O'REILLY®

Deep Learning Receptury

Helion

Douwe Osinga

Tytuł oryginału: Deep Learning Cookbook: Practical Recipes to Get Started Quickly

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-5231-5

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)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Narzędzia i techniki

W tym rozdziale przyjrzymy się narzędziom i technikom powszechnie używanym do uczenia głębokiego. Rozdział ten świetnie nadaje się do tego, by przeczytać go raz i zdobyć ogólną orientację w temacie, a następnie wracać, kiedy będzie to potrzebne.

Zaczniemy od przeglądu różnych typów sieci neuronowych, które będą używane w tej książce. Większość z prezentowanych w niej receptur koncentruje się na tym, by wykonać określone zadanie, i jedynie pobieżnie opisuje architektury wykorzystywanych sieci.

Następnie skoncentrujemy się na tym, jak zdobywać dane. Technologiczni giganci, tacy jak Facebook i Google, mają dostęp do ogromnych ilości danych, które mogą stosować do swoich celów badawczych, jednak w internecie możemy znaleźć także dostatecznie dużo danych, by móc zrobić coś ciekawego. Receptury zamieszczone w tej książce korzystają z danych pochodzących z szerokiego zakresu źródeł.

Kolejna część rozdziału będzie poświęcona wstępnemu przetwarzaniu danych. To bardzo ważne zagadnienie, które niejednokrotnie jest pomijane. A nawet jeśli nasza sieć neuronowa została odpowiednio przygotowana i jeśli dysponujemy świetnymi danymi, to i tak musimy się upewnić, że dane te będą w najlepszy możliwy sposób dostarczone do sieci.

1.1. Typy sieci neuronowych

W tym rozdziale oraz w dalszej części książki będziemy mówić o *sieciach* i *modelach*. Sieć to skrótowe określenie sieci neuronowej, odnoszące się do grupy połączonych ze sobą *warstw*. Dane są przekazywane do sieci z jednej jej strony, a z drugiej strony sieci są zwracane wyniki. Każda z warstw sieci implementuje pewną operację matematyczną, która jest wykonywana na danych przechodzących przez warstwę, i dysponuje pewnymi zmiennymi, które można modyfikować, by dokładnie określać działanie tej warstwy. W tym kontekście *danymi* są *tensory*, czyli wielowymiarowe wektory (które zazwyczaj mają dwa lub trzy wymiary).

Pełny opis różnych rodzajów warstw i realizowanych przez nie operacji matematycznych wykracza poza zakres tej książki. Najprostszym typem warstwy jest warstwa *w pełni połączona*; pobiera ona dane w formie macierzy, mnoży tę macierz przez inną macierz, nazywaną *wagami* (ang. *weights*), i dodaje do wyniku kolejną macierz, nazywaną obciążeniem (ang. *bias*). Za każdą warstwą znajduje się

funkcja aktywacji — funkcja matematyczna, która odwzorowuje wyniki jednej warstwy na wejścia następnej. Na przykład prosta funkcja aktywacji nazywana ReLU przekazuje wszystkie wartości dodatnie bez zmian, natomiast wartości ujemne zmienia na zera.

Z technicznego punktu wiedzenia termin *sieć* odnosi się do architektury — sposobu, w jaki poszczególne warstwy są ze sobą połączone. Z kolei *model* to sieć oraz wszystkie zmienne określające sposób jej działania. Uczenie modelu modyfikuje te zmienne, tak by zwracane przez sieć prognozy lepiej odpowiadały oczekiwanym wynikom. Jednak w praktyce oba te terminy często są używane zamiennie.

Terminy „uczenie głębokie" oraz „sieci neuronowe" w rzeczywistości obejmują szeroki zakres różnych modeli. Większość z tych sieci będzie dysponować podobnymi elementami (na przykład niemal wszystkie sieci służące do klasyfikacji będą korzystać z konkretnej wersji *funkcji straty*). Choć przestrzeń modeli jest bardzo zróżnicowana, to jednak większość z nich będzie należeć do jednej z kilku ogólnych kategorii. Niektóre modele będą korzystać z elementów należących do różnych kategorii; na przykład wiele sieci służących do klasyfikowania obrazów używa w pełni połączonej „czołowej" sekcji do wykonywania ostatecznej klasyfikacji.

Sieci w pełni połączone

Sieci w pełni połączone były pierwszym typem sieci neuronowych, nad którymi prowadzono prace. Aż do późnych lat 80. ubiegłego wieku były w centrum zainteresowania. W sieciach tego typu każda jednostka wyjściowa jest wyliczana jako suma ważona wszystkich wejść. Określenie „w pełni połączona" oddaje sposób działania sieci: każde wyjście jest połączone ze wszystkimi wejściami. Można to zapisać przy użyciu następującego wzoru:

$$y_i = \sum_j W_{ij} x_j$$

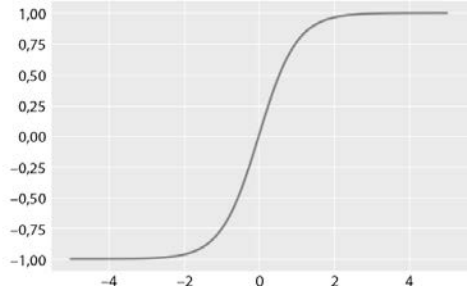
Większość publikacji przedstawia sieci w pełni połączone za pomocą zapisu macierzowego. W tym przypadku mnożymy wektor wejść przez macierz wag W i uzyskujemy w ten sposób wektory wyjść:

$$y = Wx$$

Ponieważ mnożenie macierzy jest operacją liniową, sieć zawierająca jedynie mnożenie macierzy byłaby w stanie uczyć się tylko odwzorowań liniowych. Aby sieci neuronowe miały większe możliwości wyrazu, mnożenie macierzy jest uzupełniane przez nieliniową funkcję aktywacji. Może to być dowolna funkcja różniczkowalna, jednak powszechnie stosuje się tylko kilka takich funkcji. Do niedawna najczęściej stosowaną funkcją aktywacji był tangens hiperboliczny, *tanh*, który zresztą aż do dziś można znaleźć w niektórych modelach (rysunek 1.1).

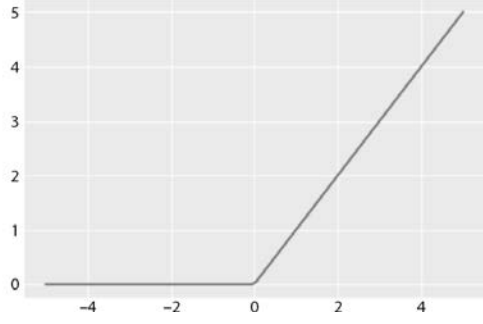
Problem związany ze stosowaniem tej funkcji polega na tym, że staje się ona bardzo „płaska" dla wartości wejściowych znacząco odbiegających od zera. W efekcie gradient tej funkcji jest bardzo mały, co oznacza, że uzyskanie zmiany zachowania sieci może zająć bardzo dużo czasu. Ostatnio większą popularnością cieszą się inne funkcje aktywacji. Jedną z najpopularniejszych jest funkcja aktywacji określana jako *prostowana jednostka liniowa* (ang. *Rectified Linear Unit* albo w skrócie *ReLU*, patrz rysunek 1.2).

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Rysunek 1.1. Wykres funkcji aktywacji tanh

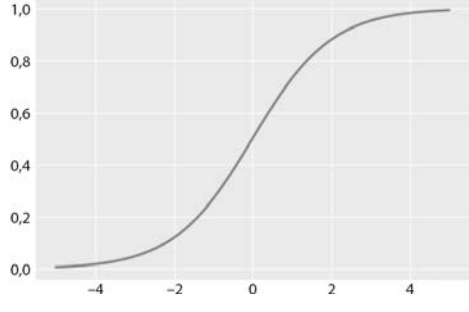
$$relu(x) = \begin{cases} 0 & \text{jeżeli } x < 0 \\ x & \text{jeżeli } x \geq 0 \end{cases}$$



Rysunek 1.2. Wykres funkcji aktywacji ReLU

I w końcu, wiele sieci neuronowych w swojej ostatniej warstwie używa funkcji *sigmoidalnej* (ang. *sigmoid*). Funkcja ta charakteryzuje się tym, że zawsze zwraca wartości z zakresu od 0 do 1. Ta cecha sprawia, że wyniki sieci można traktować jako prawdopodobieństwa (rysunek 1.3):

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

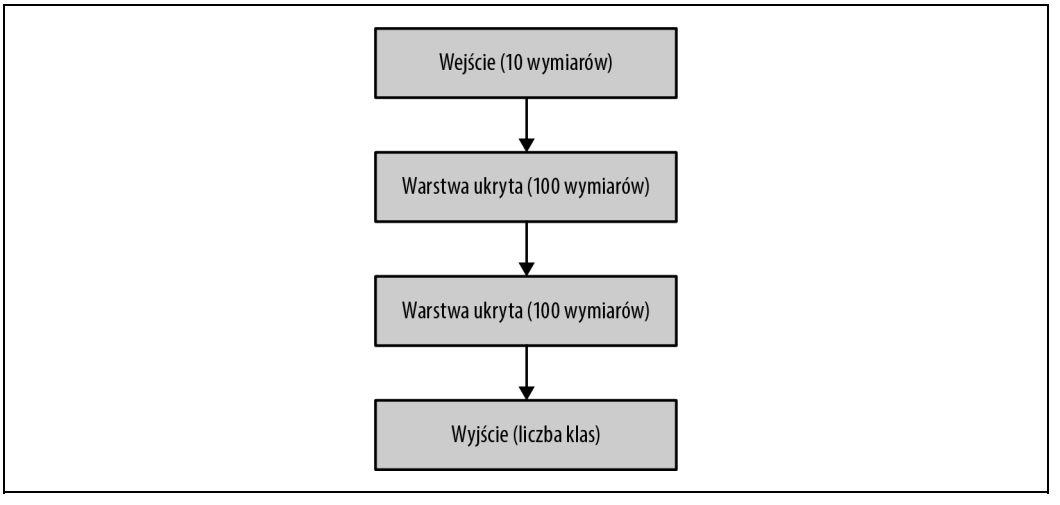


Rysunek 1.3. Wykres sigmoidalnej funkcji aktywacji

Operacja mnożenia macierzy oraz wykonywana za nią funkcja aktywacji są nazywane *warstwą* sieci neuronowej. W niektórych przypadkach kompletna sieć neuronowa może się składać nawet ze 100 warstw, jednak w sieciach w pełni połączonych ich liczba zazwyczaj nie przekracza kilku.

Jeśli rozwiązujemy zadanie klasyfikacyjne typu „jakiego gatunku kot jest widoczny na zdjęciu?”, to ostatnia warstwa sieci jest nazywana *warstwą klasyfikacyjną*. Liczba wyjść tej warstwy zawsze będzie odpowiadać liczbie dostępnych klas, spośród których należy dokonać wyboru.

