadków z wykorzystaniem rzeczywistych danych.

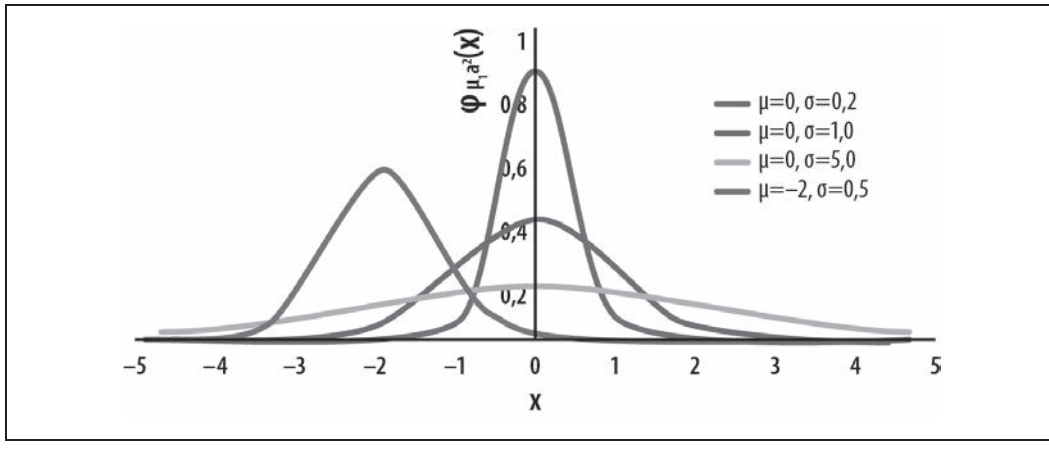
Dodawanie szumu za pomocą rozkładu Gaussa

Wcześniej omawialiśmy dyskretne rozkłady prawdopodobieństwa jako narzędzie do przekształcania dyskretnych wyborów w wartości ciągłe. Napomknęliśmy również o idei ciągłego rozkładu prawdopodobieństwa, ale nie zagłębialiśmy się w szczegóły.

Ciągłe rozkłady prawdopodobieństwa (precyzyjniej zwane funkcjami gęstości prawdopodobieństwa) są użytecznym narzędziem matematycznym do modelowania zdarzeń losowych, które mogą mieć

różny zakres wyników. Do naszych celów wystarczy myśleć o funkcjach gęstości prawdopodobieństwa jako użytecznym narzędziu do modelowania niektórych błędów pomiarowych w gromadzeniu danych. Rozkład Gaussa jest szeroko stosowany do modelowania szumu.

Jak widać na rysunku 3.5, rozkład Gaussa może mieć różne wartości **średniej** μ i **odchylenia standardowego** σ. Średnia rozkładu Gaussa to wynikająca z niego średnia wartość, a odchylenie standardowe jest miarą rozproszenia wartości wokół tej średniej. Ogólnie rzecz biorąc, dodanie zmiennej losowej o rozkładzie Gaussa do jakiejś wielkości stanowi standardowy sposób jej rozmycia, sprawiając, że jest ona nieznacznie zróżnicowana. Jest to bardzo przydatna sztuczka do wymyślania nietrywialnych syntetycznych zbiorów danych.



Rysunek 3.5. Przykłady rozkładów Gaussa o różnych średnich i odchyleniach standardowych

Zwracamy uwagę, że rozkład Gaussa nazywany jest również rozkładem normalnym. Rozkład Gaussa o średniej μ i odchyleniu standardowym σ jest oznaczany jako $N(\mu, \sigma)$. Ta skrócona notacja jest wygodna i będziemy ją stosować wielokrotnie w kolejnych rozdziałach.

Ćwiczebna regresja zbioru danych

Najprostszą formą regresji liniowej jest nauka parametrów dla jednowymiarowej linii. Załóżmy, że nasze punkty danych x są jednowymiarowe. Następnie załóżmy, że rzeczywiste wartości etykiet y są generowane przez regułę liniową

$$y=wx+b$$

Tutaj w i b są parametrami podlegającymi uczeniu, które muszą zostać oszacowane na podstawie danych metodą gradientu prostego. Aby sprawdzić, czy możemy nauczyć się tych parametrów za pomocą TensorFlow, wygenerujemy sztuczny zestaw danych składający się z punktów na linii prostej. Aby utrudnić nieco proces uczenia, do zbioru danych dodamy niewielki szum Gaussa.

Zapiszmy równanie naszej linii zakłócone przez szum Gaussa:

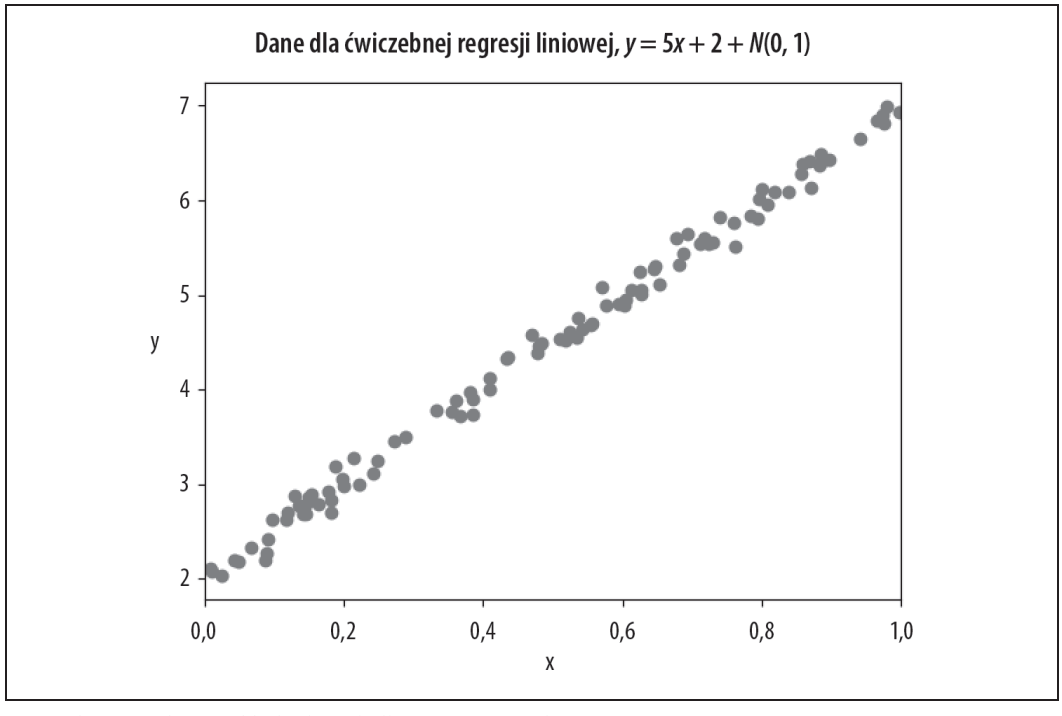
$$y = wx + b + N(0, \varepsilon).$$

Tutaj ε oznacza odchylenie standardowe szumu. Następnie możemy użyć NumPy do wygenerowania sztucznego zbioru danych pobranych z tego rozkładu, jak pokazano w listingu 3.2.

Listing 3.2. Próbkowanie sztucznego zbioru danych przy użyciu NumPy

```
# Generowanie danych syntetycznych
N = 100
w_true = 5
b_true = 2
noise_scale = .1
x_np = np.random.rand(N, 1)
noise = np.random.normal(scale=noise_scale, size=(N, 1))
# Konwersja kształtu y_np na (N,)
y_np = np.reshape(w_true * x_np + b_true + noise, (-1))
```

Dane te wykreśliliśmy przy użyciu Matplotlib na rysunku 3.6 (jeśli chcesz zobaczyć kod użyty do wygenerowania wykresu, znajdziesz go w powiązanym z tą książką repozytorium dostępnym pod adresem *ftp://ftp.helion.pl/przyklady/glutef.zip*) w celu sprawdzenia, czy dane syntetyczne wyglądają rozsądnie. Zgodnie z oczekiwaniami rozkład danych tworzy linię prostą, z niewielką liczbą błędów pomiarowych.



Rysunek 3.6. Wykres rozkładu danych dla regresji ćwiczebnej

Ćwiczebna klasyfikacja zbiorów danych

Nieco trudniej jest stworzyć syntetyczną klasyfikację zbioru danych. Z logicznego punktu widzenia chcemy uzyskać dwie odrębne klasy punktów, które można łatwo rozdzielić. Załóżmy, że zbiór danych składa się tylko z dwóch typów punktów: (–1, –1) i (1, 1). Wówczas algorytm uczenia musiałby nauczyć się reguły, która oddziela te dwie wartości danych.

$$y_0 = (-1, -1)$$

$$y_1 = (1, 1)$$

Tak jak poprzednio, utrudnijmy trochę to zadanie, dodając szum o rozkładzie Gaussa do obu typów punktów:

$$y_0 = (-1, -1) + N(0, \varepsilon),$$

$$y_1 = (1, 1) + N(0, \varepsilon).$$

Jest tu jednak pewna trudność. Nasze punkty leżą na płaszczyźnie, natomiast wprowadzany przez nas wcześniej szum Gaussa jest jednowymiarowy. Na szczęście, istnieje wielowymiarowe rozszerzenie rozkładu Gaussa. Nie będziemy tu omawiać zawiłości wielowymiarowego rozkładu Gaussa, ale nie trzeba ich rozumieć, aby podążać za naszą dyskusją.