Warstwy umieszczone wewnątrz sieci są nazywane *warstwami ukrytymi*, a poszczególne wyjścia używane w warstwach ukrytych — *jednostkami ukrytymi*. Określenie „ukryty” pochodzi stąd, że jednostki te nie są widoczne bezpośrednio na zewnątrz sieci, tak jak jej wejścia i wyjścia. Liczba wyjść w tych warstwach ukrytych zależy od modelu (patrz rysunek 1.4):



Rysunek 1.4. Przykładowa struktura sieci neuronowej z warstwami ukrytymi

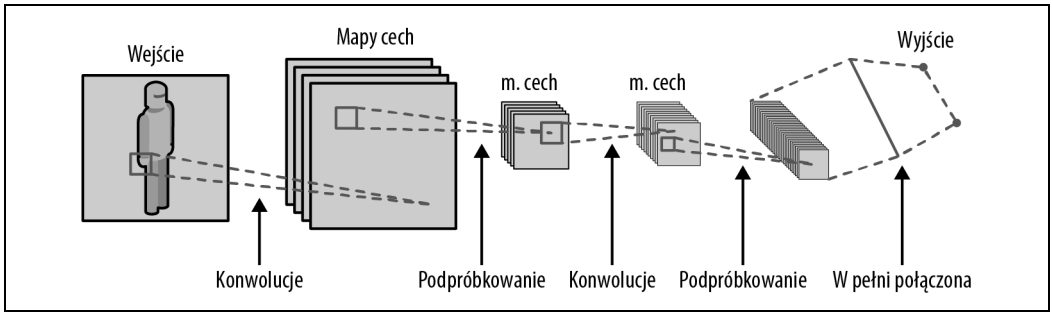
Choć istnieje kilka praktycznych zasad dotyczących wyboru liczby oraz wielkości warstw ukrytych, to jednak — ogólnie rzecz biorąc — oprócz metody prób i błędów nie istnieje żadna ogólna strategia dotycząca doboru struktury sieci.

Sieci konwolucyjne

W ramach początkowych badań nad sieciami neuronowymi używano sieci w pełni połączonych, próbując rozwiązywać przy ich użyciu szerokie spektrum problemów. Jednak kiedy danymi wejściowymi są obrazy, to sieci w pełni połączone nie są optymalnym rozwiązaniem. Obrazy są bardzo duże: obraz o wymiarach 256×256 pikseli (taka rozdzielczość jest często stosowana w zadaniach klasyfikacji) wymaga użycia 256×256×3 wejść (każdy piksel ma bowiem 3 kolory). Gdyby model miał jedną warstwę ukrytą składającą się z 1000 jednostek, to warstwa ta miałaby niemal 200 milionów parametrów (uczonych wartości)! Ponieważ uzyskanie dobrych wyników klasyfikacji obrazów wymaga zastosowania kilku warstw, to gdybyśmy chcieli implementować je przy użyciu warstw w pełni połączonych, model miałby miliardy parametrów.

W przypadku zastosowania takiej liczby parametrów niemal niemożliwe byłoby uniknięcie *nadmiernego dopasowania* modelu (problem nadmiernego dopasowania zostanie dokładniej opisany w następnym rozdziale; ogólnie rzecz biorąc, odpowiada on jednak sytuacji, w której sieć neuronowa tylko zapamiętuje wyniki, a nie jest w stanie ich uogólniać). *Konwolucyjne sieci neuronowe* (ang.

Convolutional Neural Networks, określane skrótowo jako *CNN*) pozwalają na tworzenie niemal nadnaturalnie dobrych klasyfikatorów przy wykorzystaniu znacznie mniejszej liczby parametrów. Sieci konwolucyjne robią to, naśladując sposób, w jaki widzą ludzie i zwierzęta (patrz rysunek 1.5):



Rysunek 1.5. Poglądowa struktura konwolucyjnej sieci neuronowej

Podstawowymi operacjami sieci konwolucyjnych są *konwolucje*. Zamiast wykonywać określoną funkcję na całym obrazie wejściowym, konwolucja wykonuje ją kolejno dla niewielkich fragmentów obrazu. W każdym miejscu konwolucja wykonuje *jądro* (zazwyczaj jest to operacja mnożenia macierzy oraz funkcja aktywacji, podobnie jak w przypadku sieci w pełni połączonych). Pojedyncze jądra są także często nazywane *filtrami*. W efekcie zastosowania jądra dla całego obszaru obrazu powstaje nowy, najprawdopodobniej mniejszy obraz. Na przykład często stosowane są filtry o wymiarach 3×3. Gdybyśmy zastosowali 32 takie filtry dla obrazu wejściowego, to sieć potrzebowałaby 3·3·3 (kolorów wejściowych)·32, czyli 864 parametry — to bardzo duża oszczędność w stosunku do sieci w pełni połączonej!

Podpróbkowanie

Ta operacja ogranicza liczbę parametrów, lecz teraz stajemy przed innym problemem. Każda warstwa sieci może w danej chwili „analizować” jedynie fragment obrazka o wymiarach 3×3; a skoro tak, to w jaki sposób można rozpoznawać obiekty, które potencjalnie mogą zajmować cały obszar obrazu?

By rozwiązać ten problem, typowe konwolucyjne sieci neuronowe korzystają z *podpróbkowania* — techniki służącej do zmniejszania wymiarów obrazu przekazywanego wewnątrz sieci. Dwoma mechanizmami najczęściej używanymi do podpróbkowania są:

Konwolucje kroczące (ang. *strided convolutions*)
: W tym przypadku po prostu pomijamy jeden lub kilka pikseli podczas przesuwania filtra konwolucyjnego po powierzchni obrazu. W efekcie uzyskujemy mniejszy obraz. Na przykład jeśli obraz wejściowy miał wymiary 256×256 pikseli, a pomijamy co drugi piksel, to obraz wyjściowy będzie miał wymiary 128×128 pikseli (dla ułatwienia pomijamy przy tym obramowanie umieszczane przy krawędziach obrazu). Tego rodzaju kroczące ograniczanie rozdzielczości obrazu jest powszechnie stosowane w sieciach generujących (patrz punkt „Sieci typu GAN i autoenkodery”).

Odpytywanie (ang. *pooling*)
: Zamiast stosowania pomijania pikseli przez operację konwolucji wiele sieci neuronowych ogranicza liczbę wejść poprzez stosowanie *warstw odpytujących* (ang. *pooling layers*). Warstwa

odpytująca jest w zasadzie inną formą warstwy konwolucyjnej — zamiast mnożyć wejścia przez macierz, wykonuje operacje odpytywania. Najczęściej operacja ta sprowadza się do użycia operatora *max* lub operatora *average*. Pierwszy z tych operatorów wyznacza wartość maksymalną dla każdego z *kanałów* (kolorów) w obszarze próbkowania. Z kolei drugi operator wyznacza średnią wartości poszczególnych kanałów w obszarze próbkowania. (Można go sobie wyobrażać jako pewnego rodzaju zamazywanie wejścia).

Jedną z możliwości wyobrażania sobie podpróbkowania jest uznanie go za sposób zwiększania poziomu abstrakcji tego, co robi sieć neuronowa. Na najniższym poziomie konwolucje wykrywają małe, lokalne cechy. Istnieje wiele takich cech, które nie są zlokalizowane zbyt głęboko. Wraz z każdym etapem odpytywania zwiększa się poziom abstrakcji; liczba cech jest zmniejszana, lecz zwiększa się głębokość każdej z nich. Ten proces jest kontynuowany tak długo, aż uzyskamy bardzo niewiele cech o wysokim poziomie abstrakcji, których możemy używać do wykonywania prognoz.

Prognozy

Po połączeniu większej liczby warstw konwolucyjnych z odpytującymi konwolucyjne sieci neuronowe używają jednej lub dwóch warstw w pełni połączonych, aby dokonać końcowych prognoz i zwrócić wyniki.

Sieci rekurencyjne

Rekurencyjne sieci neuronowe (ang. *Recurrent Neural Networks*, w skrócie *RNN*) są pod względem pojęciowym podobne do sieci konwolucyjnych, choć różnią się od nich pod względem struktury. Sieci rekurencyjne są często stosowane w sytuacjach, gdy dane wejściowe mają charakter sekwencyjny. Dane tego typu często występują podczas operacji na tekstach lub przetwarzania mowy. Zamiast przetwarzać jeden, kompletny przykład (jak w przypadku przetwarzania obrazu przy użyciu sieci konwolucyjnej), problemy sekwencyjne można rozwiązywać fragment po fragmencie. Rozważmy na przykład tworzenie sieci neuronowej, która będzie pisać dla nas sztuki wzorowane na dziełach Szekspira. Wejściem dla takiej sieci neuronowej byłyby oczywiście istniejące sztuki Szekspira:

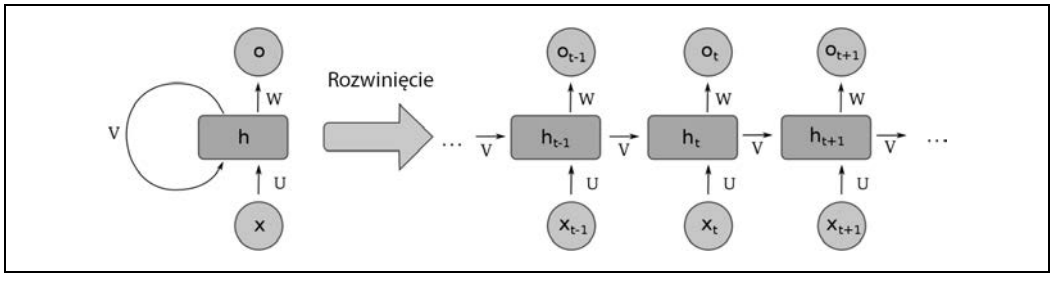
```
Lear. Attend the lords of France and Burgundy, Gloucester.
Glou. I shall, my liege.
```

W tym przykładzie naszym celem jest stworzenie sieci, która będzie prognozować następne słowo w sztuce. W tym celu sieć musi „pamiętać” tekst, który widziała wcześniej. Właśnie taki mechanizm zapewniają rekurencyjne sieci neuronowe. Pozwalają one także budować modele, które w naturalny sposób mogą operować na danych wejściowych o zmiennej długości (takich jak zdania lub fragmenty wypowiedzi). Najprostszą postać rekurencyjnej sieci neuronowej przedstawia następujący schemat (patrz rysunek 1.6).