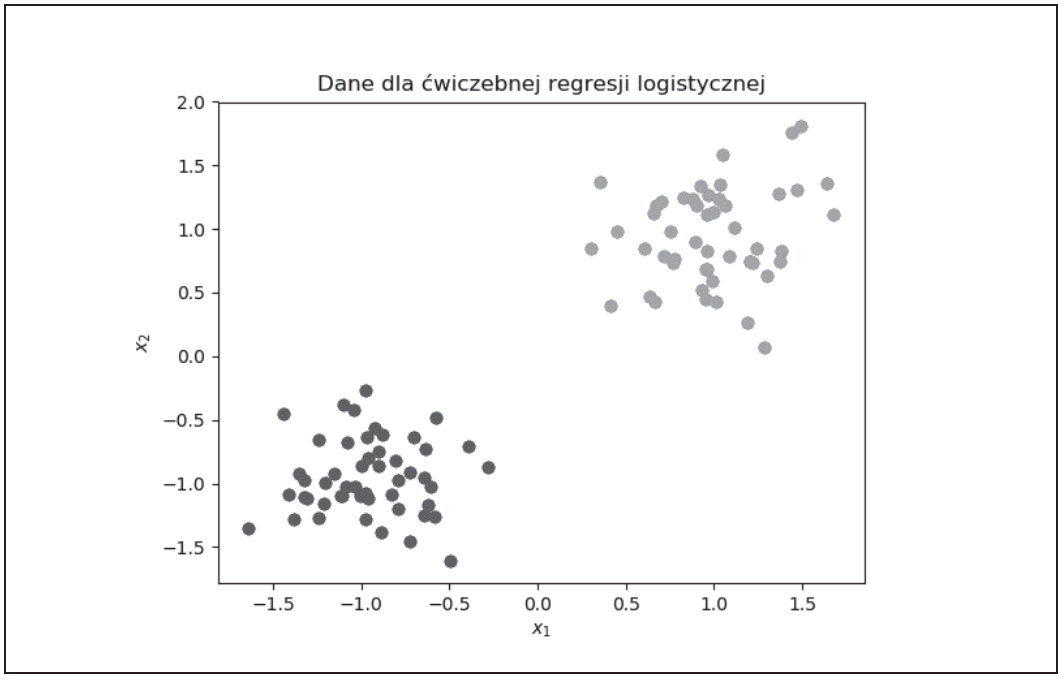
Kod NumPy służący do wygenerowania syntetycznego zbioru danych w listingu 3.3 jest nieco bardziej skomplikowany niż w przypadku regresji liniowej, ponieważ musimy użyć funkcji stosu, np. `vstack`, aby zebrać dwa różne typy punktów danych i skojarzyć je z różnymi etykietami (do łączenia etykiet jednowymiarowych użyjemy pokrewnej funkcji, np. `concatenate`).

Listing 3.3. Przykład ćwiczebnej klasyfikacji zbiorów danych z NumPy

```
# Generowanie danych syntetycznych
N = 100
# Zera tworzą rozkład gaussowski wyśrodkowany na (–1, –1)
# epsilon wynosi .1
x_zeros = np.random.multivariate_normal(
    mean=np.array((-1, -1)), cov=.1*np.eye(2), size=(N/2,))
y_zeros = np.zeros((N/2,))
# Jedynki tworzą rozkład gaussowski wyśrodkowany na (1, 1)
# epsilon wynosi .1
x_ones = np.random.multivariate_normal(
    mean=np.array((1, 1)), cov=.1*np.eye(2), size=(N/2,))
y_ones = np.ones((N/2,))

x_np = np.vstack([x_zeros, x_ones])
y_np = np.concatenate([y_zeros, y_ones])
```

Rysunek 3.7 przedstawia dane wygenerowane przez ten kod za pomocą Matplotlib w celu sprawdzenia, czy rozkład jest zgodny z oczekiwaniami. Widzimy, że dane znajdują się w dwóch klasach, które są starannie rozdzielone.



Rysunek 3.7. Wykres rozkładu danych dla klasyfikacji ćwiczebnej

Nowe koncepcje TensorFlow

Tworzenie prostych systemów uczenia maszynowego w TensorFlow będzie wymagało nauczenia się kilku nowych koncepcji.

Węzły zastępcze

Węzeł zastępczy (ang. *placeholder*) jest sposobem wprowadzania informacji do grafu obliczeń TensorFlow. Pomyśl o węzłach zastępczych jako węzłach wejściowych, przez które informacje są wprowadzane do TensorFlow. Podstawową funkcją używaną do tworzenia węzłów zastępczych jest `tf.placeholder` (listing 3.4).

Listing 3.4. Tworzenie węzła zastępczego TensorFlow

```
>>> tf.placeholder(tf.float32, shape=(2,2))
<tf.Tensor 'Placeholder:0' shape=(2, 2) dtype=float32>
```

Węzły zastępcze będziemy wykorzystywać do wprowadzania punktów danych x i etykiet y do naszych algorytmów regresji i klasyfikacji.

Słowniki zasilające i wyprowadzenia

Przypomnijmy, że w TensorFlow możemy wyliczyć tensory przy użyciu `sess.run(`*`zmienna`*`)`. W jaki sposób wprowadzamy wartości dla węzłów zastępczych w naszych obliczeniach TensorFlow? Odpowiedzią jest skonstruowanie słowników zasilających (ang. *feed dictionaries*). Słowniki zasilające

to słowniki Pythona, które odwzorowują tensory TensorFlow na obiekty `np.ndarray`, zawierające konkretne wartości dla tych węzłów zastępczych. Słownik zasilający najlepiej jest postrzegać jako wejście do grafu obliczeń TensorFlow. Czym zatem jest wyjście? TensorFlow nazywa te dane wyjściowe wyprowadzeniami (ang. *fetches*). Widziałeś już wyprowadzenia. Wykorzystywaliśmy je obszernie w poprzednim rozdziale bez nazywania ich w ten sposób; wyprowadzenie ma postać tensora (lub tensorów), którego wartość jest pobierana z wykresu obliczeniowego po tym, jak obliczenie (przy użyciu węzłów zastępczych ze słownika zasilającego) zostanie zakończone (listing 3.5).

Listing 3.5. Korzystanie z wyprowadzeń

```
>>> a = tf.placeholder(tf.float32, shape=(1,))
>>> b = tf.placeholder(tf.float32, shape=(1,))
>>> c = a + b
>>> with tf.Session() as sess:
       c_eval = sess.run(c, {a: [1.], b: [2.]})
       print(c_eval)
[3.]
```

Zakresy nazw

Skomplikowane programy TensorFlow będą zawierać wiele zdefiniowanych tensorów, zmiennych i pól zastępczych. `tf.name_scope(`*`nazwa`*`)` zapewnia prosty mechanizm ustalania zakresów dla zarządzania tymi zbiorami zmiennych (listing 3.6). Nazwy wszystkich elementów wykresu obliczeniowego utworzonych w ramach wywołania `tf.name_scope(`*`nazwa`*`)` będą poprzedzone członem *nazwa*.

Listing 3.6. Organizowanie węzłów zastępczych za pomocą zakresów nazw

```
>>> N = 5
>>> with tf.name_scope("wezly_zastepcze"):
      x = tf.placeholder(tf.float32, (N, 1))
      y = tf.placeholder(tf.float32, (N,))
>>> x
<tf.Tensor 'wezly_zastepcze/Placeholder:0' shape=(5, 1) dtype=float32>
```

To narzędzie organizacyjne jest najbardziej przydatne w połączeniu z TensorBoard, ponieważ wspomaga system wizualizacji w automatycznym grupowaniu elementów grafu w tym samym zakresie nazw. Więcej na temat TensorBoard dowiesz się w następnym podrozdziale.

Optymalizatory

Podstawowe elementy przedstawione w dwóch ostatnich podrozdziałach podpowiadają już, jak odbywa się uczenie maszynowe w TensorFlow. Wiesz już, jak dodawać węzły zastępcze dla punktów danych i etykiet oraz jak używać operacji tensorowych do definiowania funkcji straty. Brakującym elementem jest posługiwanie się metodą gradientu prostego przy użyciu TensorFlow.

W rzeczywistości możliwe jest definiowanie algorytmów optymalizacyjnych, takich jak gradient prosty, bezpośrednio w Pytonie, przy użyciu podstawowych elementów TensorFlow. TensorFlow zapewnia jednak zbiór algorytmów optymalizacyjnych w module `tf.train`. Algorytmy te mogą być dodawane jako węzły do wykresu obliczeniowego TensorFlow.

Którego optymalizatora powinienem używać?

Moduł `tf.train` oferuje wiele możliwych optymalizatorów, np. `tf.train.GradientDescentOptimizer`, `tf.train.MomentumOptimizer`, `tf.train.AdagradOptimizer`, `tf.train.AdamOptimizer` i wiele innych. Czym się one różnią?