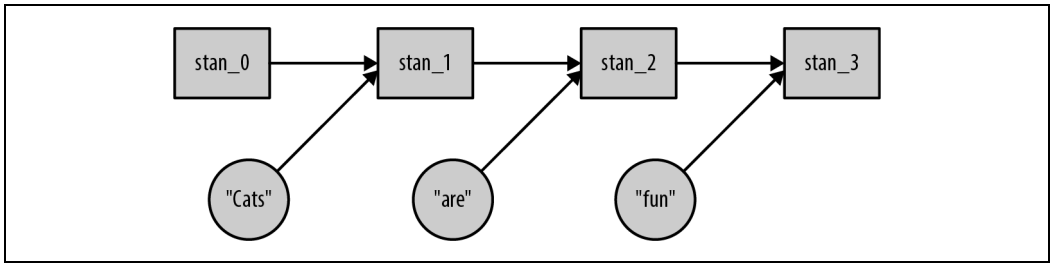
Po względem koncepcyjnym można wyobrażać sobie sieć typu RNN jako bardzo głęboką sieć w pełni połączoną, która została „rozwinięta”. W tym koncepcyjnym modelu każda warstwa sieci ma dwa wejścia, a nie jak wcześniej, tylko jedno (patrz rysunek 1.7).

Przypomnijmy sobie, że w sieci w pełni połączonej używana była operacja mnożenia macierzy, którą można było zapisać w następujący sposób:

$$y = Wx$$



Rysunek 1.6. Schematyczna postać rozwinięcia rekurencyjnej sieci neuronowej



Rysunek 1.7. Inna koncepcyjna postać rekurencyjnej sieci neuronowej

Najprostszym sposobem dodania do tej operacji drugiego wejścia jest po prostu dołączenie go do ukrytego stanu:

$$ukryty_i = W\{ukryty_{i-1} \mid x\}$$

gdzie symbol „ | ” oznacza operację dołączenia czy też konkatenacji. Podobnie jak w przypadku sieci w pełni połączonych, także w sieciach rekurencyjnych, aby uzyskać nowy stan po operacji mnożenia macierzy można zastosować funkcję aktywacji:

$$ukryty_i = f(W\{ukryty_{i-1} \mid x\})$$

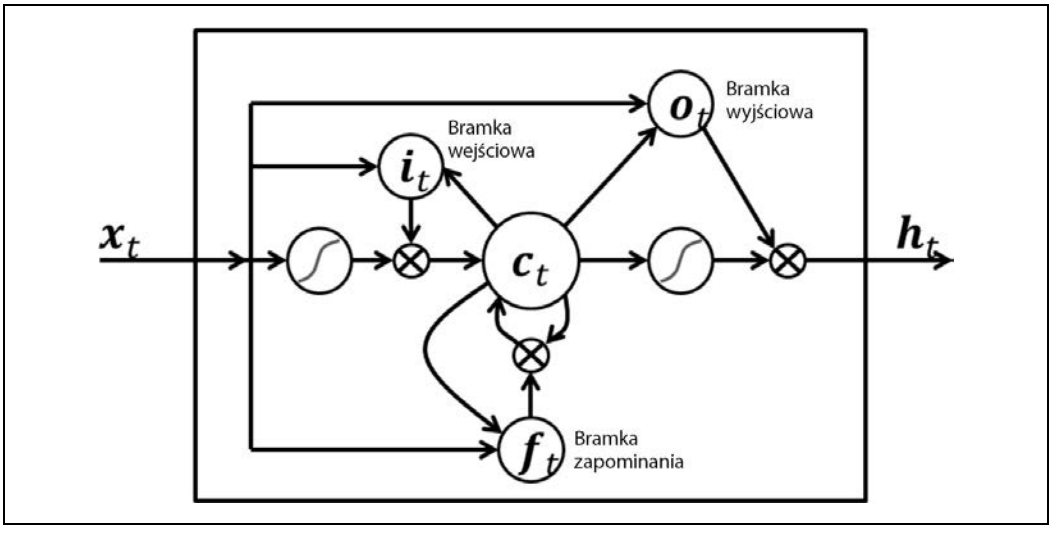
Dzięki takiej interpretacji sieci rekurencyjnej bez trudu można zrozumieć, jak należy ją uczyć: sieć RNN należy potraktować jak rozwiniętą sieć w pełni połączoną i uczyć ją w normalny sposób. W literaturze jest to określane jako *propagacja wsteczna przez czas* (ang. *backpropagation through time*, w skrócie *BPTT*). Jeśli dysponujemy bardzo długimi danymi wejściowymi, to powszechnie stosowanym rozwiązaniem jest dzielenie ich na mniejsze fragmenty i uczenie sieci każdego z nich niezależnie. Choć takie rozwiązanie nie sprawdzi się dla każdego problemu, jest jednak raczej bezpieczną i często stosowaną techniką.

Znikające gradienty i sieci LSTM

W przypadku długich danych wejściowych nasze proste sieci RNN niestety spisują się gorzej, niż byśmy sobie tego życzyli. Dzieje się tak, gdyż struktura sprawia, że stają się one podatne na problem „znikających gradientów”. Przyczyną tego problemu jest bardzo duża głębokość nierozwiniętych rekurencyjnych sieci neuronowych. Za każdym razem, gdy wartość jest przekazywana do funkcji aktywacji, istnieje spore prawdopodobieństwo, że przekazywany dalej gradient będzie bardzo mały (na przykład funkcja aktywacji ReLU sprawi, że dla każdej wartości wejściowej mniejszej od zera

gradient będzie wynosił zero). Kiedy tak się stanie dla jednej jednostki, sieć za tą jednostką już nie będzie uczona. A to z kolei sprawi, że do dalszej części sieci będzie przekazywany jeszcze słabszy sygnał uczący. W efekcie obserwowane uczenie sieci będzie bardzo wolne lub sieć nawet w ogóle nie będzie się uczyć.

Aby poradzić sobie z tym problemem, naukowcy opracowali alternatywny mechanizm tworzenia rekurencyjnych sieci neuronowych. Podstawowy model rozwijania sieci w czasie został zachowany, jednak zamiast wykonywania prostego mnożenia macierzy i następującego po nim wywołania funkcji aktywacji zastosowano nieco bardziej złożony mechanizm przekazywania wartości do kolejnych warstw sieci (Wikipedia: *https://en.wikipedia.org/wiki/Long_short-term_memory#/media/File:Peephole_Long_Short-Term_Memory.svg*; patrz rysunek 1.8):



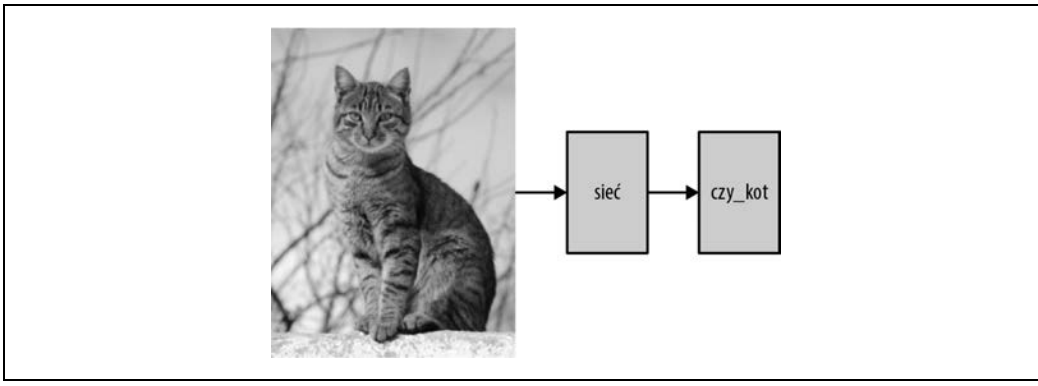
Rysunek 1.8. Schematyczna postać węzła sieci neuronowej typu LSTM

W sieciach neuronowych określanych jako *długa pamięć krótkoterminowa* (ang. *long short-term memory*, w skrócie *LSTM*) pojedyncza operacja mnożenia macierzy została zastąpiona czterema takimi operacjami, a dodatkowo wprowadzono ideę *bramek*, które są mnożone przez wektor. Podstawową cechą, która sprawia, że sieci LSTM uczą się bardziej efektywnie niż zwyczajne rekurencyjne sieci neuronowe, jest to, że zawsze istnieje ścieżka prowadząca od ostatecznej prognozy do dowolnej warstwy zachowującej gradienty. Szczegółowy opis mechanizmu zapewniającego tę możliwość wykracza poza zakres tego rozdziału, niemniej jednak w internecie można znaleźć kilka doskonałych poradników (*http://colah.github.io/posts/2015-08-Understanding-LSTMs/*) poświęconych temu zagadnieniu.

Sieci typu GAN i autoenkodery

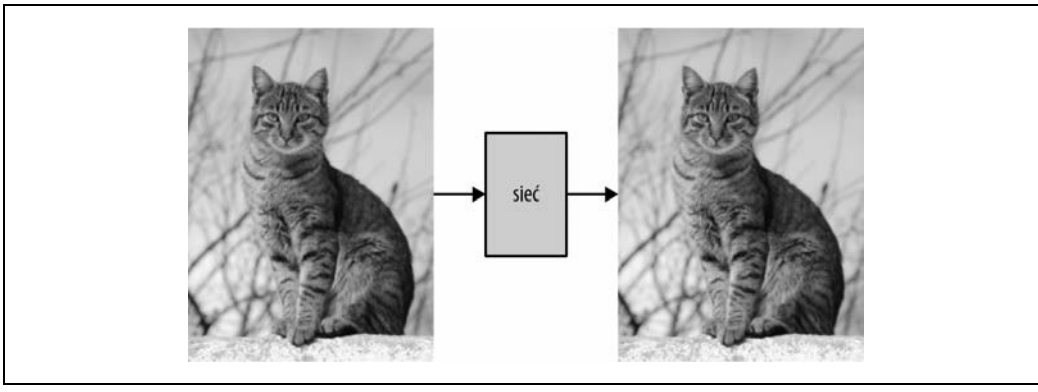
W odróżnieniu od sieci przedstawionych w poprzednich punktach rozdziału sieci neuronowe typu GAN (ang. *Generative Adversarial Networks*) oraz autoenkodery nie wprowadzają żadnych nowych komponentów strukturalnych. Zamiast tego korzystają ze struktury, która w możliwie jak najlepszym stopniu odpowiada konkretnemu rozwiązywanemu problemowi; na przykład w przypadku

operowania na obrazach zarówno sieci typu GAN, jak i autoenkodery będą używać konwolucji. Tym, co wyróżnia te rodzaje sieci neuronowych, jest sposób ich uczenia. Większość typowych sieci neuronowych jest uczona pod kątem prognozowania wyników (czy to jest kot?) na podstawie danych wejściowych (zdjęcia; patrz rysunek 1.9):



Rysunek 1.9. Sieć neuronowa używana do prognozowania

Autoenkodery są uczone po to, by zwracać obraz przekazany na wejściu (patrz rysunek 1.10):