Prawie wszystkie z nich opierają się na idei gradientu prostego. Przypomnijmy sobie zasadę gradientu prostego, którą przedstawiliśmy wcześniej:

$$W = W - \alpha \nabla W$$

Z matematycznego punktu widzenia ta zasada aktualizacji jest prymitywna. Badacze odkryli wiele matematycznych sztuczek umożliwiających przyśpieszenie optymalizacji bez użycia zbyt wielu dodatkowych obliczeń. Na ogół dobrym domyślnym rozwiązaniem jest moduł `tf.train.AdamOptimizer`, który jest stosunkowo solidny. (Wiele metod optymalizacji jest bardzo wrażliwych na wybór hiperparametru. Początkujący powinni unikać trudniejszych metod, dopóki nie zrozumieją dobrze zachowania różnych algorytmów optymalizacyjnych).

Listing 3.7 przedstawia krótki fragment kodu, który dodaje do wykresu obliczeniowego optymalizator minimalizujący predefiniowaną stratę *l*.

Listing 3.7. Dodawanie optymalizatora Adam do grafu obliczeniowego TensorFlow

```
learning_rate = .001
with tf.name_scope("optymalizatory"):
  train_op = tf.train.AdamOptimizer(learning_rate).minimize(l)
```

Stosowanie gradientów z TensorFlow

Wspomnieliśmy wcześniej, że w TensorFlow możliwe jest bezpośrednie wdrożenie algorytmów gradientu prostego. Choć większość przypadków użycia nie wymaga reimplementacji zawartości `tf.train`, warto przyjrzeć się bezpośrednio wartościom gradientu do celów debugowania. `tf.gradients` dostarcza użyteczne narzędzie do tego celu (listing 3.8).

Listing 3.8. Bezpośrednie stosowanie gradientów

```
>>> W = tf.Variable((3,))
>>> l = tf.reduce_sum(W)
>>> gradW = tf.gradients(l, W)
>>> gradW
[<tf.Tensor 'gradients/Sum_grad/Tile:0' shape=(1,) dtype=int32>]
```

Kod ten symbolicznie ściąga gradienty straty *l* z uwzględnieniem modyfikowalnego parametru `W` (`tf.Variable`). `tf.gradients` zwraca listę pożądanych gradientów. Zauważ, że gradienty same w sobie są tensorami! TensorFlow przeprowadza różniczkowanie symboliczne, co oznacza, że same gradienty są częścią wykresu obliczeniowego. Dodatkowym efektem ubocznym symbolicznych gradientów TensorFlow jest możliwość układania pochodnych w stosy. Może to być czasami przydatne w przypadku bardziej zaawansowanych algorytmów.

Tworzenie podsumowań i zapis do plików dla TensorBoard

Bardzo przydatne może być uzyskanie wizualnego wglądu w strukturę programu tensorowego. Zespół TensorFlow dostarcza w tym celu pakiet TensorBoard. TensorBoard uruchamia serwer WWW (działający domyślnie na hoście lokalnym), który wyświetla różne przydatne wizualizacje programu TensorFlow. Aby jednak programy TensorFlow mogły być kontrolowane przez TensorBoard, programiści muszą ręcznie wprowadzić instrukcje zapisu dzienników. Funkcja `tf.train.FileWriter()` określa ścieżkę dostępu do katalogu dzienników programu TensorBoard, a `tf.summary` zapisuje podsumowania różnych zmiennych TensorFlow do podanego katalogu. W tym rozdziale będziemy używać tylko `tf.summary.scalar`, który tworzy podsumowania wielkości skalarnych, aby śledzić wartość funkcji straty. Pomocną funkcją jest `tf.summary.merge_all()`, która dla ułatwienia łączy wiele podsumowań w jedno.

Wycinek kodu przedstawiony w listingu 3.9 dodaje podsumowanie dla straty i określa ścieżkę do katalogu z dziennikami.

Listing 3.9. Dodawanie podsumowania dla straty

```
with tf.name_scope("podsumowania"):
  tf.summary.scalar("strata", l)
  merged = tf.summary.merge_all()

train_writer = tf.summary.FileWriter('/tmp/lr-train', tf.get_default_graph())
```

Uczenie modeli z TensorFlow

Załóżmy teraz, że mamy określone węzły zastępcze dla punktów danych i etykiet oraz zdefiniowaliśmy stratę za pomocą operacji tensorowych. Do wykresu obliczeniowego dodaliśmy węzeł optymalizatora `train_op`, którego możemy użyć do wykonywania kroków gradientu prostego (wprawdzie możemy użyć innego optymalizatora, ale dla wygody będziemy odwoływać się do aktualizacji w formie gradientu prostego). Jak możemy iteracyjnie zastosować metodę gradientu prostego, aby umożliwić uczenie na tym zbiorze danych?

Odpowiedź jest prosta: użyjemy pętli `for` Pythona. W każdej iteracji wykorzystujemy `sess.run()` do pobierania wartości `train_op` wraz z scalonym podsumowaniem operacji `merged` i stratą `l` z wykresu. Przy użyciu słownika zasilającego wprowadzamy wszystkie punkty danych i etykiety do `sess.run()`.

Tę prostą metodę uczenia przedstawia wycinek kodu z listingu 3.10. Zauważ, że dla prostoty pedagogicznej nie używamy tu minigrup. Będziemy z nich korzystać w kolejnych rozdziałach, przy uczeniu na większych zbiorach danych.

Listing 3.10. Prosty przykład uczenia modelu

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Model uczenia
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("krok %d, strata: %f" % (i, loss))
    train_writer.add_summary(summary, i)
```

Uczenie modeli liniowych i logistycznych w TensorFlow

Ten podrozdział łączy wszystkie koncepcje TensorFlow wprowadzone w poprzednim podrozdziale w celu uczenia modeli regresji liniowej i logistycznej na podstawie ćwiczebnych zestawów danych, które opisywaliśmy wcześniej w tym rozdziale.

Regresja liniowa w TensorFlow

W tym punkcie podamy kod do definiowania modelu regresji liniowej w TensorFlow i nauki wag. Jest to proste zadanie, które można łatwo wykonać bez TensorFlow. Niemniej jednak jest to dobre ćwiczenie do zrealizowania w TensorFlow, ponieważ łączy w sobie nowe koncepcje, które wprowadziliśmy w tym rozdziale.

Definiowanie i trening regresji liniowej w TensorFlow

Model regresji liniowej jest prosty:

$$y = wx + b .$$

Tutaj w i b są wagami, których chcemy się nauczyć. Przekształcamy je w obiekty `tf.Variable`, a następnie wykorzystujemy operacje tensorowe do skonstruowania straty L^2:

$$L(x, y) = (y - wx - b)^2 .$$

Kod w listingu 3.11 implementuje te operacje matematyczne w TensorFlow. Wykorzystuje również `tf.name_scope` do grupowania różnych operacji oraz dodaje `tf.train.AdamOptimizer` do nauki i `tf.summary` na użytek TensorBoard.

Listing 3.11. Definiowanie modelu regresji liniowej

```
# Generowanie wykresu tensorflow
with tf.name_scope("wezly_zastepcze"):
  x = tf.placeholder(tf.float32, (N, 1))
  y = tf.placeholder(tf.float32, (N,))
with tf.name_scope("wagi"):
  # Zauważ, że x jest skalarem, więc W jest pojedynczą wagą, której można się nauczyć.
  W = tf.Variable(tf.random_normal((1, 1)))
  b = tf.Variable(tf.random_normal((1,)))
with tf.name_scope("predykcja"):
  y_pred = tf.matmul(x, W) + b
with tf.name_scope("strata"):
  l = tf.reduce_sum((y - y_pred)**2)
# Dodawanie optymalizatorów treningowych
with tf.name_scope("optymalizatory"):
  # Ustawienie współczynnika uczenia na .001 zgodnie z wcześniejszymi zaleceniami.
  train_op = tf.train.AdamOptimizer(.001).minimize(l)
with tf.name_scope("podsumowania"):
  tf.summary.scalar("strata", l)
  merged = tf.summary.merge_all()
```

W listingu 3.12 trenuje się następnie ten model w sposób omówiony wcześniej (bez użycia minigrup).

Listing 3.12. Szkolenie modelu regresji liniowej

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Trenowanie modelu
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("krok %d, strata: %f" % (i, loss))
    train_writer.add_summary(summary, i)
```

Cały kod dla tego przykładu znajduje się w repozytorium związanym z tą książką (*ftp://ftp.helion.pl/przyklady/glutef.zip*). Zachęcamy wszystkich czytelników do uruchomienia pełnego skryptu dla przykładu regresji liniowej, aby mogli osobiście przekonać się, jak działa algorytm uczenia. Przykład ten jest na tyle mały, że do jego uruchomienia nie potrzeba dostępu do żadnego specjalistycznego sprzętu komputerowego.

Przyjmowanie gradientów dla regresji liniowej

Równanie dla systemu liniowego, który modelujemy, to $y = wx+b$, gdzie w i b są wagami do nauczenia. Jak wspomnieliśmy wcześniej, strata dla tego systemu wynosi $L = (y–wx–b)^2$. Niektóre analizy macierzowe mogą zostać użyte do obliczenia gradientów parametrów uczenia bezpośrednio dla w:

$$\nabla w = \frac{\partial L}{\partial w} = -2\left(y - wx - b\right)x^T$$

i dla b

$$\nabla b = \frac{\partial L}{\partial b} = -2\left(y - wx - b\right).$$

Równania te umieszczamy tutaj tylko jako wzmiankę dla ciekawskich czytelników. Nie będziemy systematycznie pokazywać, jak obliczać pochodne funkcji straty, z którymi zetkniemy się w tej książce. Chcielibyśmy jednak zauważyć, że w przypadku skomplikowanych systemów ręczne obliczanie pochodnej funkcji straty pomaga intuicyjnie poznać sposób uczenia sieci głębokiej. Ta intuicja może zapewnić projektantowi skuteczne wskazówki, dlatego zachęcamy zaawansowanych czytelników do samodzielnego zajęcia się tym tematem.

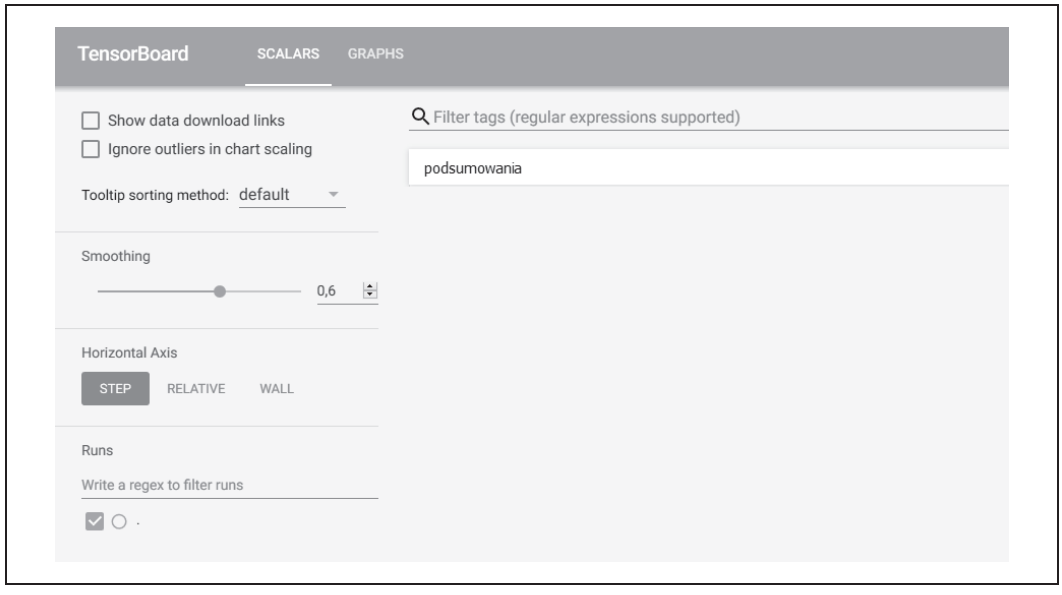
Wizualizacja modeli regresji liniowej z TensorBoard

Model zdefiniowany w poprzednim punkcie wykorzystuje `tf.summary.FileWriter` do zapisu dzienników do katalogu */tmp/lr-train*. Za pomocą polecenia w listingu 3.13 możemy wywołać TensorBoard na tym katalogu z dziennikami (TensorBoard jest domyślnie instalowany wraz z TensorFlow).

Listing 3.13. Wywołanie TensorBoard

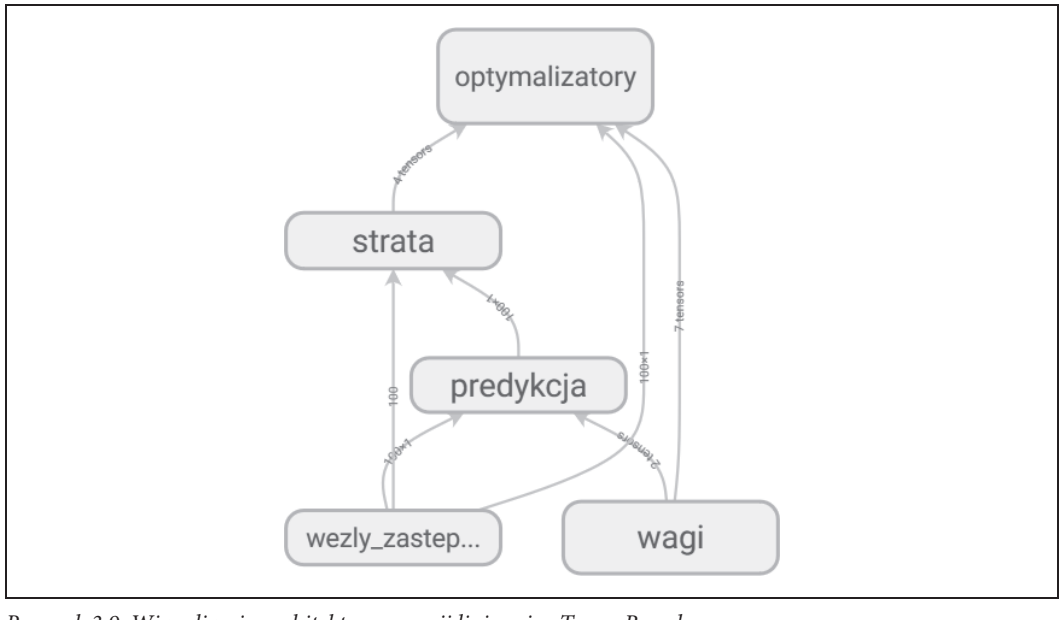
```
tensorboard --logdir=/tmp/lr-train
```

To polecenie uruchamia TensorBoard na porcie podłączonym do lokalnego hosta. Użyj przeglądarki, aby otworzyć ten port. Ekran TensorBoard będzie wyglądał podobnie do tego na rysunku 3.8 (szczegóły mogą się różnić w zależności od wersji TensorBoard).



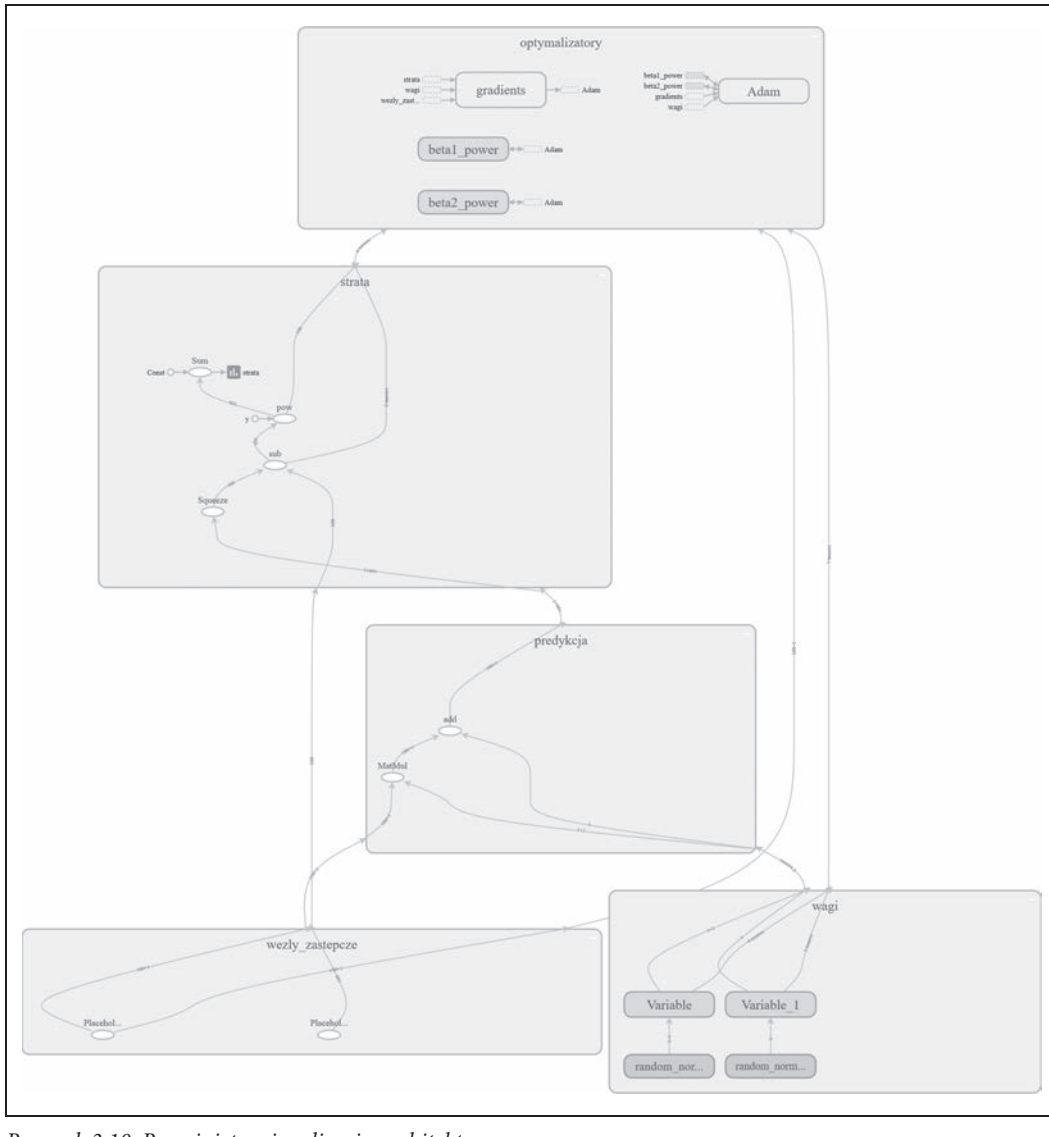
Rysunek 3.8. Zrzut ekranu panelu TensorBoard

Przejdź do zakładki *Graphs*, a zobaczysz wizualizację zdefiniowanej przez nas architektury TensorFlow, co przedstawiono na rysunku 3.9.