Rysunek 1.10. Schematyczne zastosowanie sieci typu autoenkoder

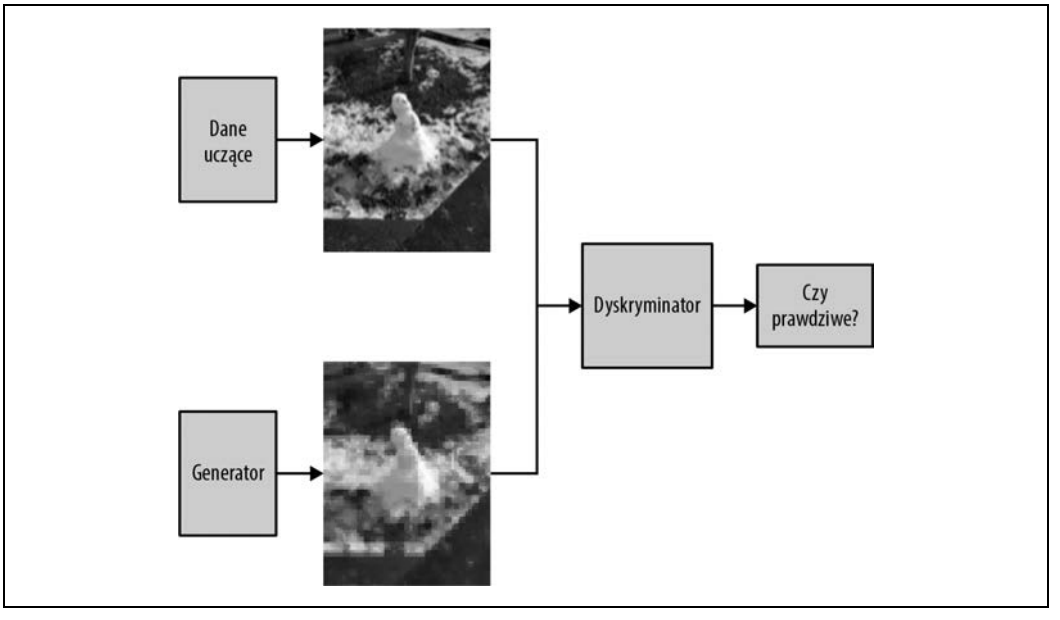
A w jakim celu moglibyśmy chcieć wykonywać taką operację? Jeśli warstwy ukryte umieszczone wewnątrz sieci neuronowej zawierają reprezentację obrazu wejściowego, która ma (znacząco) mniej informacji niż początkowy obraz, a jednocześnie na podstawie tej reprezentacji ów początkowy obraz można odtworzyć, to sprowadza się to do pewnego rodzaju kompresji: można wziąć taki obraz i reprezentować go jedynie przy użyciu wartości z warstwy ukrytej. Można to sobie wyobrazić tak, że używamy sieci neuronowej do rzutowania początkowego obrazu na jakąś abstrakcyjną przestrzeń. Każdy punkt tej przestrzeni można następnie skonwertować z powrotem na obraz.

Autoenkodery z powodzeniem stosowano na małych obrazach, jednak mechanizmu ich uczenia nie dawało się przeskalować i używać do rozwiązywania większych problemów. W praktyce okazuje się, że przestrzeń wykorzystywana do odtwarzania obrazów nie jest dostatecznie „gęsta” i wiele jej punktów w rzeczywistości nie reprezentuje spójnych obrazów.

Przykład zastosowania tego typu sieci neuronowych zostanie przedstawiony w rozdziale 13.

Sieci typu GAN są jednymi z nowszych modeli sieci neuronowych, które potrafią generować realistyczne obrazy. Ich działanie bazuje na podziale problemu na dwie części: sieć generatora oraz sieć dyskryminatora. Sieć dyskryminatora pobiera niewielką, losową próbkę i generuje obraz (lub tekst). Z kolei sieć dyskryminatora stara się określić, czy wejściowy obraz jest „prawdziwy", czy też został stworzony przez sieć generatora.

Podczas uczenia takiej sieci uczone są jednocześnie obie jej części (patrz rysunek 1.11):



Rysunek 1.11. Schematyczna postać sieci typu GAN

Próbkujemy obrazy z sieci generatora i przekazujemy je poprzez sieć dyskryminatora. Sieć generatora jest nagradzana za tworzenie obrazów, które będą w stanie oszukać dyskryminator. Sieć dyskryminatora musi także prawidłowo rozpoznawać prawdziwe obrazy (nie może za każdym razem stwierdzać, że obraz jest podrobiony). Dzięki temu, że obie sieci wzajemnie ze sobą konkurują, rozwiązanie to pozwala uzyskać sieć generatora zdolną do tworzenia naturalnych obrazów o bardzo wysokiej jakości. W rozdziale 14. zostały przedstawione receptury pokazujące zastosowanie sieci typu GAN do generowania ikon.

Wnioski

Sieci neuronowe mogą mieć bardzo wiele różnych architektur, a ich postać w głównej mierze będzie zależeć od przeznaczenia danej sieci. Projektowanie nowych typów sieci jest zadaniem typowo naukowym, a nawet ponowna implementacja typu sieci opisanego w publikacji jest bardzo trudnym zadaniem. W praktyce najprostszym rozwiązaniem jest znalezienie przykładów sieci, które robią coś zbliżonego do tego, o co nam chodzi, i sukcesywne zmienianie ich tak, by zaczęły robić dokładnie to, o co nam chodzi.

1.2. Pozyskiwanie danych

Jednym z głównych powodów wzrostu zainteresowania uczeniem głębokim w ciągu kilku ostatnich lat jest gwałtowne zwiększenie się ilości dostępnych danych. Dwadzieścia lat temu sieci neuronowe były uczone przy wykorzystaniu tysięcy obrazów, obecnie firmy takie jak Facebook czy Google do uczenia swoich sieci używają miliardów obrazów.

Dzięki dostępowi do wszystkich informacji pochodzących od użytkowników ci internetowi giganci bez wątpienia znajdują się na uprzywilejowanej pozycji w dziedzinie uczenia głębokiego. Niemniej jednak w internecie istnieje wiele ogólnodostępnych źródeł danych, które przy niewielkim nakładzie pracy z powodzeniem można wykorzystać do uczenia własnych sieci. W tym podrozdziale przedstawię najważniejsze spośród tych źródeł. W przypadku każdego z nich opiszę, jak można pobierać dane, a także które z popularnych bibliotek mogą nam pomóc w przetwarzaniu tych danych oraz jakie są typowe sposoby korzystania z nich. Oprócz tego w dalszej części książki wskażę receptury, które z tych źródeł korzystają.

Wikipedia

Angielska Wikipedia nie tylko zawiera ponad 5 milionów artykułów, lecz także materiały w setkach innych języków (*https://en.wikipedia.org/wiki/List_of_Wikipedias*), choć ich dokładność i jakość bywają różne. Podstawowa idea wiki obejmowała tylko odnośniki jako jedyny sposób określania struktury, jednak wraz z upływem czasu możliwości Wikipedii nieco się poszerzyły.

Strony kategorii zawierają listy odnośników do stron poświęconych temu samemu zagadnieniu, a ponieważ strony w Wikipedii zawierają odnośniki powrotne do kategorii, z powodzeniem można ich używać jako znaczników. Kategorie mogą być bardzo proste, takie jak „Koty”, lecz czasami w ich nazwach są także zawarte dodatkowe informacje, które można potraktować jak pary klucz – – wartość skojarzone ze stronami, na przykład „Mammals described in 1758”. Niemniej jednak hierarchia kategorii w Wikipedii jest tworzona raczej doraźnie. Co więcej, rekurencyjne kategorie można wykryć, poruszając się wyłącznie w górę drzewa.

Szablony (ang. *templates*) zostały pierwotnie zaprojektowane jako segmenty kodu stron wiki, które miały być automatycznie dołączane (ang. *transcluded*) na stronach. Można je dodawać poprzez umieszczenie nazwy takiego szablonu wewnątrz `{{dwóch par nawiasów klamrowych}}`. To rozwiązanie pozwoliło zachować spójny wygląd stron Wikipedii; na przykład wszystkie strony poświęcone miastom mają ramkę zawierającą informacje o populacji, lokalizacji oraz flagę bądź herb miejscowości, dzięki czemu są zawsze wyświetlane w taki sam, spójny sposób.

Szablony te mają parametry (takie jak wielkość populacji) i można je postrzegać jako sposób osadzania na stronach Wikipedii informacji strukturalnych. W rozdziale 4. wykorzystamy tę cechę Wikipedii, by pobierać filmy, których następnie użyjemy do uczenia systemu sugerowania.

Wikidata

Serwis *Wikidata* to kuzyn Wikipedii, zawierający dane strukturalne. Jest nieco mniej znany i nieco mniej kompletny, jednak ma bardziej ambitny charakter. Został stworzony jako źródło danych, z którego na zasadach licencji *public domain* (domeny publicznej) każdy może korzystać. Właśnie z tego względu stanowi doskonałe źródło ogólnodostępnych i bezpłatnych informacji.

Wszystkie dane w tym serwisie są przechowywane w formie tripletów o postaci (temat, predykat, obiekt). Wszystkie tematy i predykaty mają w serwisie poświęcone im odrębne artykuły, zawierające listę wszystkich dostępnych predykatów dla danego zagadnienia. Obiekty mogą być wpisami Wikidata lub literałami, takimi jak łańcuchy, liczby czy też daty. Inspiracją dla tej struktury są wczesne idee semantycznej sieci.

Wikidata posiada także swój własny język zapytań, przypominający nieco SQL i uzupełniony o pewne interesujące rozszerzenia. Na przykład:

```
SELECT ?item ?itemLabel ?pic
WHERE
{
        ?item wdt:P31 wd:Q146 .
        OPTIONAL {
               ?item wdt:P18 ?pic
        }
        SERVICE wikibase:label {
                bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en"
        }
}
```

Powyższe zapytanie pobiera serię danych o kotach wraz ze zdjęciami. Wszystko, co zaczyna się od znaku zapytania, jest traktowane jako zmienna. Zapis `wdt:P31`, czy też właściwość 31, oznacza „jest instancją”, natomiast zapis `wd:Q146` to klasa kotów domowych. A zatem czwarty wiersz zapytania zapisuje w `item` wszystko, co jest instancją kota. Klauzula `OPTIONAL{ … }` stara się następnie wyszukać obraz dla danego elementu. A w końcu ostatni magiczny fragment zapytania stara się znaleźć etykietę dla elementu, stosując przy tym mechanizm automatycznego doboru języka i używając języka angielskiego jako opcji awaryjnej.

W rozdziale 10. będziemy używać Wikidaty i Wikipedii do pobierania kanonicznych obrazów dla kategorii na potrzeby mechanizmu odwrotnego wyszukiwania obrazów.

OpenStreetMap

Serwis *OpenStreetMap* to Wikipedia świata map. O ile w przypadku Wikipedii idea jest taka, że gdyby każdy umieścił w Wiki wszystko, co wie, to uzyskalibyśmy najlepszą z możliwych encyklopedii, o tyle w przypadku OpenStreetMap (OSM) pomysł zakłada, że jeśli każdy umieści w serwisie wszystkie drogi, które zna, to uzyskamy najlepszy na świecie system map. Godne uwagi jest to, że oba te pomysły bardzo dobrze zdały egzamin w praktyce.

Choć pokrycie OSM jest raczej nierównomierne — zdarzają się bowiem obszary, w których map niemal w ogóle nie ma, a są i takie, w których z powodzeniem konkuruje on z mapami Google, a może nawet przewyższa je pod względem liczby szczegółów — to jednak sama ilość danych oraz fakt, że serwis OSM jest dostępny bezpłatnie, sprawiają, że jest to doskonały zasób dla wszelkiego rodzaju projektów o charakterze geograficznym.

OSM można pobrać bezpłatnie w formacie binarnym lub w formie ogromnego pliku XML. Dane obejmujące cały świat zajmują dziesiątki gigabajtów, jednak jeśli poszukujemy mniejszych fragmentów, to w internecie można także znaleźć witryny, w których dane te są podzielone na kraje lub regiony.

Dane binarne oraz XML mają tę samą strukturę: mapa składa się z serii *węzłów*, z których każdy ma *szerokość* i *długość* oraz grupę *dróg* łączących zdefiniowane wcześniej węzły w większe struktury. Do tego dochodzą *zależności*, które łączą wszystkie wymienione wcześniej elementy (czyli węzły, drogi i zależności) w superstruktury.

Węzły służą do reprezentowania punktów na mapie i określania ich cech, jak również do definiowania kształtów dróg. Drogi są używane do tworzenia prostych kształtów, takich jak budynki i segmenty ulic. I w końcu zależności są używane do tworzenia wszystkiego, co zawiera więcej niż jeden kształt, aż do bardzo dużych struktur, takich jak linie brzegowe lub granice państw.

W dalszej części książki przyjrzymy się modelowi, który pobiera obrazy satelitarne oraz wyrenderowane mapy i stara się automatycznie rozpoznawać na nich drogi. Dane, których użyjemy, nie będą pochodzić konkretnie z mapy OSM, jednak właśnie do takich zastosowań OSM jest używany w rozwiązaniach związanych z uczeniem głębokim. Na przykład projekt *images-to-OSM* (*https://github.com/jremillard/images-to-osm*) pokazuje, w jaki sposób można nauczyć sieć neuronową wyodrębniania na zdjęciach satelitarnych kształtów boisk sportowych w celu poprawienia samych map OSM.

Twitter

Twitter zapewne miałby problemy, gdyby konkurował z Facebookiem jako sieć społecznościowa, jednak jako źródło tekstów do uczenia głębokiego modeli nadaje się znacznie lepiej. API Twittera jest dopracowane i pozwala na korzystanie z serwisu we wszelkiego rodzaju aplikacjach. Dla początkujących hackerów uczenia głębokiego najbardziej interesujące będzie zapewne API do obsługi strumienia danych.

Tak zwane Firehose API, udostępniane przez Twitter, przekazuje wszystkie tweety bezpośrednio do klienta. Jak można sobie wyobrazić, w grę wchodzą naprawdę duże ilości danych. Poza tym Twitter pobiera za taki dostęp do danych naprawdę spore kwoty. Nieco mniej znany jest fakt, że Twitter udostępnia także nieco uproszczoną i bezpłatną wersję Firehose API. Zwraca ona jedynie 1% wszystkich tweetów, jednak ta ilość i tak jest wystarczająca dla wielu zastosowań związanych z przetwarzaniem tekstów.

Tweety mają ograniczoną wielkość i są uzupełniane przez interesujące metainformacje, takie jak autor, znacznik czasu, czasami lokalizacja, no i oczywiście znaczniki, obrazy i adresy URL. W rozdziale 7. dowiemy się, jak użyć tego API do utworzenia klasyfikatora prognozującego emoji na podstawie fragmentu tekstu. Podłączymy się do strumienia tweetów i będziemy zachowywać te z nich, które zawierają dokładnie jedną ikonę emoji. Zebranie solidnego zbioru uczącego zajmuje zazwyczaj parę godzin, jednak gdy dysponuje się komputerem ze stabilnym połączeniem internetowym, zostawienie aplikacji działającej przez kilka godzin nie powinno być żadnym problemem.

Twitter jest popularnym źródłem danych w kwestii zagadnień związanych z analizą nastawienia, a prognozowanie emoji można uznać za jeden z wariantów takiej analizy. Danych z tego serwisu z powodzeniem używano jednak także do uczenia modeli przeznaczonych do wykrywania języka, uściślania lokalizacji oraz rozpoznawania nazwanych obiektów.

Projekt Gutenberg

Na długo przed pojawieniem się serwisu *Książki Google*, a w zasadzie na długo przed założeniem Google'a czy nawet przed powstaniem WWW, w 1971 roku został utworzony *Projekt Gutenberg* (*http://www.gutenberg.org/*), którego celem była cyfryzacja książek. Zawiera on pełne teksty ponad 50 tysięcy tytułów, nie tylko powieści, poezji, opowiadań czy dramatów, lecz także książek kucharskich, podręczników oraz różnych periodyków. Większość z tych prac jest dostępna publicznie i można je bezpłatnie pobrać z witryny WWW projektu.

To naprawdę masa tekstów udostępnianych w wygodnym formacie. Jeśli komuś nie przeszkadza to, że większość z nich stanowią teksty dość stare (gdyż nie są już objęte prawami autorskimi), to serwis ten stanowi doskonałe źródło danych do eksperymentów z zakresu przetwarzania tekstów. W rozdziale 5. skorzystamy z Projektu Gutenberg, by pobrać komplet dzieł Szekspira, które posłużą nam jako podstawa do generowania nowych tekstów przypominających prace tego pisarza. Jeśli używamy odpowiedniej biblioteki Pythona, to do pobrania tych książek wystarczy jeden wiersz kodu:

```
shakespeare = strip_headers(load_etext(100))
```

Materiały dostępne w Projekcie Gutenberg są w większości napisane w języku angielskim, choć można także znaleźć niewielką ilość publikacji w innych językach. Początkowo projekt zawierał wyłącznie materiały zapisane w zwykłym kodzie ASCII, jednak z czasem poszerzono go i obecnie obejmuje już kilka różnych sposobów kodowania; pobierając teksty inne niż w języku angielskim, musimy zatem się upewnić, że dysponujemy odpowiednim kodowaniem, gdyż jeszcze nie wszystkie teksty na świecie są zapisane w UTF-8. W rozdziale 8. pobierzemy wszystkie dialogi z zestawu książek dostępnych w serwisie Projekt Gutenberg, a następnie nauczymy chatbota naśladować te konwersacje.

Flickr

Flickr (*https://www.flickr.com/*) to serwis do dzielenia się zdjęciami, uruchomiony w 2004 roku. Został stworzony jako projekt poboczny dla internetowej gry typu MMO o nazwie *Game Neverending*. Okazało się, że sama gra nie przynosi dochodów, jednak twórcy firmy zauważyli, że jej dział zajmujący się udostępnianiem zdjęć zaczyna dobrze sobie radzić, dokonali zatem *zwrotu* — czyli całkowicie zmienili podstawowy profil działalności firmy. Rok później Flickr został sprzedany firmie Yahoo.

Spośród licznych serwisów do dzielenia się zdjęciami Flickr z kilku powodów wyróżnia się jako użyteczne źródło zdjęć do prowadzenia eksperymentów z zakresu uczenia głębokiego.

Pierwszy powód jest taki, że Flickr istnieje już bardzo długo i zdążył zgromadzić miliardy zdjęć. Choć być może liczba ta jest niewielka w porównaniu z tym, ile zdjęć publikują co miesiąc użytkownicy Facebooka, jednak ponieważ użytkownicy udostępniają w tym serwisie zdjęcia, z których są dumni, są one zazwyczaj lepszej jakości i bardziej godne ogólnego zainteresowania.

Drugim powodem są stosowane licencje. Użytkownicy serwisu Flickr wybierają licencje regulujące zasady, na jakich będą udostępniane ich zdjęcia, a wielu z nich wybiera jedną z wersji licencji *Creative Commons* (*https://creativecommons.org/*), które pozwalają na pewne sposoby korzystania ze zdjęć bez konieczności pytania o pozwolenie. Choć zazwyczaj nie jest to nam potrzebne, jeśli używamy zdjęć tylko do uczenia naszego nowego, świetnego algorytmu i interesuje nas jedynie efekt

końcowy. Licencje nabierają jednak kluczowego znaczenia, kiedy musimy publikować używane zdjęcia lub ich zmodyfikowane wersje. A Flickr właśnie na to pozwala.

Ostatnią, choć zapewne zarazem najważniejszą zaletą serwisu Flickr w porównaniu z jego konkurentami jest API. Podobnie jak w przypadku Twittera, także API Flickra jest bardzo dobrze przemyślane, korzysta z technologii REST i pozwala na wykorzystywanie wszystkich możliwości serwisu w sposób programowy. I podobnie jak to jest w przypadku Twittera, istnieją także biblioteki Pythona ułatwiające korzystanie z tego API, co w ogromnym stopniu ułatwia prowadzenie eksperymentów. Wszystkim, czego potrzeba, jest odpowiednia biblioteka i klucz do API serwisu Flickr.

W kontekście tej książki najważniejszymi możliwościami tego API są wyszukiwanie zdjęć oraz ich pobieranie. Wyszukiwanie jest całkiem wszechstronne i obejmuje większość możliwości dostępnych w witrynie serwisu, choć niestety brakuje w nim kilku bardziej zaawansowanych filtrów. Jeśli chodzi o pobieranie, to zdjęcia są udostępniane w wielu różnych rozmiarach. Często najlepiej jest zacząć od zdjęć o niewielkich rozmiarach, a dopiero później zabrać się za przetwarzanie większych.

W rozdziale 9. zastosujemy API Flickra, by pobrać dwa zestawy zdjęć — jeden z psami, a drugi z kotami — i nauczyć klasyfikator różnic między nimi.

Internet Archive

Deklarowaną misją „internetowego archiwum" — *Internet Archive* (*https://archive.org/*) — jest zapewnianie „ogólnego dostępu do wszelkiej wiedzy". Projekt ten jest zapewne najbardziej znany ze względu na mechanizm Wayback Machine — internetowy interfejs pozwalający na przeglądanie stron WWW z różnych momentów w czasie. Zawiera on blisko 300 miliardów migawek (pierwsze pochodzą z 2001 roku), które są zapisane w indeksie określanym w terminologii serwisu jako „trójwymiarowy".

To internetowe archiwum jest jednak czymś więcej niż tylko Wayback Machine — zawiera ono ogromny zbiór dokumentów, multimediów oraz danych obejmujących wszystko, zaczynając od książek, które nie są już objęte prawami autorskimi, przez zdjęcia NASA, aż po okładki płyt CD z muzyką i materiałami wideo. Cały ten zbiór wart jest przeglądania i często może zainspirować do utworzenia nowego projektu.

Jednym z interesujących przykładów jest zbiór wszystkich komentarzy z serwisu Reddit do 2015 roku, zawierający ponad 50 milionów wpisów. Początkowo powstał on jako projekt jednego z użytkowników serwisu Reddit, który cierpliwie używał jego API do pobierania wszystkich komentarzy, po czym ogłosił to w serwisie. Kiedy padło pytanie, gdzie przechowywać te wszystkie dane, okazało się, że internetowe archiwum jest dobrym rozwiązaniem (choć dokładnie te same dane można znaleźć w usłudze BigQuery firmy Google, gdzie jeszcze łatwiej można poddawać je analizom).

Przedstawionym w tej książce przykładem wykorzystania tego serwisu jest *zestaw pytań z serwisu Stack Exchange* (*https://archive.org/details/stackexchange*). Ponieważ serwis Stack Exchange zawsze korzystał z licencji Creative Commons, nic nie stałoby na przeszkodzie, byśmy sami skopiowali te dane, jednak pobranie ich z Internet Archive jest znacznie łatwiejsze. W tej książce użyjemy tego zbioru danych, by nauczyć model dopasowywania odpowiedzi do pytań (patrz rozdział 6.).

Przeglądanie

Jeśli do tworzonego projektu potrzebujemy jakichś specyficznych danych, to istnieje całkiem spore prawdopodobieństwo, że nie będą one łatwo dostępne za pośrednictwem API. A nawet jeśli będzie takie publiczne API, to jego przepustowość może być tak niewielka, że skutecznie przekreśli możliwości jego praktycznego zastosowania. Trudno jest zdobyć historyczne wyniki z naszej ulubionej dyscypliny sportowej. Być może jakaś lokalna gazeta publikuje je w swojej witrynie internetowej, jednak zapewne nie udostępnia żadnego API ani zrzutu danych. Instagram ma dobre API, jednak zmiany wprowadzone ostatnio w warunkach świadczenia usługi sprawiają, że bardzo trudno jest pozyskać duży zestaw danych uczących.

W takich przypadkach zawsze można skorzystać z techniki *web scraping*, określanej także czasami jako „zeskrobywanie". W najprostszym przypadku będzie nam chodzić o pobranie kopii witryny na lokalny komputer, przy czym nie będziemy dysponować żadną wiedzą na temat jej struktury ani postaci używanych adresów URL. W takim przypadku należy zacząć od strony głównej witryny, pobrać wszystkie znajdujące się na niej odnośniki, a następnie w taki sam sposób przetwarzać wszystkie kolejne znajdowane odnośniki aż do momentu, gdy wszystkie zostaną przetworzone. W taki sam sposób robi to Google, tylko na większą skalę. *Scrapy* (*https://scrapy.org/*) to użyteczny framework, którego można używać do operacji tego typu.

Czasami witryna będzie mieć oczywistą strukturę, na przykład witryny poświęcone podróżom zawierają strony o krajach, regionach, miastach w tych regionach, a w końcu o atrakcjach turystycznych w poszczególnych miastach. W takich przypadkach bardziej użyteczne może być stworzenie wyspecjalizowanego rozwiązania, które będzie sukcesywnie analizować poszczególne warstwy hierarchii, aż do pobrania wszystkich atrakcji.

W innych przypadkach może być dostępne wewnętrzne API, z którego będzie można skorzystać. Wiele witryn koncentrujących się na zawartości wczytuje ogólną strukturę stron, a następnie używa wywołań zwrotnych, by pobierać z serwera dane w formacie JSON i na bieżąco wstawiać je do szablonu strony. Takie rozwiązanie bardzo ułatwia tworzenie stron, których zawartość można w nieskończoność przewijać. Kod JSON zwracany z serwera często ma postać, którą bez trudu można zrozumieć. Dotyczy to także parametrów przesyłanych w żądaniach na serwer. Rozszerzenie o nazwie *Request Maker* (*http://www.bit.ly/request-maker*) przeglądarki Chrome pokazuje wszystkie żądania generowane przez stronę i doskonale nadaje się do sprawdzania, czy pobiera ona coś ciekawego z internetu.

Oczywiście istnieją także witryny, które nie chcą, by ich zawartość była w taki sposób pobierana. Choć firma Google zbudowała swoje imperium na właśnie takim przetwarzaniu zawartości internetu, to jednak wiele jej usług w bardzo inteligentny sposób wykrywa wszelkie próby *web scrapingu* i może blokować wszelkie żądania przychodzące z danego adresu IP, aż do momentu wypełnienia zabezpieczenia captcha. Można próbować zmieniać częstotliwość generowanych żądań oraz nagłówki identyfikujące przeglądarkę, jednak i tak może się okazać, że w którymś momencie trzeba będzie skorzystać z ręcznego wyświetlania stron w przeglądarce.

W takich sytuacjach niezwykle pomocny może być WebDriver — framework opracowany w celu testowania stron WWW z wykorzystaniem instrumentalizacji przeglądarek. Pozwala on na przeglądanie stron przy użyciu wybranej przeglądarki, dzięki czemu, z punktu widzenia serwera, wszystko

wygląda tak, jakby stronę przeglądał rzeczywisty użytkownik. Można także „klikać” w odnośniki przy użyciu skryptów kontrolnych, aby przechodzić na inne strony, a następnie sprawdzać wyniki tych operacji. W skryptach kontrolujących takie poruszanie się po witrynie warto umieszczać opóźnienia czasowe symulujące przeglądanie zawartości przez użytkownika. To powinno w zupełności wystarczyć.

Kod przedstawiony w rozdziale 10. używa technik *web scrapingu* do pobierania obrazów z Wikipedii. Dostępny jest schemat adresu URL pozwalający na pobranie obrazu o określonym identyfikatorze, jednak nie zawsze działa on bezbłędnie. W takich przypadkach pobieramy stronę zawierającą obraz, a następnie analizujemy graf odnośników aż do momentu znalezienia interesującego nas obrazu.

Inne możliwości

Dane można pobierać na wiele sposobów. Witryna *ProgrammableWeb* (*https://www.programmableweb.com/*) zawiera listę ponad 18 tysięcy ogólnodostępnych API (choć niektóre z nich są w złym stanie). Poniżej podałem trzy z nich, na które warto zwrócić uwagę:

Common Crawl
: Przejrzenie jednej witryny jest wykonalne, o ile nie jest ona zbyt duża. Ale co zrobić, jeśli chcemy przejrzeć wszystkie główne witryny w internecie? Serwis *http://commoncrawl.org* co miesiąc pobiera około 2 miliardów stron WWW i udostępnia je w łatwym do przetworzenia formacie. AWS udostępnia te dane jako zbiór publiczny, zatem osoby korzystające z tej platformy dysponują łatwym sposobem wykonywania zadań operujących globalnie na całym internecie.

Facebook
: Wraz z upływem lat API Facebooka zmieniło nieznacznie swój charakter — z użytecznego zasobu pozwalającego na tworzenie aplikacji korzystających z danych Facebooka stało się zasobem pozwalającym na tworzenie aplikacji poprawiających jakość tych danych. Choć jest to zupełnie zrozumiałe z punktu widzenia samego Facebooka, to jednak jako poszukiwacze danych często możemy się zastanawiać, jak dużo danych Facebook może upubliczniać. Niemniej jednak API Facebooka jest użytecznym zasobem, zwłaszcza API Places, które przydaje się w sytuacjach, gdy mapy OSM nie są dość szczegółowe.

Rząd USA
: Rząd USA na wszystkich swoich poziomach publikuje bardzo wiele danych, przy czym wszystkie z nich są publicznie dostępne. Na przykład *dane demograficzne* (*https://www.census.gov/*) zawierają szczegółowe informacje o populacji kraju, natomiast portal *Data.gov* zawiera zbiory danych obejmujących szerokie spektrum tematów. Poza tym poszczególne stany i miasta udostępniają własne zasoby, które również są warte zainteresowania.

1.3. Wstępne przetwarzanie danych

Głębokie sieci neuronowe wyjątkowo dobrze radzą sobie ze znajdowaniem wzorców występujących w danych, co może pomóc w uczeniu modelu prognozowania etykiet, jakie należy przypisywać danym. Oznacza to także, że trzeba bardzo ostrożnie dobierać dane przekazywane uczonym

sieciom — każdy występujący w danych wzorzec, który nie będzie związany z naszym problemem, może sprawić, że sieć nauczy się nie tego, o co nam chodzi. Dzięki właściwemu wstępnemu przetworzeniu danych możemy się upewnić, że w jak największym stopniu uprościmy proces uczenia sieci.

Przygotowywanie zrównoważonego zbioru uczącego

Legendarna już historia opowiada o tym, jak to armia USA starała się nauczyć sieć neuronową odróżniania zamaskowanych czołgów od zwyczajnego lasu — to bardzo użyteczne zadanie w kontekście automatycznego analizowania zdjęć satelitarnych. Na pierwszy rzut oka wszystko zostało zrobione prawidłowo. Pewnego dnia samolot przeleciał nad zamaskowanymi czołgami i zrobił im zdjęcia, a innego dnia zrobiono zdjęcia lasu bez czołgów — zwracano przy tym uwagę, by te zdjęcia nie były identyczne. Zdjęcia podzielono następnie na dwa zbiory: uczący i testowy, po czym rozpoczęto uczenie sieci.

Sieć została pomyślnie nauczona i zaczęła zwracać dobre wyniki. Jednak kiedy badacze wysłali sieć w celu przetestowania jej w praktyce, uznano, że to żart. Prognozy zwracane przez sieć okazały się bowiem bardzo losowe. Po przeprowadzeniu dokładniejszych poszukiwań okazało się, że wystąpił problem związany z danymi wejściowymi. Wszystkie zdjęcia zawierające zamaskowane czołgi zostały zrobione w słoneczny dzień, natomiast zdjęcia przedstawiające sam las — w dzień pochmurny. A zatem choć badacze sądzili, że ich sieć nauczyła się rozpoznawania zamaskowanych czołgów w lesie, tak naprawdę nauczyli ją rozpoznawania pogody na zdjęciach.

Wstępne przetworzenie danych ma na celu upewnienie się, że sieć będzie odbierać sygnały, które chcemy jej przekazać, i nie zostanie rozproszona przez inne sygnały, niemające żadnego znaczenia. Pierwszym krokiem na drodze do realizacji tego celu jest upewnienie się, że dysponujemy odpowiednimi danymi wejściowymi. W optymalnym przypadku dane te powinny w jak największym stopniu odpowiadać rzeczywistym sytuacjom.

Upewnienie się, że sygnał w danych jest tym, którego próbujemy nauczyć sieć, wydaje się czymś oczywistym, jednak łatwo tutaj się pomylić. Zdobywanie danych jest trudne, a każde ich źródło ma swoje szczególne cechy.

Istnieje kilka rzeczy, które można zrobić, gdy okaże się, że nasze dane są zanieczyszczone. Oczywiście najlepszą z nich jest zadbanie o ich odpowiednie zrównoważenie. A zatem, wracając do przykładu z czołgami i lasem, możemy zadbać o zrobienie zdjęć lasu i zamaskowanych czołgów w każdej pogodzie. (Jeśli się nad tym zastanowimy, to dojdziemy do wniosku, że nawet gdyby wszystkie zdjęcia zostały wykonane przy słonecznej pogodzie, to uzyskany zbiór danych i tak nie byłby optymalny — powinien on bowiem zawierać zdjęcia zrobione we wszystkich warunkach pogodowych).

Drugim rozwiązaniem może być odrzucenie części danych, tak by ich zbiór stał się bardziej zrównoważony. Może zrobiono trochę zdjęć zamaskowanych czołgów w pochmurny dzień, ale nie dość dużo — w takim przypadku można by odrzucić niektóre zdjęcia wykonane przy słonecznej pogodzie. Oczywiście takie rozwiązanie zmniejsza wielkość zbioru uczącego, przez co czasami nie będzie można z niego skorzystać. (W takich sytuacjach może nam pomóc technika rozszerzania danych (ang. *data augmentation*) opisana w punkcie „Wstępne przetwarzanie obrazów”).

I w końcu trzecim rozwiązaniem jest poprawianie danych wejściowych, na przykład poprzez wykorzystanie filtra, który sprawi, że warunki pogodowe na zdjęciach będą się wydawać zbliżone. Takie

rozwiązanie jest jednak ryzykowne, gdyż może doprowadzić do wystąpienia innych artefaktów, które sieć może wykrywać.

Tworzenie wsadów danych

Sieci neuronowe używają danych we wsadach (pakietach składających się z par danych wejściowych i wyjściowych). Ważne jest, by upewnić się, że wsady te cechują się odpowiednią losowością. Wyobraźmy sobie, że dysponujemy zbiorem zdjęć: pierwsza połowa tego zbioru to zdjęcia kotów, a druga — zdjęcia psów. Bez odpowiedniego przemieszania zdjęć sieć nie miałaby możliwości nauczenia się czegokolwiek na podstawie tych zbiorów danych: niemal wszystkie pary zawierałyby bądź to tylko zdjęcia kotów, bądź psów. Jeśli będziemy korzystać z biblioteki Keras i jeśli dane te zostaną w całości wczytane do pamięci, to taki efekt ich wymieszania będzie można łatwo osiągnąć przy użyciu jednego wywołania metody `fit`:

```
char_cnn_model.fit(training_data, training_labels, epochs=20, batch_size=128)
```

To wywołanie stworzy losowe wsady o wielkości 128, pobierając do nich dane ze zmiennych `tra ↪ining_data` oraz `training_labels`. Biblioteka Keras zajmie się odpowiednim losowym doborem danych. O ile tylko wszystkie dane są dostępne w pamięci, jest to standardowy sposób postępowania.

W niektórych okolicznościach możemy chcieć wywoływać metodę `fit`, przekazując do niej po jednym wsadzie; w takim przypadku sami musimy zadbać o odpowiednie wymieszanie danych. Z powodzeniem można do tego celu wykorzystać metodę `numpy.random.shuffle`, choć trzeba zadbać o zachowanie powiązań między danymi i ich etykietami.

Jednak nie zawsze wszystkie dane będą dostępne w pamięci. Czasami będzie ich zbyt dużo bądź też będą musiały być przetwarzane na bieżąco i nie będą dostępne w optymalnym formacie. W takich sytuacjach można skorzystać z metody `fit_generator`:

```
char_cnn_model.fit_generator(
    data_generator(train_tweets, batch_size=BATCH_SIZE),
    epochs=20
)
```

W tym przypadku `data_generator` działa jak generator zwracający wsady danych. Musi on zadbać o to, by dane były wybrane w odpowiednio losowy sposób. Jeśli dane te pochodzą z pliku, to ich losowy wybór raczej nie jest możliwy. Jeśli dane pochodzą z dysku SSD, a ich rekordy są dokładnie tej samej wielkości, to możemy wybierać je, przesuwając się losowo w odpowiednie miejsca pliku. W pozostałych przypadkach, jeśli dodatkowo zawartość pliku jest w jakiś sposób posortowana, możemy zwiększyć losowość danych, tworząc kilka uchwytów do tego samego pliku, przy czym każdy z nich będzie wskazywać inną lokalizację.

W przypadku konfigurowania generatora, który ma na bieżąco generować wsady, także trzeba pamiętać o zapewnieniu ich losowości. Na przykład w rozdziale 4. stworzymy system do rekomendowania filmów, ucząc go na podstawie artykułów z Wikipedii i używając jako jednostek uczących odnośników prowadzących ze strony filmu do innych stron w internecie. Najprostszym sposobem wygenerowania takich par (StronaPoczątkowa, StronaDocelowa) byłoby losowe wybranie strony StronaPoczątkowa, a następnie losowe wybranie odnośnika StronaDocelowa spośród wszystkich odnośników umieszczonych na stronie StronaPoczątkowa.

Oczywiście takie rozwiązanie będzie działać, choć spowoduje częstsze wybieranie odnośników ze stron zawierających mniejszą liczbę odnośników niż ze stron, na których tych odnośników jest więcej. Prawdopodobieństwo wybrania w pierwszym kroku strony StronaPoczątkowa, zawierającej tylko jeden odnośnik, jest takie samo jak prawdopodobieństwo wybrania strony zawierającej setki odnośników. Z kolei w drugim kroku ten jedyny odnośnik na pewno zostanie wybrany, natomiast prawdopodobieństwo wyboru jednego z odnośników ze strony, na której były ich setki, jest znacznie mniejsze.

Uczenie, testowanie i weryfikacja danych

Po skonfigurowaniu naszych czystych, znormalizowanych danych, a jednocześnie przed rozpoczęciem fazy uczenia musimy jeszcze podzielić dane na zbiory: uczący, testowy i (byś może także) weryfikacyjny. Zabiegi te, podobnie jak wiele innych związanych z uczeniem sieci neuronowych, są związane z unikaniem nadmiernego dopasowania. Sieci neuronowe prawie zawsze zapamiętują pewną część danych uczących, zamiast uczyć się na podstawie ich uogólniania. Dzięki wydzieleniu niewielkiego fragmentu danych do odrębnego zbioru, którego nie będziemy używać podczas uczenia sieci, możemy następnie zmierzyć, w jakim stopniu sieć zapamiętała dane uczące; po każdej epoce uczenia mierzymy dokładność uzyskiwaną dla zbioru uczącego oraz testowego i jeśli tylko obie te wartości nie będą się zbytnio różnić od siebie, to wszystko będzie w porządku.

Jeśli wszystkie używane dane znajdują się w pamięci, to do podzielenia ich na zbiór uczący i testowy można zastosować metodę `train_test_split` z biblioteki `sklearn`:

```
data_train, data_test, label_train, label_test = train_test_split(
    data, labels, test_size=0.33, random_state=42)
```

Powyższe wywołanie spowoduje utworzenie zbioru testowego zawierającego 33% danych. Zmienna `random_state` jest używana jako losowe ziarno, które gwarantuje, że jeśli dwukrotnie wykonamy ten sam program, to za każdym razem uzyskamy identyczne wyniki.

Ucząc sieć neuronową na podstawie danych przekazywanych przez generator, sami musimy je podzielić. Jednym z ogólnych, choć nie szczególnie wydajnych sposobów przeprowadzenia takiej operacji jest poniższa funkcja:

```
def train_or_test(gen, train=True):
    for i, x in enumerate(gen):
        if (i % 4 == 0) != train:
            yield x
```

Kiedy parametr `train` przyjmie wartość `False`, funkcja będzie zwracać każdy co czwarty element przekazywany przez generator `gen`. Jeśli parametr ten przyjmie wartość `True`, to zwracane będą trzy z każdych czterech elementów.

Czasami z całości dostępnych danych wyodrębniany jest także trzeci zbiór, nazywany *zbiorem weryfikującym* (ang. *validation set*). Nazwa ta może być nieco myląca, gdyż w przypadku stosowania dwóch zbiorów danych zbiór testowy także jest czasami nazywany zbiorem weryfikującym. Jednak w sytuacji gdy mamy zbiór uczący, weryfikujący oraz testowy, to zbiór weryfikujący jest używany do mierzenia wydajności podczas dostrajania modelu. Zbiór testowy powinien być używany jedynie po zakończeniu dostrajania, kiedy w kodzie nie będą już wprowadzane żadne zmiany.

Powodem wprowadzenia tego trzeciego zbioru danych jest próba uchronienia nas przed ręcznym doprowadzeniem do nadmiernego dopasowania sieci. W złożonych sieciach neuronowych liczba opcji dostrajania, czy też hiperparametrów, może być bardzo duża. Znajdowanie odpowiednich wartości dla tych hiperparametrów jest problemem optymalizacyjnym, który sam w sobie także może być podatny na nadmierne dopasowanie. Parametry sieci będziemy modyfikować do momentu, gdy wydajność jej działania przestanie się zwiększać. Dzięki zastosowaniu trzeciego, testowego zbioru danych, który nie był używany podczas dostrajania sieci, możemy mieć pewność, że niechcący nie zoptymalizowaliśmy hiperparametrów sieci pod kątem zbioru weryfikacyjnego.

Wstępne przetwarzanie tekstów

Wiele problemów, do których rozwiązania są wykorzystywane sieci neuronowe, jest związanych z przetwarzaniem tekstów. W takich sytuacjach wstępne przetwarzanie tekstów wejściowych jest związane z ich odwzorowywaniem na wektor lub macierz, które będzie można przekazać do sieci.

Zazwyczaj pierwszym krokiem jest podział tekstu na jednostki, przy czym najczęściej stosowane są dwa podstawowe rozwiązania: podział na słowa lub na litery.

Podział tekstu na strumień poszczególnych znaków jest bardzo prostym zadaniem i daje przewidywalną liczbę różnych tokenów. Jeśli cały nasz tekst jest zapisany na podstawie jednego fonemu, to liczba różnych tokenów jest bardzo ograniczona.

Podział tekstu na słowa jest nieco bardziej złożoną strategią, zwłaszcza w przypadku języków, w których początki i końce słów nie są oznaczane. Co więcej, w takiej sytuacji nie ma żadnej górnej granicy liczby różnych tokenów, które uzyskamy w wyniku podziału tekstu. Wiele zestawów narzędziowych służących do przetwarzania tekstów dysponuje funkcją do podziału (ang. *tokenize*), która zazwyczaj pozwala także na usuwanie znaków z akcentami oraz opcjonalną konwersję znaków w tokenach na małe litery.

Proces określany jako *stemming*, polegający na konwersji słów na ich formy podstawowe (poprzez odrzucenie wszelkich gramatycznych końcówek), także może się okazać pomocny, zwłaszcza w językach, które są pod względem gramatycznym bardziej skomplikowane od języka angielskiego. W rozdziale 8. przedstawiona zostanie strategia podziału podsłów, która będzie polegać na podziale bardziej złożonych słów na podtokeny, co zapewni konkretny górny limit liczby różnych tokenów.

Po wykonaniu podziału tekstu na tokeny można przystąpić do ich wektoryzacji. Najprostszym rozwiązaniem jest przypisywanie poszczególnym tokenom wartości liczby całkowitej *i* z zakresu od 0 do liczby dostępnych tokenów, a następnie wyrażenie każdego tokena w formie wektora, który wszędzie będzie zawierał 0, z wyjątkiem *i*-tego elementu, którego wartością będzie 1. Metoda ta jest określana jako *kodowanie zerojedynkowe* (ang. *one-hot encoding*), a w Pythonie można ją wykonać przy użyciu poniższego fragmentu kodu:

```
idx_to_token = list(set(tokens))
token_to_idx = {token: idx for idx, token in enumerate(idx_to_token)}
one_hot = lambda token: [1 if i == token_to_idx[token] else 0
                         for i in range(len(idx_to_token))]
encoded = np.asarray([one_hot(token) for token in tokens])
```

Ten kod zwróci dużą dwuwymiarową tablicę, gotową do przekazania do sieci neuronowej.

Rozwiązanie to będzie działać w sytuacji, gdy przetwarzamy tekst na podstawie znaków. Można z niego także korzystać podczas operowania na słowach, jednak w przypadku tekstów zawierających duże ilości słów stanie się ono niewygodne. Istnieją jednak dwie strategie kodowania, które pozwalają ominąć ten problem.

Pierwszą z nich jest traktowanie dokumentu jako „worka ze słowami". W tym przypadku nie zwracamy uwagi na kolejność występowania słów — interesuje nas tylko sam fakt wystąpienia słowa. Dokument możemy wyrazić jako wektor, którego poszczególne elementy będą odpowiadać poszczególnym tokenom. W najprostszym przypadku w wektorze tym będziemy umieszczać 1, jeśli dany token występuje w dokumencie, lub 0 w przeciwnym razie.

Ponieważ w języku angielskim 100 najczęściej występujących słów stanowi przeważnie około połowy wszystkich tekstów, zatem nie najlepiej nadają się one do zadań klasyfikacji — będą się bowiem znajdować niemal we wszystkich dokumentach, a więc uwzględnianie ich w wektorze do niczego się nam raczej nie przyda. Popularne jest zatem rozwiązanie polegające na usuwaniu tych słów z „worka", dzięki czemu sieć neuronowa będzie mogła skoncentrować się na słowach, które rzeczywiście mają znaczenie.

Bardziej skomplikowaną wersją tego rozwiązania jest ważenie termów — odwrotna częstotliwość w dokumentach (TF-IDF). W tej metodzie zamiast zapisywać w wektorze 1, jeśli dany token istnieje w dokumencie, zapisujemy w nim względną częstotliwość występowania danego termu w dokumencie w stosunku do tego, jak często występuje on w całym korpusie dokumentów. Intuicyjnie rzecz biorąc, chodzi tu o to, że większe znaczenie ma wystąpienie w dokumencie tokena pojawiającego się rzadziej niż takiego, który jest cały czas. Biblioteka scikit-learn udostępnia metody pozwalające na automatyczne obliczanie tych wartości.

Drugim sposobem radzenia sobie z kodowaniem słów jest wykorzystanie wektorów właściwościowych (ang. *word embeddings*). Temu rozwiązaniu jest poświęcony cały rozdział 3. niniejszej książki. Zamieszczone w tym rozdziale informacje pozwolą zrozumieć sposób działania tego rozwiązania. W przypadku korzystania z wektorów właściwościowych z każdym tokenem kojarzony jest wektor o określonej wielkości — zazwyczaj mieści się ona w granicach od 50 do 300. Kiedy sieci neuronowej przekażemy dokument reprezentowany jako sekwencja identyfikatorów tokenów, jej warstwa właściwościowa (ang. *embedding layer*) automatycznie odszuka odpowiednie wektory właściwościowe i zwróci dwuwymiarową tablicę.

Warstwa ta nauczy się odpowiednich wag dla każdego z termów, zupełnie tak samo jak każda inna warstwa w sieci neuronowej. Czasami wymaga to wiele uczenia, i to zarówno pod względem przetwarzania, jak i niezbędnych ilości danych. Wspaniałą cechą tego rozwiązania jest to, że istnieją gotowe, nauczone zestawy wektorów właściwościowych, które można pobrać i wykorzystać do przygotowania warstwy właściwościowej własnej sieci. Przykład takiego rozwiązania można znaleźć w rozdziale 7.

Wstępne przetwarzanie obrazów

Głębokie sieci neuronowe okazały się bardzo wydajnym narzędziem do przetwarzania obrazów, i to w różnym zakresie — zaczynając od rozpoznawania kotów w klipach wideo, a kończąc na nadawaniu zdjęciom typu selfie stylu różnych artystów. Jednak, podobnie jak w przypadku tekstów, także i obrazy muszą zostać w odpowiedni sposób wstępnie przetworzone.

Pierwszym krokiem jest normalizacja. Wiele sieci operuje jedynie na obrazach o określonej wielkości, dlatego też pierwszym krokiem jest przycięcie lub przeskalowanie obrazów do odpowiednich, docelowych wymiarów. Często stosowane jest zarówno przycinanie względem punktu środkowego, jak i bezpośrednia zmiana wymiarów, choć czasami najlepiej sprawdza się połączenie obu tych czynności, gdyż pozwala zachować więcej z obrazu i zapewnić większą kontrolę nad zniekształceniami związanymi ze skalowaniem.

Normalizacja kolorów polega zazwyczaj na odjęciu od każdego piksela wartości średniej i podzieleniu wyniku przez odchylenie standardowe. To sprawia, że większość wartości grupuje się wokół 0, a prawie 70% z nich znajduje się w wygodnym zakresie [–1, 1]. Nowym rozwiązaniem związanym z normalizacją kolorów jest tak zwana *normalizacja wsadowa* (ang. *batch normalization*), która polega na tym, że dane nie są normalizowane od razu w całości, lecz od wartości pikseli odejmowana jest średnia wsadu, a wynik jest dzielony przez odchylenie standardowe. Takie rozwiązanie zapewnia lepsze wyniki i można go używać jako jednego z fragmentów sieci.

Rozszerzanie danych (ang. *data augmentation*) jest strategią służącą do zwiększania ilości danych uczących poprzez dodawanie zmodyfikowanych obrazów ze zbioru uczącego. Jeśli dodamy do danych uczących wersje obrazów stanowiących ich lustrzane odbicie, to możemy podwoić wielkość zbioru uczącego — w końcu lustrzane odbicie kota to wciąż kot. Jednak z innego punktu widzenia takie rozwiązanie to w zasadzie pokazanie sieci, że odbicia lustrzane można zignorować. Jeśli na wszystkich zdjęciach koty będą się patrzeć w jedną stronę, to sieć może uznać, że jest to jedna z cech bycia kotem — dodanie lustrzanych odbić uniemożliwi wyciągnięcie takich wniosków.

Biblioteka Keras udostępnia wygodną klasę `ImageDataGenerator`, którą można skonfigurować tak, by wykonywała wiele różnych modyfikacji obrazów, w tym obroty, przesunięcia, modyfikacje kolorów oraz powiększenia. Można jej używać jako generatora danych dla metody `fit_generator` modelu:

```
datagen = ImageDataGenerator(
    rotation_range=20,
    horizontal_flip=True)

model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs)
```

Wnioski

Wstępne przetwarzanie danych jest bardzo ważnym krokiem, który należy wykonywać przed przystąpieniem do głębokiego uczenia modelu. Wszelkie takie operacje mają na celu jak największe ułatwienie sieciom uczenia się właściwych rzeczy i niedopuszczenie do tego, by sieć została rozproszona przez nieistotne cechy występujące w danych wejściowych. Przygotowanie zrównoważonego zbioru uczącego, tworzenie odpowiednio losowych wsadów oraz różne sposoby normalizacji — to wszystko są operacje zaliczane do ogólnie pojętego wstępnego przetwarzania danych.

Skorowidz

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

O'REILLY®

Uczenie głębokie – rzecz dla kreatywnych filozofów!

Pomysł, by komputery wykorzystywać do generowania inteligentnych rozwiązań, narodził się w zamierzchłych dla informatyki czasach, mniej więcej w połowie XX wieku. Bardzo długo jednak idea ta – z powodu ograniczeń technologicznych – nie mogła wyjść poza rozważania teoretyczne. Dziś osoby zainteresowane uczeniem głębokim są w komfortowej sytuacji: mogą korzystać z ogólnie dostępnych frameworków uczenia głębokiego, sięgać po ogromne zbiory danych, a ponadto wyniki tego rodzaju badań znalazły się w centrum zainteresowania biznesu. Okazuje się, że nawet bez szczególnego przygotowania teoretycznego można budować i udoskonalać potężne modele sieci neuronowych oraz uczenia głębokiego i wdrażać je w konkretnych sytuacjach.

Dzięki tej książce, nawet jeśli nie posiadasz zaawansowanej wiedzy o uczeniu głębokim (oryg. *deep learning*), zaczniesz szybko tworzyć rozwiązania z tego zakresu. Zamieszczone tu receptury pozwolą Ci sprawnie zaznajomić się z takimi zastosowaniami uczenia głębokiego jak klasyfikacja, generowanie tekstów, obrazów i muzyki. Cennym elementem książki są informacje o rozwiązywaniu problemów z sieciami neuronowymi – testowanie sieci wciąż jest trudnym zagadnieniem. Ponadto znalazły się w niej porady dotyczące pozyskiwania danych niezbędnych do trenowania sieci, a także receptury, dzięki którym łatwiej zacząć użytkować modele w środowiskach produkcyjnych.

Douwe Osinga – jest doświadczonym inżynierem oprogramowania. Pochodzi z Haarlem w Holandii, do niedawna pracował dla Google w Zurychu, Hajdarabadzie i Sydney. Założył trzy startupy. Jest typowym globtroterem; zawodowo uwielbia wyzwania, które zmuszają go do łączenia technologii i wyników własnej pomysłowości.

Z tej książki dowiesz się, jak: :

- tworzyć użyteczne aplikacje, które docenią użytkownicy
- obliczać podobieństwo tekstów
- wizualizować wewnętrzny stan systemu sztucznej inteligencji
- napisać usługę odwrotnego wyszukiwania obrazów za pomocą wyuczonych sieci
- wykorzystać sieci GAN, autoenkodery i LSTM do generowania ikon
- wykrywać style w utworach muzycznych

ISBN 978-83-283-5231-5
9 788328 352315
Cena: 49,00 zł