Rysunek 3.9. Wizualizacja architektury regresji liniowej w TensorBoard

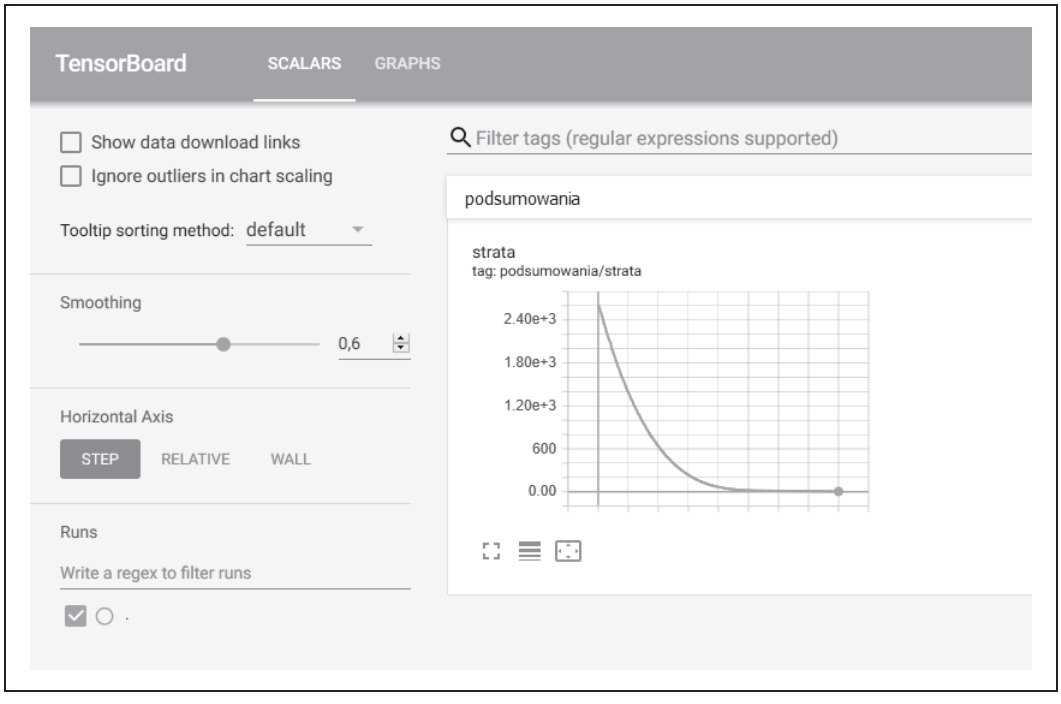
Zwróć uwagę, że wizualizacja ta pogrupowała wszystkie elementy wykresu obliczeniowego należące do różnych zakresów nazw. Poszczególne grupy są ze sobą połączone zgodnie z ich zależnościami na wykresie obliczeniowym. Możesz rozwinąć wszystkie pogrupowane elementy, aby zobaczyć ich zawartość. Rysunek 3.10 przedstawia rozwiniętą architekturę.



Rysunek 3.10. Rozwinięta wizualizacja architektury

Jak widać, jest tu wiele ukrytych węzłów, które nagle stają się widoczne! Funkcje TensorFlow takie jak `f.train.AdamOptimizer` często ukrywają wiele wewnętrznych zmiennych w `tf.name_scope`. Rozwijanie w TensorBoard zapewnia łatwy sposób, aby sprawdzić, co system rzeczywiście tworzy. Chociaż wizualizacja wygląda dość skomplikowanie, większość z tych szczegółów jest ukrytych i nie musisz się tym martwić.

Wróć do strony głównej i otwórz sekcję *podsumowania*. Powinieneś teraz zobaczyć krzywą straty, która będzie przypominać tę na rysunku 3.11. Zwróć uwagę na jej gładki, opadający kształt. Strata spada gwałtownie na początku, podczas nauki wstępnej, następnie spadek słabnie, a krzywa łagodnie opada.



Rysunek 3.11. Przeglądanie krzywej strat w TensorBoard

Style debugowania wizualnego i niewizualnego

Czy użycie narzędzia, jakim jest TensorBoard, jest konieczne do dobrego wykorzystania systemu takiego jak TensorFlow? To zależy. Czy korzystanie z GUI lub interaktywnego debuggera jest konieczne do bycia profesjonalnym programistą?

Różni programiści mają różne style. Niektórzy z nich uważają, że możliwości wizualizacyjne TensorBoard stanowią kluczową część ich pracy przy programowaniu tensorowym. Inni uznają, że TensorBoard nie jest zbytnio użyteczny, i korzystają w większym stopniu z debugowania deklaracji `print`. Oba style programowania tensorowego i debugowania są poprawne, tak samo jak istnieją wspaniali programiści, którzy całkowicie polegają na debuggerach, i tacy, którzy nimi gardzą.

Ogólnie rzecz biorąc, TensorBoard jest bardzo przydatny do debugowania i kształtowania elementarnej intuicji na temat dostępnego zbioru danych. Radzimy, abyś pracował w tym stylu, który najlepiej Ci odpowiada.

Wskaźniki oceny modeli regresji

Do tej pory nie rozmawialiśmy jeszcze o tym, jak ocenić, czy trenowany model czegoś się nauczył. Pierwszą metodą oceny, czy model został przeszkolony, jest spojrzenie na krzywą strat w celu upewnienia się, że ma ona odpowiedni kształt. Jak to zrobić, dowiedziałeś się w poprzednim punkcie. Czego jeszcze powinieneś spróbować?

Teraz chcemy, abyś przyjrzał się **wskaźnikom** związanym z modelem. Wskaźnik jest narzędziem do porównywania etykiet przewidywanych z prawdziwymi. Dla problemów z regresją istnieją dwa popularne wskaźniki: R^2 oraz RMSE (pierwiastek błędu średniokwadratowego — ang. *root-mean-squared error*). R^2 jest miarą korelacji pomiędzy dwiema zmiennymi, która przyjmuje wartości pomiędzy +1 a 0. +1 oznacza korelację doskonałą, a 0 oznacza brak korelacji. W ujęciu matematycznym R^2 dla dwóch zbiorów danych X i Y definiuje się jako

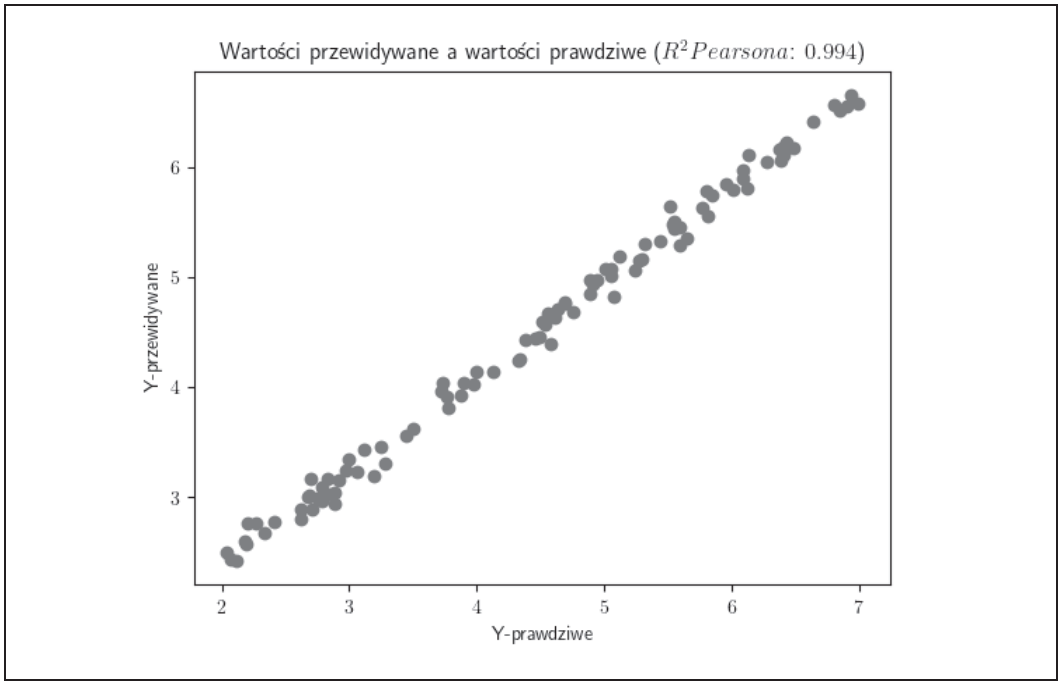
$$R^2 = \frac{\mathrm{cov}(X, Y)^2}{\sigma_X^2 \sigma_Y^2}$$

gdzie cov*(X, Y)* jest kowariancją X i Y, czyli miarą tego, jak oba zbiory danych różnią się od siebie nawzajem, podczas gdy σ_X i σ_Y są odchyleniami standardowymi, miarami tego, jak bardzo różni się każdy z tych zbiorów indywidualnie. Intuicyjnie R^2 mierzy, jak wiele niezależnych odchyleń w każdym zbiorze można wytłumaczyć ich wspólną zmiennością.

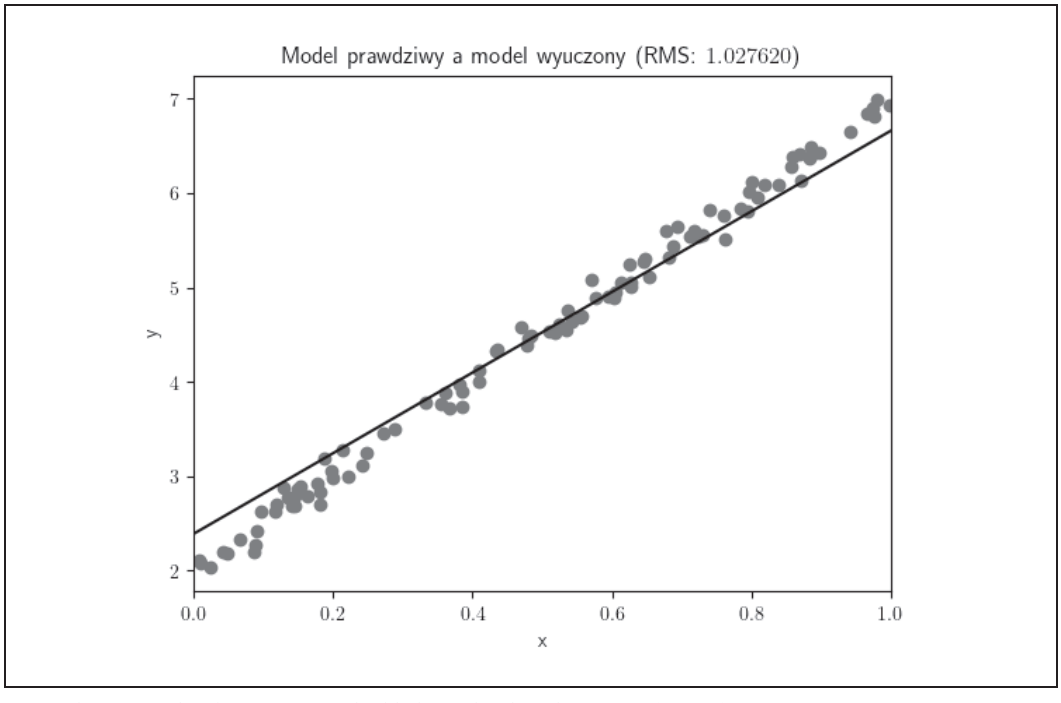
Wiele typów R^2!

Należy zauważyć, że istnieją dwie powszechnie stosowane w praktyce definicje R^2. Częstym błędem początkującego (i eksperta) jest mylenie tych dwóch definicji. W tej książce zawsze będziemy używać kwadratu współczynnika korelacji Pearsona (ang. *Pearson correlation coefficient*) (rysunek 3.12). Druga definicja nazywana jest współczynnikiem determinacji (ang. *coefficient of determination*). Ten drugi typ R^2 jest często bardziej mylący, ponieważ nie posiada dolnej granicy 0, jak ma to miejsce w przypadku kwadratu współczynnika korelacji Pearsona.

Na rysunku 3.12 przewidywane i prawdziwe wartości są silnie skorelowane, z R^2 o wartości bliskiej 1. Wygląda na to, że uczenie wykonało wspaniałą pracę w tym systemie i udało się nauczyć prawdziwej reguły. *Nie tak szybko*. Zauważ, że skala na dwóch osiach na rysunku nie jest taka sama! Okazuje się, że R^2 nie karze za różnice w skali. Aby zrozumieć, co się wydarzyło w tym systemie, musimy rozpatrzyć alternatywny wskaźnik na rysunku 3.13.



Rysunek 3.12. Wykreślenie współczynnika korelacji Pearsona



Rysunek 3.13. Wykreślanie pierwiastka błędu średniokwadratowego (RMSE)

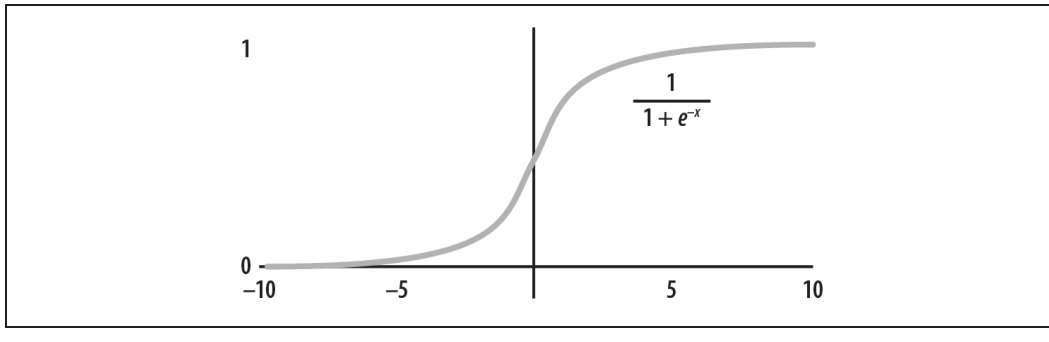
RMSE jest miarą średniej różnicy między wartościami przewidywanymi a rzeczywistymi. Na rysunku 3.13 wykreślamy wartości przewidywane i prawdziwe etykiety jako dwie oddzielne funkcje, używając punktów danych *x* jako naszej osi x. Zauważ, że linia uczenia nie jest prawdziwą funkcją! RMSE jest stosunkowo wysoki i diagnozuje błąd, w przeciwieństwie do R^2, który go nie wykrył.

Co się stało w tym systemie? Dlaczego TensorFlow nie nauczył się prawidłowej funkcji, mimo że został wytrenowany do konwergencji? Ten przykład dobrze ilustruje jedną ze słabości algorytmów gradientu prostego. Nie ma gwarancji znalezienia prawdziwego rozwiązania! Algorytm gradientu prostego może zostać uwięziony w lokalnych minimach. Oznacza to, że może on znaleźć rozwiązania, które wyglądają dobrze, ale w rzeczywistości nie są najniższymi minimami funkcji straty L.

Dlaczego w takim razie w ogóle stosować metodę gradientu prostego? W przypadku prostych systemów często rzeczywiście lepiej jest unikać gradientu prostego i stosować inne algorytmy, które mają większe gwarancje osiągnięcia sukcesu. Jednak w przypadku skomplikowanych systemów, takich jak te, które przedstawimy w kolejnych rozdziałach, nie istnieją jeszcze żadne alternatywne algorytmy, które poradziłyby sobie lepiej niż metoda gradientu prostego. Prosimy, abyście pamiętali o tym fakcie, gdy przejdziemy dalej do głębokiego uczenia.

Regresja logistyczna w TensorFlow

W tym punkcie zdefiniujemy prosty klasyfikator za pomocą TensorFlow. Warto najpierw zastanowić się, czym jest równanie dla klasyfikatora. Powszechnie stosowanym trikiem matematycznym jest wykorzystanie funkcji sigmoidalnej. Sigmoida wykreślona na rysunku 3.14, powszechnie oznaczana symbolem σ, jest funkcją przekształcającą liczby rzeczywiste $\mathbb{R}$ do zakresu (0, 1). Ta właściwość jest przydatna, ponieważ możemy interpretować wyjście sigmoidy jako prawdopodobieństwo zajścia zdarzenia (przekształcanie zdarzeń dyskretnych w wartości ciągłe to powracający motyw w uczeniu maszynowym).



Rysunek 3.14. Wykres funkcji sigmoidalnej

Poniżej prezentujemy równania stosowane do przewidywania prawdopodobieństwa dyskretnej zmiennej 0/1. Równania te definiują prosty model regresji logistycznej:

$$y_0 = \sigma(wx + b),$$

$$y_1 = 1 - \sigma(wx + b).$$

TensorFlow zapewnia gotowe funkcje do obliczania straty entropii krzyżowej (ang. *cross-entropy*) dla wartości sigmoidalnych. Najprostszą z tych funkcji jest `tf.nn.sigmoid_cross_entropy_with_logits`. (Logit jest odwrotnością sigmoidy. W praktyce oznacza to po prostu bezpośrednie przekazanie do TensorFlow argumentu sigmoidy, *wx+b*, zamiast samej wartości sigmoidalnej σ (*wx+b*)). Zalecamy użycie implementacji TensorFlow zamiast ręcznego definiowania entropii krzyżowej, ponieważ występują tu trudne problemy numeryczne, które powstają przy obliczaniu straty entropii krzyżowej.

Listing 3.14 definiuje prosty model regresji logistycznej w TensorFlow.

Listing 3.14. Definiowanie prostego modelu regresji logistycznej

```
# Generowanie wykresu tensorflow
with tf.name_scope("wezly_zastepcze"):
  # Zauważ, że nasze punkty danych x są 2-wymiarowe.
  x = tf.placeholder(tf.float32, (N, 2))
  y = tf.placeholder(tf.float32, (N,))
with tf.name_scope("wagi"):
  W = tf.Variable(tf.random_normal((2, 1)))
  b = tf.Variable(tf.random_normal((1,)))
with tf.name_scope("predykcja"):
  y_logit = tf.squeeze(tf.matmul(x, W) + b)
  # sigmoida podaje klasę prawdopodobieństwa dla 1
  y_one_prob = tf.sigmoid(y_logit)
  # Zaokrąglenie P(y=1) da prawidłową prognozę.
  y_pred = tf.round(y_one_prob)

with tf.name_scope("strata"):
  # Obliczanie entropii krzyżowej dla każdego punktu danych
  entropy = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_logit, labels=y)
  # Sumowanie wszystkich wkładów
  l = tf.reduce_sum(entropy)
with tf.name_scope("optymalizatory"):
  train_op = tf.train.AdamOptimizer(.01).minimize(l)

  train_writer = tf.summary.FileWriter('/tmp/logistic-train', tf.get_default_graph())
```

Kod treningowy dla tego modelu w listingu 3.15 jest identyczny jak dla modelu regresji liniowej.

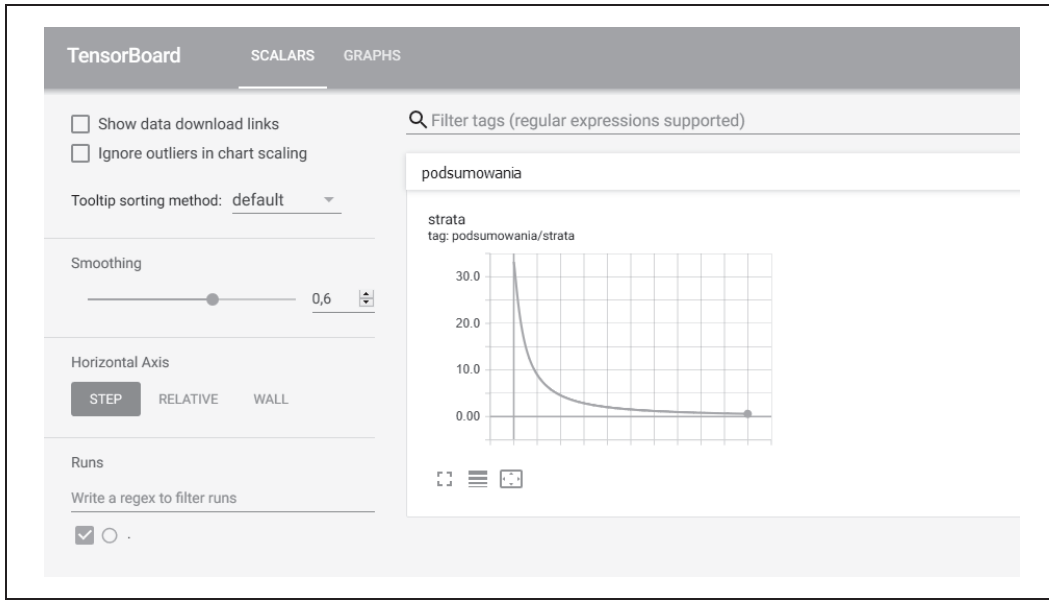
Listing 3.15. Trenowanie modelu regresji logistycznej

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
```

```
# Trenowanie modelu
for i in range(n_steps):
  feed_dict = {x: x_np, y: y_np}
  _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
  print("strata: %f" % loss)
  train_writer.add_summary(summary, i)
```

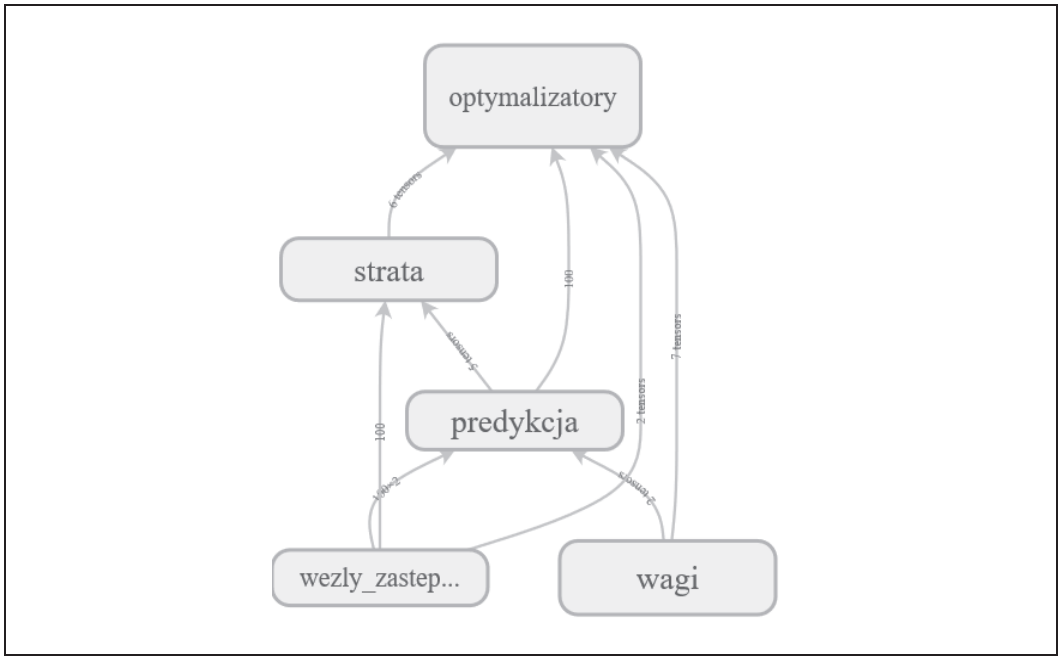
Wizualizacja modeli regresji logistycznej za pomocą TensorBoard

Tak jak poprzednio, do wizualizacji modelu można użyć TensorBoard. Rozpocznij od wizualizacji funkcji straty, jak pokazano na rysunku 3.15. Zwróć uwagę, że tak jak poprzednio, funkcja straty jest zgodna z czystym wzorcem. Występuje tu gwałtowny spadek straty, po którym następuje stopniowe wygładzanie.



Rysunek 3.15. Wizualizacja funkcji straty regresji logistycznej

Możesz również wyświetlić wykres TensorFlow w TensorBoard. Ponieważ struktura zakresów była podobna do struktury zastosowanej w regresji liniowej, uproszczony wykres przedstawiony na rysunku 3.16 nie będzie zbytnio odbiegał wyglądem.

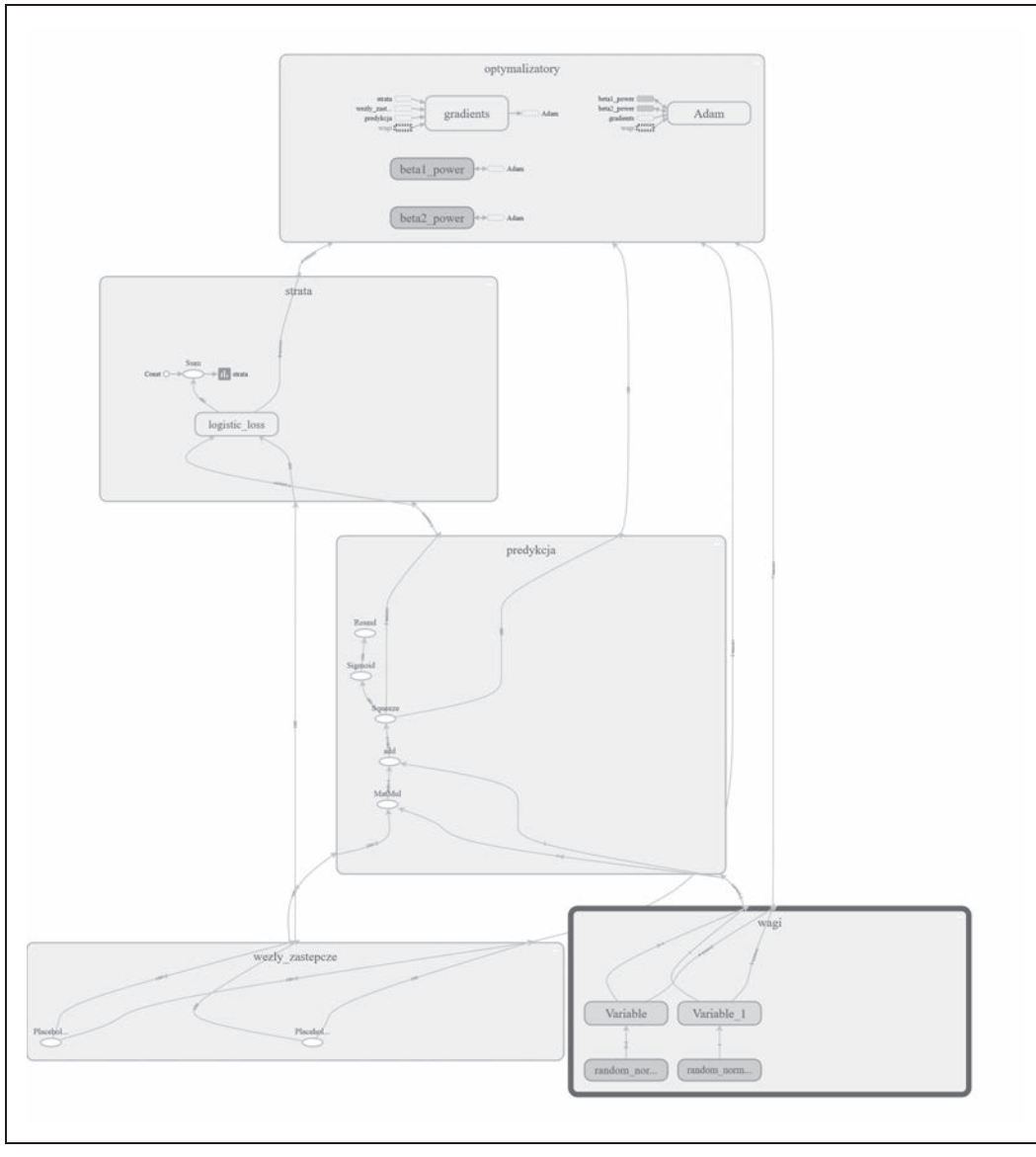


Rysunek 3.16. Wizualizacja wykresu obliczeniowego dla regresji logistycznej

Jeśli jednak rozwiniesz węzły na tym zgrupowanym wykresie, jak na rysunku 3.17, okaże się, że bazowy wykres obliczeniowy jest inny. W szczególności funkcja straty jest zupełnie inna od tej, która jest używana przy regresji liniowej (i powinna taka być).

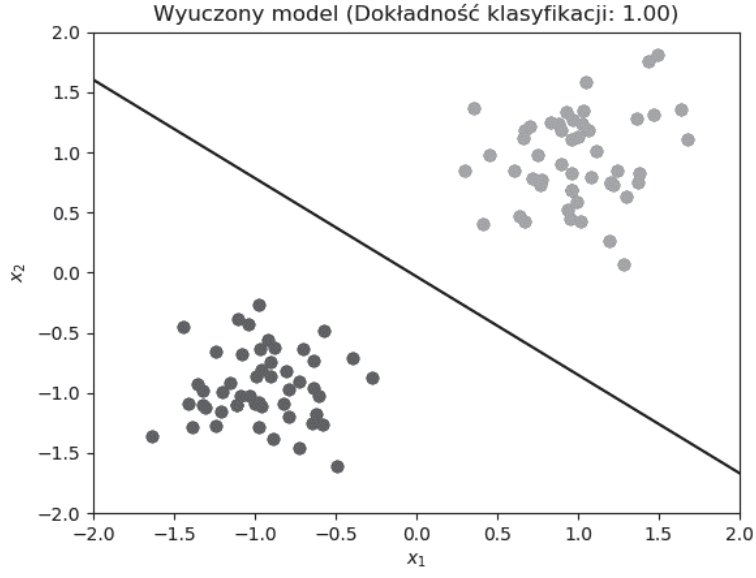
Wskaźniki oceny modeli klasyfikacji

Teraz, gdy przeszkoliłeś model klasyfikacji regresji logistycznej, musisz zapoznać się z wskaźnikami przydatnymi do oceny modeli klasyfikacji. Chociaż równania regresji logistycznej są bardziej skomplikowane niż równania regresji liniowej, podstawowe wskaźniki oceny są prostsze. Dokładność klasyfikacji sprawdza po prostu wycinek punktów danych, które są poprawnie sklasyfikowane przez nauczony model. W rzeczywistości przy nieco większym wysiłku istnieje możliwość rezygnacji z **linii oddzielającej**, której nauczył się model regresji logistycznej. Ta linia wyświetla wyuczoną przez model granicę oddzielającą pozytywne i negatywne przykłady. (Zadanie wyprowadzenia tej linii z równań regresji logistycznej pozostawiamy jako ćwiczenie dla zainteresowanych czytelników. Rozwiązanie zawarte jest w kodzie dla tego podrozdziału).



Rysunek 3.17. Rozszerzony wykres obliczeniowy dla regresji logistycznej

Na rysunku 3.18 przedstawiamy wyuczone klasy i linię oddzielającą. Zauważ, że linia starannie oddziela pozytywne i negatywne przykłady i ma doskonałą dokładność (1,0). Wynik ten nasuwa interesującą kwestię. Regresja jest często trudniejszym problemem do rozwiązania niż klasyfikacja. Istnieje wiele możliwych linii, które mogłyby starannie oddzielić punkty danych na rysunku 3.18, ale tylko jedna, która idealnie pasowałaby do danych regresji liniowej.



Rysunek 3.18. Przeglądanie wyuczonych klas i linii oddzielającej dla regresji logistycznej

Podsumowanie

W tym rozdziale pokazaliśmy, jak zbudować i przeszkolić kilka prostych systemów uczenia w TensorFlow. Zaczęliśmy od przeglądu podstawowych pojęć matematycznych, w tym funkcji straty i gradientu prostego. Następnie wprowadziliśmy nowe koncepcje TensorFlow, takie jak węzły zastępcze, zakresy i TensorBoard. Zakończyliśmy ten rozdział studiami przypadków, w których szkoliliśmy systemy regresji liniowej i logistycznej na ćwiczebnych zbiorach danych. W tym rozdziale omówiliśmy sporo materiału i nic nie szkodzi, jeśli jeszcze wszystkiego nie przyswoiłeś. Wprowadzony tu materiał będzie wykorzystywany przez resztę tej książki.

W rozdziale 4. wprowadzimy Cię do Twojego pierwszego modelu uczenia głębokiego i do w pełni połączonych sieci oraz pokażemy Ci, jak definiować i szkolić w pełni połączone sieci w TensorFlow. W kolejnych rozdziałach zajmiemy się bardziej skomplikowanymi sieciami głębokimi, ale wszystkie te architektury będą wykorzystywać te same podstawowe zasady uczenia, które zostały wprowadzone w tym rozdziale.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

O'REILLY®

TensorFlow: trenuj sieć profesjonalnie!

Uczenie maszynowe jest coraz powszechniejsze. Niemal każdego dnia stykamy się z tego rodzaju oprogramowaniem, a możliwości tworzonych systemów stale rosną. Zdobycie praktycznych umiejętności w zakresie budowy i treningu sieci neuronowych staje się dla profesjonalnych programistów koniecznością. Spośród wielu narzędzi służących do tworzenia systemów uczenia maszynowego warto zwrócić uwagę na TensorFlow — nową bibliotekę udostępnioną przez Google, przeznaczoną do projektowania i wdrażania zaawansowanych architektur głębokiego uczenia. Bez wątpienia jest to narzędzie, które pozwala na wykonywanie zadań znacznie wykraczających poza standardowy zakres uczenia maszynowego.

Ta książka jest przeznaczona dla praktyków, przede wszystkim programistów, architektów i naukowców, którzy chcą się nauczyć projektowania systemów uczących. Podstawowe pojęcia dotyczące uczenia maszynowego wyjaśniono tu poprzez praktyczne przykłady. Przedstawiono możliwości TensorFlow jako systemu do przeprowadzania obliczeń na tensorach. Omówiono zastosowania tej biblioteki w wielu bardzo różnych dziedzinach: do budowy systemów służących do rozpoznawania obrazów, rozumienia tekstu napisanego ręcznie przez człowieka czy przewidywania właściwości potencjalnych leków. Dzięki tej książce można bez trudu zrozumieć matematyczne podstawy systemów uczenia maszynowego, a następnie wykorzystać je podczas tworzenia profesjonalnych sieci neuronowych.

W tej książce między innymi:

- podstawy uczenia maszynowego i rozpoczęcie pracy z TensorFlow
- budowa prototypów i modeli z optymalizacją hiperparametrów
- przetwarzanie obrazów w splotowych sieciach neuronowych
- obsługa zbiorów danych języka naturalnego
- trenowanie sieci za pomocą procesorów graficznych i procesorów tensorowych

Bharath Ramsundar — jest twórcą DeepChem, pakietu open source opartego na TensorFlow, służącego do opracowywania leków. Przygotowuje doktorat z informatyki na Uniwersytecie Stanforda.

Reza Bosagh Zadeh — jest wykładowcą na Uniwersytecie Stanforda. Zawodowo zajmuje się uczeniem maszynowym, obliczeniami rozproszonymi i dyskretną matematyką stosowaną. Opracował algorytmy uczenia maszynowego stojące za systemem proponowania kont do śledzenia na Twitterze.

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

ISBN 978-83-283-5705-1

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł