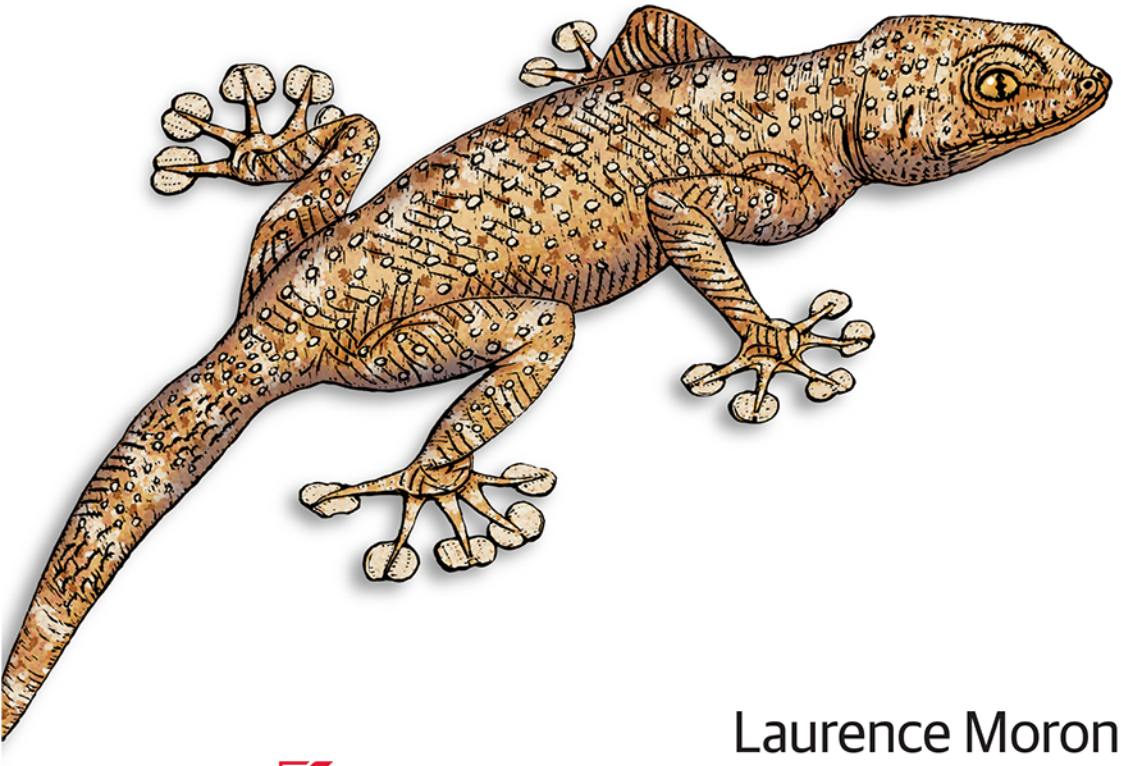
O'REILLY®

Sztuczna inteligencja i uczenie maszynowe dla programistów

Praktyczny przewodnik po sztucznej inteligencji



Laurence Moroney
Przedmowa: Andrew Ng

Helion

Tytuł oryginału: AI and Machine Learning for Coders: A Programmer's Guide to Artificial Intelligence

Tłumaczenie: Jacek Janusz

ISBN: 978-83-283-7850-6

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 7.

Użycie rekurencyjnych sieci neuronowych do przetwarzania języka naturalnego

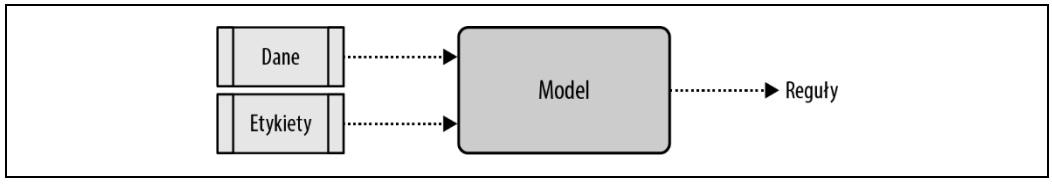
Z rozdziału 5. wiesz, w jaki sposób tekst można przekształcać na ciągi tokenów. Po zamianie zdań na tensory, składające się z liczb, można je następnie wprowadzić do sieci neuronowej. W kolejnym rozdziale przeanalizowaliśmy osadzenia, które pozwalały zestawiać słowa o podobnym znaczeniu, aby umożliwić określanie rodzaju emocji. Z sukcesem wdrożyliśmy pomysł w praktyce poprzez zbudowanie klasyfikatora sarkazmu. Nie wzięliśmy jednak pod uwagę tego, że zdania nie są po prostu zbiorami niezależnych od siebie słów. Kolejność, w jakiej pojawiają się słowa, ma często bardzo wielki wpływ na znaczenie zdania. Na przykład przymiotniki mogą modyfikować znaczenie rzeczowników, obok których się pojawiają. Samo słowo „blue" może nie mieć znaczenia emocjonalnego, podobnie zresztą jak „sky". Gdy je jednak połączysz, aby uzyskać wyrażenie „blue sky", pojawia się wyraźny oddźwięk emocjonalny, który jest zwykle pozytywny. Niektóre rzeczowniki mogą wpływać na inne, na przykład „rain cloud", „writing desk" czy „coffee mug".

Aby wziąć pod uwagę takie związki, należy uwzględnić *rekurencję* w architekturze modelu. W tym rozdziale przeanalizujemy różne rozwiązania. Dowiesz się, w jaki sposób model może zdobyć wiedzę o kolejności słów, a dzięki temu lepiej zrozumieć tekst. W tym celu wykorzystamy **rekurencyjną sieć neuronową** (RNN).

Podstawy rekurencji

Aby zrozumieć, na czym polega działanie rekurencji, zastanówmy się najpierw nad ograniczeniami modeli zaprezentowanymi do tej pory w książce. Sposób tworzenia modelu przedstawiłem na rysunku 7.1. Dostarczasz dane i etykiety oraz definiujesz architekturę modelu, który uczy się reguł dopasowujących dane do etykiet. Reguły są dostępne w postaci interfejsu API, który zwraca prognozowane etykiety dla przyszłych danych.

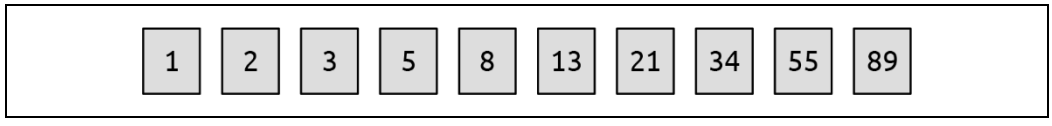
Jak widać, dane do modelu są dostarczane hurtowo. Nie został podjęty żaden wysiłek, aby zrozumieć kolejność, w jakiej się one pojawiają. Oznacza to, że znaczenie słów „blue" i „sky" jest identyczne w zdaniach takich jak „Today I am blue, because the sky is gray" i „Today I am happy, and there's a beautiful blue sky". Czytelnicy, którzy znają język angielski, od razu zauważyli różnicę w użyciu tych słów, ale model z zaprezentowaną wcześniej architekturą tak naprawdę widzi to samo.



Rysunek 7.1. Wysokopoziomowy schemat tworzenia modelu

Jak można rozwiązać ten problem? Najpierw zbadajmy naturę rekurencji, a dzięki temu będziemy mogli zrozumieć, jak działa podstawowa sieć RNN.

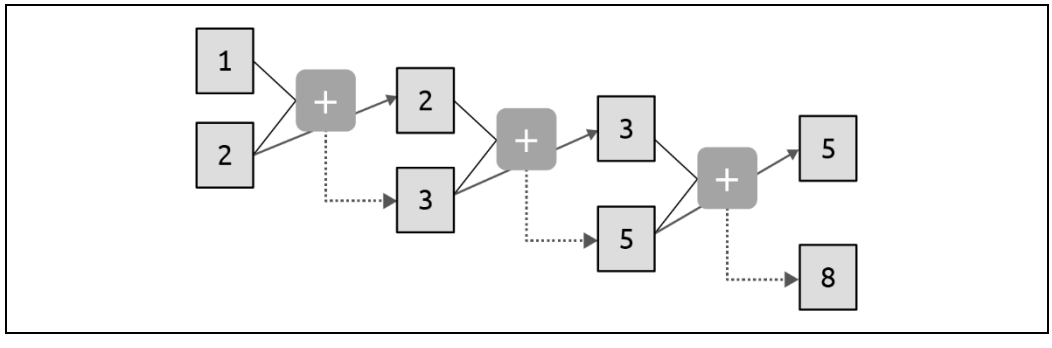
Przyjrzyjmy się słynnemu ciągowi Fibonacciego. Jeśli o nim nie słyszałeś, na rysunku 7.2 możesz zobaczyć kilka początkowych liczb tego ciągu.



Rysunek 7.2. Kilka pierwszych liczb ciągu Fibonacciego

Pomysł polega na tym, że każda kolejna liczba jest sumą dwóch liczb ją poprzedzających. Jeśli zaczniemy od liczb 1 i 2, następną będzie 1 + 2, czyli 3. W dalszej kolejności otrzymamy 2 + 3, czyli 5, 3 + 5, a więc 8, itd.

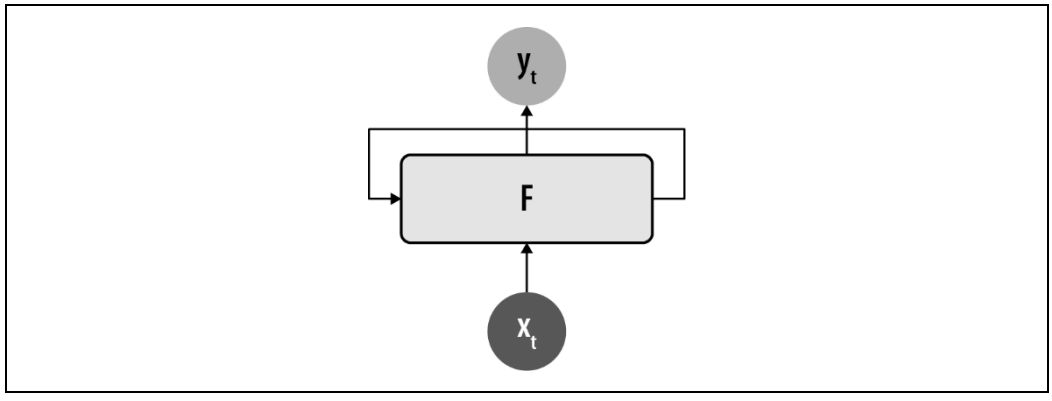
Nasze działania można zaprezentować w sposób graficzny na rysunku 7.3.



Rysunek 7.3. Graficzna reprezentacja ciągu Fibonacciego

Jaki widać, do funkcji przekazujemy liczby 1 i 2, a na wyjściu otrzymujemy 3. Przenosimy drugi parametr (2) do następnego kroku i wprowadzamy go do funkcji wraz z wartością wyjściową z poprzedniego (3). W wyniku otrzymujemy liczbę 5, którą przekazujemy do funkcji razem z drugim parametrem z poprzedniego kroku (3), aby uzyskać wartość 8. Proces ten trwa w nieskończoność, a każda kolejna operacja zależy od wcześniejszych. Można powiedzieć, że wartość 1 pokazana w lewym górnym rogu rysunku w pewien sposób „przetrwa" cały proces. Będzie zawarta w liczbie 3, która zostanie podana do funkcji drugiej, następnie w liczbie 5, która zostanie przekazana do funkcji trzeciej, itd. W ten sposób pewien fragment wartości 1 zostanie zachowany w całej sekwencji, chociaż jej wpływ na ogólny wynik będzie oczywiście niewielki.

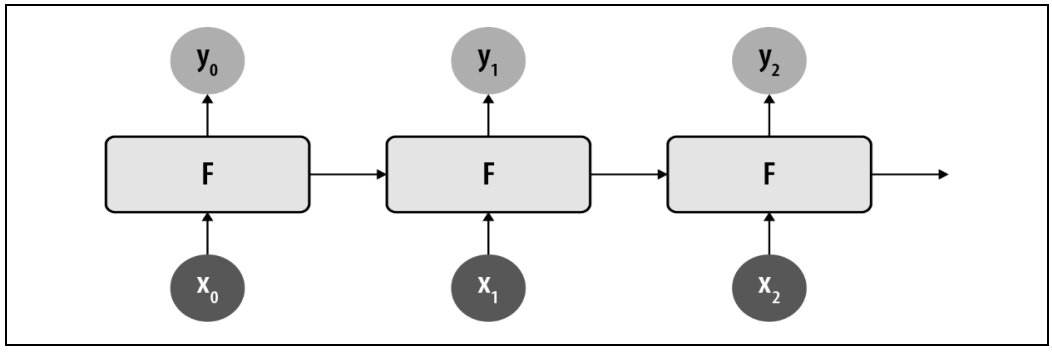
W podobny sposób działa neuron rekurencyjny. Na rysunku 7.4 przedstawiłem typowy schemat połączeń w neuronie wykorzystującym rekurencję.



Rysunek 7.4. Neuron rekurencyjny

W pewnym przedziale czasowym wartość *x* (zazwyczaj oznaczana jako *xt*) zostaje przekazana do funkcji *F*. W wyniku tej operacji w tym samym przedziale czasowym otrzymujemy wartość wyjściową *y* (zwykle oznaczaną jako *yt*). Funkcja zwraca również wartość, która jest przekazywana do następnego kroku (przedstawiona na rysunku jako strzałka prowadząca od funkcji *F* do samej siebie).

Opis stanie się bardziej zrozumiały, jeśli przyjrzysz się, w jaki sposób neurony rekurencyjne współpracują ze sobą w kolejnych przedziałach czasowych (rysunek 7.5).



Rysunek 7.5. Działanie neuronów rekurencyjnych w kolejnych przedziałach czasowych

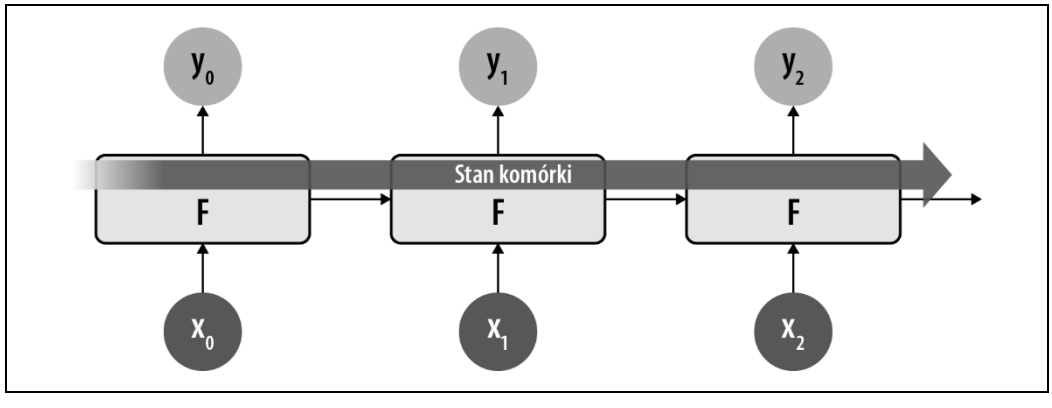
Dostarczywszy neuronowi wartości x_0, uzyskujemy wynik y_0 oraz wartość, która jest przekazywana dalej. Jest ona używana w następnym kroku razem z wartością x_1, powodując utworzenie wyniku y_1 i wartości, która ponownie zostanie dalej przekazana. W kolejnym kroku używamy jej razem z x_2, dzięki czemu tworzymy y_2 oraz wartość przekazywaną, itd. Proces ten jest podobny do tego, który wcześniej zaprezentowałem w przypadku rozwiązywania ciągu Fibonacciego. Warto o nim pamiętać, gdy chcemy sobie przypomnieć, jak działa rekurencyjna sieć neuronowa.

Zastosowanie rekurencji w przetwarzaniu języka naturalnego

Wiesz już, że rekurencyjna sieć neuronowa pozwala na obsługę kontekstu sekwencji w kolejnych przedziałach czasowych. W dalszej części książki pokażę, że sieci RNN rzeczywiście są używane do modelowania sekwencji. Korzystanie z tak prostej rekurencyjnej sieci neuronowej, jaka została przedstawiona na rysunkach 7.4 i 7.5, wiąże się jednak z pewnym problemem, który można przeoczyć. Jak już wspomniałem w przypadku ciągu Fibonacciego, poziom przekazywanego kontekstu będzie z czasem maleć. Wpływ wartości wynikowej otrzymanej z neuronu w kroku 1. jest ogromny, w kroku 2. maleje, w kroku 3. jest jeszcze mniejszy itd. Jeśli więc mamy zdanie typu „Today has a beautiful blue <coś>", wówczas słowo „blue" będzie miało silny wpływ na kolejny wyraz — możemy się domyślić, że prawdopodobnie będzie nim „sky". Co się stanie jednak z kontekstem, który wynika z wcześniejszego fragmentu zdania? Weźmy na przykład zdanie „I lived in Ireland, so in high school I had to learn how to speak and write <coś>".

W miejscu szablonu „<coś>" należy oczywiście wstawić słowo „Gaelic", ale wynika to z odpowiedniego kontekstu, którym jest słowo „Ireland" znajdujące się na samym początku zdania. Abyśmy zatem wiedzieli, czym należy zastąpić szablon „<coś>", musimy wymyślić sposób na zachowanie bardziej odległego kontekstu. Pamięć krótkotrwała sieć RNN musi więc zostać wydłużona, w związku z czym wynaleziono lepszą architekturę zwaną **długą pamięcią krótkotrwałą** (LSTM).

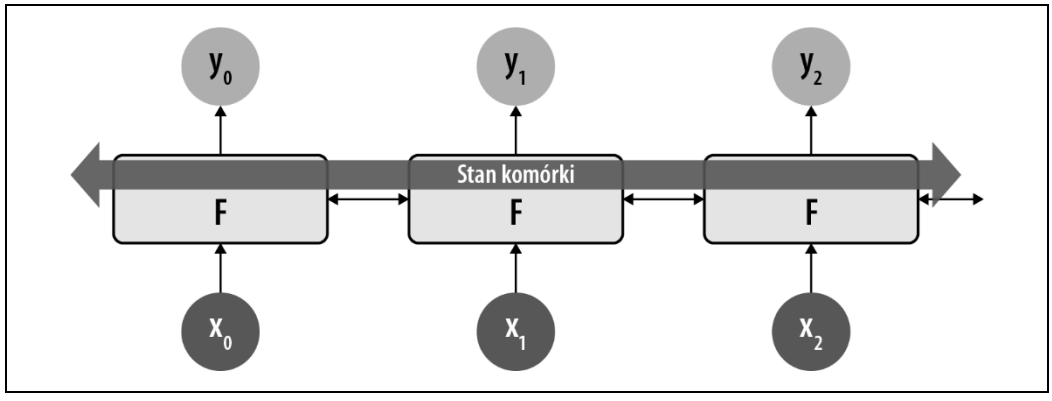
Chociaż nie będę wchodził w szczegóły związane z działaniem podstawowej architektury LSTM, na rysunku 7.6 zaprezentowałem jej wysokopoziomowy schemat. Aby dowiedzieć się więcej o operacjach wewnętrznych, zapoznaj się z doskonałym wpisem na blogu Christophera Olaha (*https://oreil.ly/6KcFA*).



Rysunek 7.6. Wysokopoziomowy schemat architektury LSTM

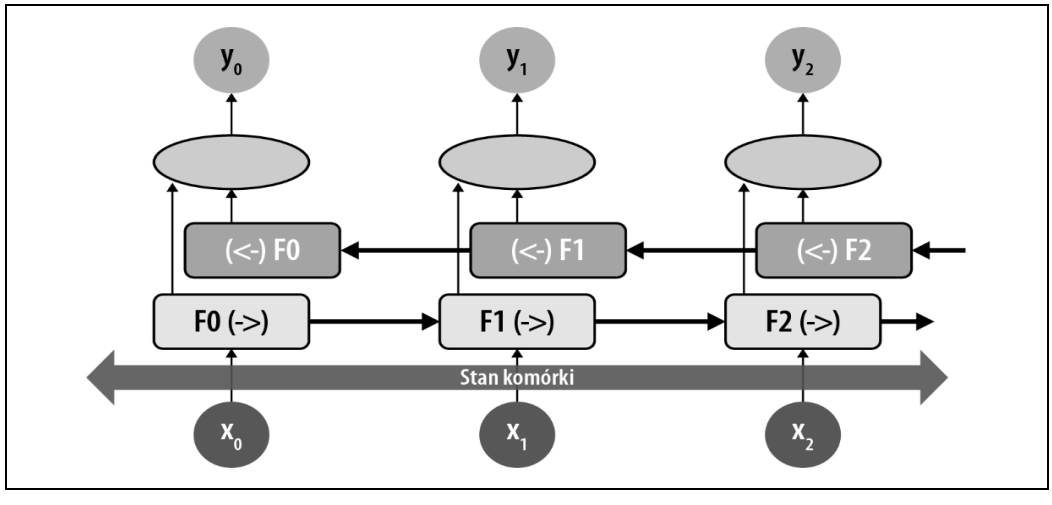
Architektura LSTM usprawnia sposób działania podstawowej sieci RNN poprzez uwzględnienie „stanu komórki", który umożliwia przekazywanie kontekstu nie tylko pomiędzy bezpośrednio występującymi po sobie krokami, ale przez całą ich sekwencję. Biorąc pod uwagę to, że mamy do czynienia z neuronami, dzięki takiemu rozwiązaniu sieć z czasem nauczy się kontekstu.

Ważną cechą architektury LSTM jest to, że może działać w dwóch kierunkach. Informację można przetwarzać w przedziałach czasowych zarówno do przodu, jak i do tyłu, dzięki czemu sieć może poznać kontekst na dwa sposoby. Spójrz na rysunek 7.7, aby się zapoznać z wysokopoziomowym schematem rozwiązania.



Rysunek 7.7. Wysokopoziomowy schemat dwukierunkowej architektury LSTM

Sieć wykonuje ewaluację, poczynając od kroku 0., a kończąc na kroku *maksymalna_liczba_kroków*, jak również odwrotnie — od *maksymalnej_liczby_kroków* do kroku 0. W każdym przedziale czasowym wynikiem *y* jest połączenie przejścia „do przodu" i „wstecz". Odpowiedni schemat można zobaczyć na rysunku 7.8.



Rysunek 7.8. Działanie dwukierunkowej architektury LSTM

Załóżmy, że każdy neuron w danym przedziale czasowym będzie oznaczony symbolem F0, F1, F2 itd. Na rysunku pokazałem kierunki przetwarzania, więc obliczenia dla neuronu F1 w przypadku przejścia do przodu oznaczyłem jako F1(->), a wstecz jako (<-)F1. Aby uzyskać wartość *y* dla określonego przedziału czasowego, sumowane są elementarne wyniki. Również stan komórki jest dwukierunkowy.

Takie rozwiązanie może być naprawdę przydatne do zarządzania kontekstem w zdaniach. W przypadku zdania „I lived in Ireland, so in high school I had to learn how to speak and write <coś>" szablon „<coś>" został zamieniony na słowo „Gaelic" po wykryciu słowa kontekstowego „Ireland". A gdyby było odwrotnie? Weźmy zdanie „I lived in <kraj>, so in high school I had to learn how to speak and write Gaelic". Analizując je od tyłu, możemy się dowiedzieć, czym powinien zostać zastąpiony szablon „<kraj>". Korzystanie z dwukierunkowych architektur LSTM może być bardzo przydatne podczas wykrywania emocji w tekście (z rozdziału 8. dowiesz się, że są one również naprawdę niesamowite podczas generowania tekstów!).

Oczywiście architektury LSTM, a w szczególności dwukierunkowe, są skomplikowane, więc należy oczekiwać, że ich trenowanie będzie przebiegać wolno. W takim przypadku warto zainwestować w procesory graficzne (a przynajmniej skorzystać ze środowiska Google Colab).

Tworzenie klasyfikatora tekstu przy użyciu rekurencyjnych sieci neuronowych

W rozdziale 6. podczas eksperymentowania utworzyliśmy przy użyciu osadzeń klasyfikator dla zbioru danych Sarcasm. Słowa przed agregacją były przekształcane w wektory, a następnie wprowadzane do warstw gęstych w celu klasyfikowania. Podczas korzystania z warstwy RNN takiej jak LSTM nie wykonuje się agregacji, więc dane wyjściowe z warstwy osadzania można przekazywać bezpośrednio do warstwy rekurencyjnej. Jeśli chodzi o jej wymiar, jest on często równy wymiarowi warstwy osadzania. Nie jest to warunek konieczny, ale może być dobrym wyborem na początku. Jak pamiętasz, w rozdziale 6. wspomniałem, że wymiar osadzania jest często czwartym pierwiastkiem rozmiaru słownika. Podczas korzystania z sieci RNN ta reguła jest w wielu przypadkach ignorowana, ponieważ spowodowałaby, że rozmiar warstwy rekurencyjnej byłby zbyt mały.

Zmodyfikujmy prostą architekturę modelu służącego do klasyfikowania sarkazmu, którą wykorzystywaliśmy w rozdziale 6., i użyjmy w niej dwukierunkowej warstwy LSTM:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

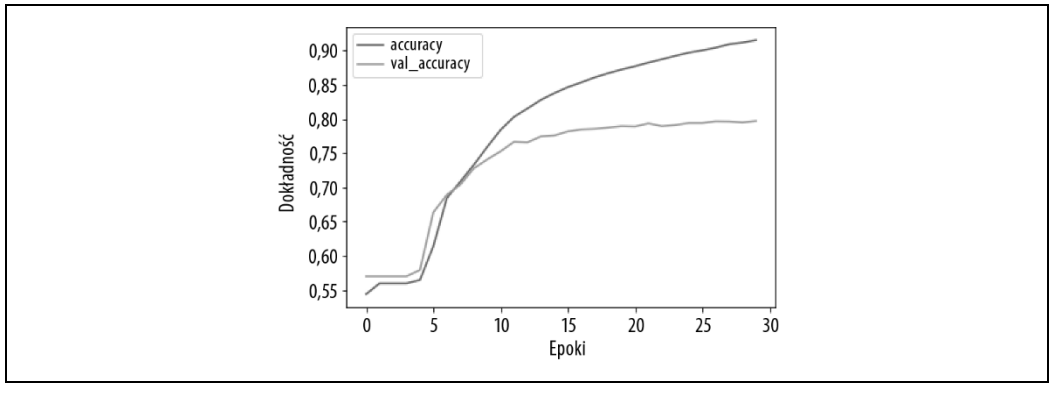
Funkcję straty i klasyfikator można zdefiniować za pomocą następującego kodu (zwróć uwagę, że współczynnik uczenia wynosi 0,00001, co jest równoważne 1e-5):

```
adam = tf.keras.optimizers.Adam(learning_rate=0.00001,
                                beta_1=0.9, beta_2=0.999, amsgrad=False)

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

Spróbuj wyświetlić podsumowanie architektury modelu. Pamiętaj, że rozmiar słownika wynosi 20 000, a wymiar osadzania jest równy 64. W wyniku tego otrzymujemy 1 280 000 parametrów w warstwie osadzania, przy czym warstwa dwukierunkowa ma 128 neuronów (64 w przód, 64 wstecz):

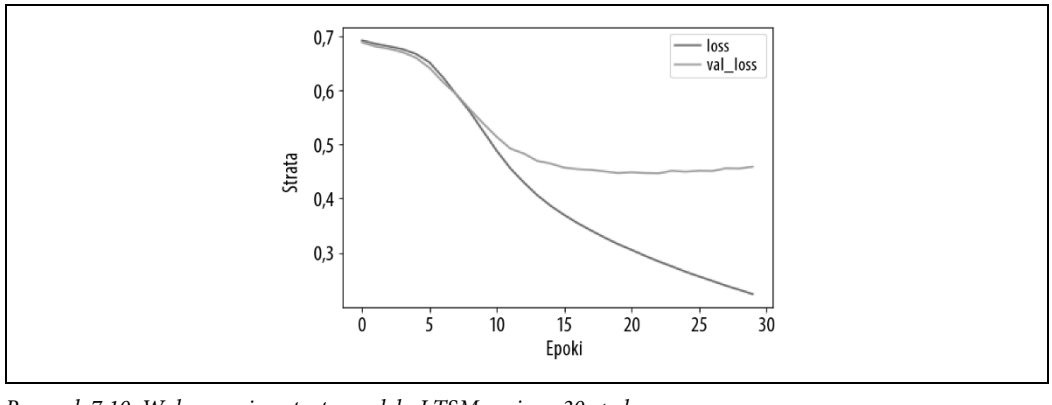
```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_11 (Embedding)     (None, None, 64)          1280000
_________________________________________________________________
bidirectional_7 (Bidirection (None, 128)               66048
_________________________________________________________________
dense_18 (Dense)             (None, 24)                3096
_________________________________________________________________
dense_19 (Dense)             (None, 1)                 25
=================================================================
Total params: 1,349,169
Trainableparams: 1,349,169
Non-trainableparams: 0
_________________________________________________________________
```

Na rysunku 7.9 pokazałem wyniki trenowania modelu przez 30 epok.



Rysunek 7.9. Wykres zmian dokładności modelu LTSM w ciągu 30 epok

Dokładność sieci dla danych treningowych szybko wzrasta do wartości powyżej 90%, jednak w przypadku danych walidacyjnych pozostaje na poziomie około 80%. Podobne wyniki otrzymywaliśmy już wcześniej. Na rysunku 7.10 widzimy, że wykresy strat rozeszły się po mniej więcej 15 epokach. W przypadku danych walidacyjnych wykres uległ spłaszczeniu, a strata, pomimo użycia 20 000 słów zamiast 2000, osiągnęła znacznie niższą wartość niż na wykresach z rozdziału 6.



Rysunek 7.10. Wykres zmian straty modelu LTSM w ciągu 30 epok

Użyliśmy tylko jednej warstwy LSTM. W kolejnym punkcie dowiesz się, w jaki sposób można łączyć ze sobą warstwy LSTM, a także sprawdzisz, jaki wpływ ma to rozwiązanie na dokładność klasyfikowania zbioru danych.

Łączenie warstw LSTM

Z poprzedniego punktu wiesz, że warstwę LSTM można umieścić po warstwie osadzania, aby ulepszyć klasyfikowanie zawartości zbioru danych Sarcasm. Warstwy LSTM można jednak układać jedną na drugiej. Takie rozwiązanie jest stosowane w wielu najnowocześniejszych modelach służących do przetwarzania języka naturalnego.

Układanie warstw LSTM w stos staje się całkiem prostą operacją przy użyciu biblioteki TensorFlow. Dodajesz je po prostu do modelu, tak jak w przypadku warstwy gęstej. Pamiętaj jedynie, że wszystkie warstwy oprócz ostatniej muszą mieć właściwość `return_sequences` równą `True`. Oto przykład:

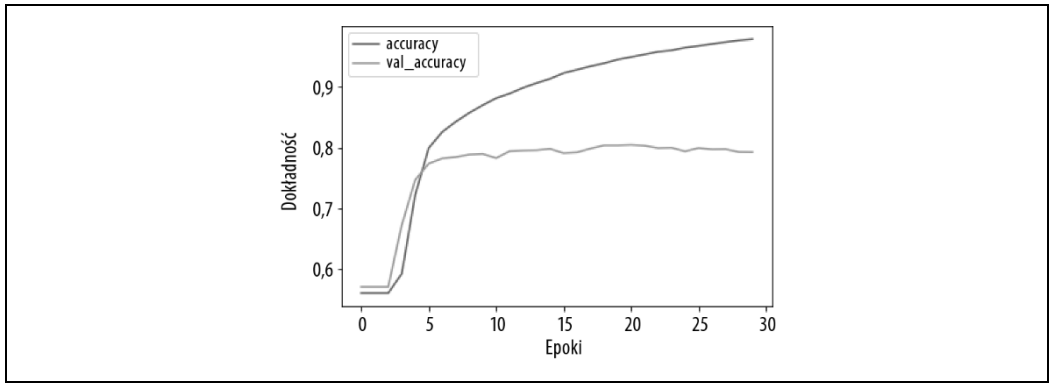
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim, return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Ostatnia warstwa może również używać właściwości `return_sequences` o wartości `True`. W takim przypadku w celu klasyfikowania zwraca ona do warstw gęstych całe sekwencje zamiast pojedynczych wartości. Może to być przydatne podczas analizowania wyników, o czym napiszę później. Architektura modelu wygląda następująco:

```
Layer (type)                 OutputShape              Param #
=================================================================
embedding_12 (Embedding)     (None, None, 64)          1280000
_________________________________________________________________
bidirectional_8 (Bidirection (None, None, 128)         66048
_________________________________________________________________
bidirectional_9 (Bidirection (None, 128)               98816
_________________________________________________________________
dense_20 (Dense)             (None, 24)                3096
_________________________________________________________________
dense_21 (Dense)             (None, 1)                 25
=================================================================
Total params: 1,447,985
Trainableparams: 1,447,985
Non-trainableparams: 0
_________________________________________________________________
```

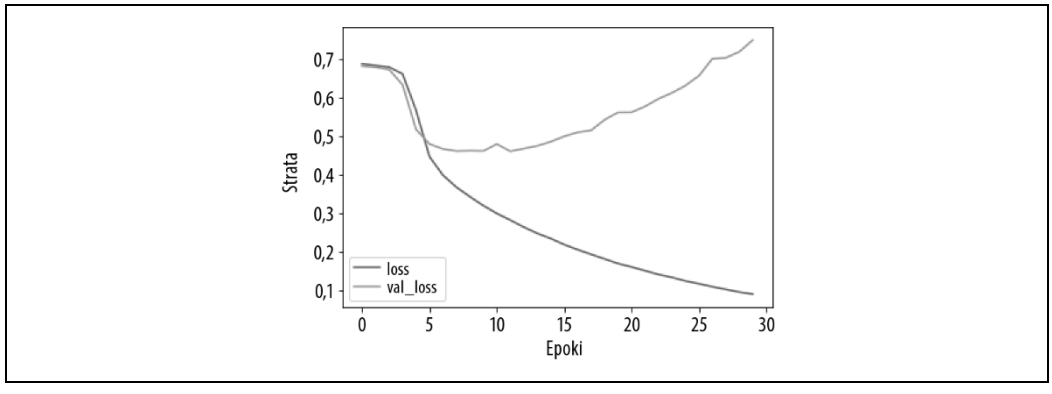
Dodanie warstwy spowoduje, że pojawi się około 100 000 nowych parametrów, których sieć będzie się musiała nauczyć. Oznacza to wzrost o mniej więcej 8%. Może to więc spowolnić działanie sieci, ale koszt ten będzie stosunkowo niski, jeśli przyniesie rozsądne korzyści.

Po trenowaniu trwającym 30 epok uzyskaliśmy wynik przedstawiony na rysunku 7.11. Chociaż dokładność zbioru walidacyjnego od pewnego momentu się nie zmienia, wykres straty (rysunek 7.12) mówi coś zupełnie innego.



Rysunek 7.11. Wykres zmian dokładności w modelu używającym kilku warstw LTSM

Jak widać na rysunku 7.12, mimo że uzyskaliśmy całkiem niezłą dokładność, strata zbioru walidacyjnego szybko wzrosła, co jest wyraźnym znakiem nadmiernego dopasowania.



Rysunek 7.12. Wykres zmian straty w modelu używającym kilku warstw LTSM

Rosnąca dokładność i zmniejszanie się straty dla zbioru treningowego, a także względnie stała dokładność i drastyczny wzrost straty dla zbioru walidacyjnego oznaczają nadmierne dopasowanie. Jest to wynik zbytniego wyspecjalizowania się modelu w danych ze zbioru treningowego. Podobną sytuację mieliśmy już w przykładach z rozdziału 6. Biorąc pod uwagę jedynie wskaźnik dokładności bez uwzględniania straty, można łatwo wyciągnąć fałszywy wniosek, że model działa prawidłowo.

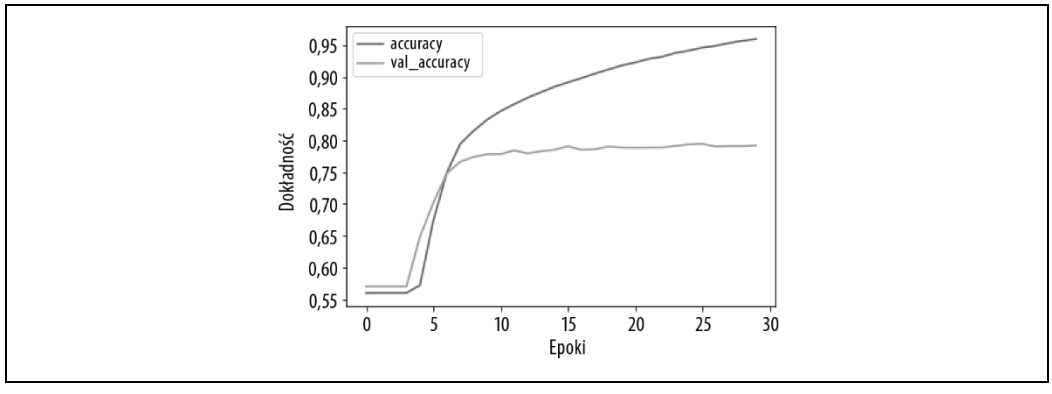
Optymalizacja modeli z wieloma warstwami LSTM

Z rozdziału 6.wiesz, że bardzo skuteczną metodą ograniczania nadmiernego dopasowania jest zmniejszanie współczynnika uczenia. Warto sprawdzić, czy takie rozwiązanie będzie miało pozytywny wpływ również na działanie rekurencyjnej sieci neuronowej.

Za pomocą poniższego kodu zmniejszamy wartość współczynnika uczenia o 20% — z 0,00001 do 0,000008:

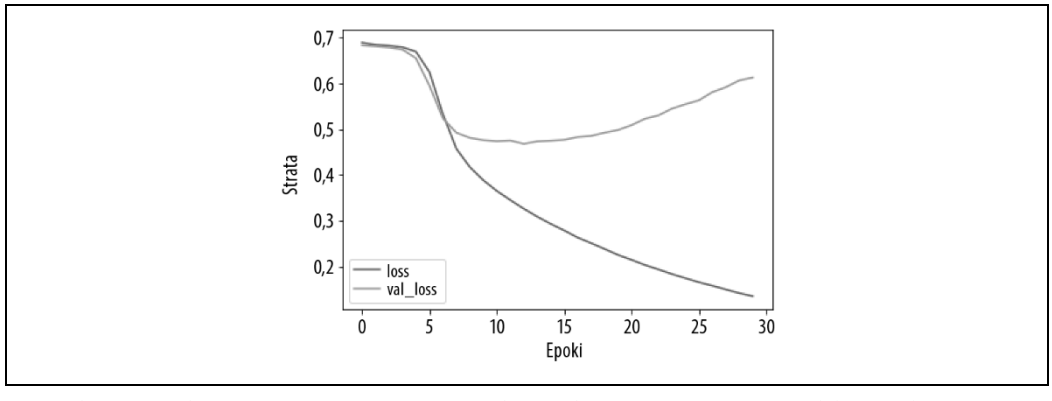
```
adam = tf.keras.optimizers.Adam(learning_rate=0.000008,
  beta_1=0.9, beta_2=0.999, amsgrad=False)

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

Wykres dokładności pokazałem na rysunku 7.13. Wygląda na to, że wyniki praktycznie nie uległy zmianie, chociaż krzywe (szczególnie w przypadku zbioru walidacyjnego) są trochę gładsze.



Rysunek 7.13. Wpływ zmniejszonej wartości współczynnika uczenia na dokładność modelu z wieloma warstwami LTSM

Pierwszy rzut oka na rysunek 7.14 również sugeruje podobny, minimalny wpływ zmniejszonej wartości współczynnika uczenia na stratę. Wykresowi należy jednak dokładniej się przyjrzeć. Pomimo mniej więcej podobnego kształtu krzywej tempo wzrostu straty jest wyraźnie niższe: po 30 epokach strata wynosi około 0,6, podczas gdy przy wyższym współczynniku uczenia była bliska 0,8. Z pewnością warto więc poeksperymentować z modyfikacją hiperparametru współczynnika uczenia.



Rysunek 7.14. Wpływ zmniejszonej wartości współczynnika uczenia na stratę modelu z wieloma warstwami LTSM

Użycie dropoutu

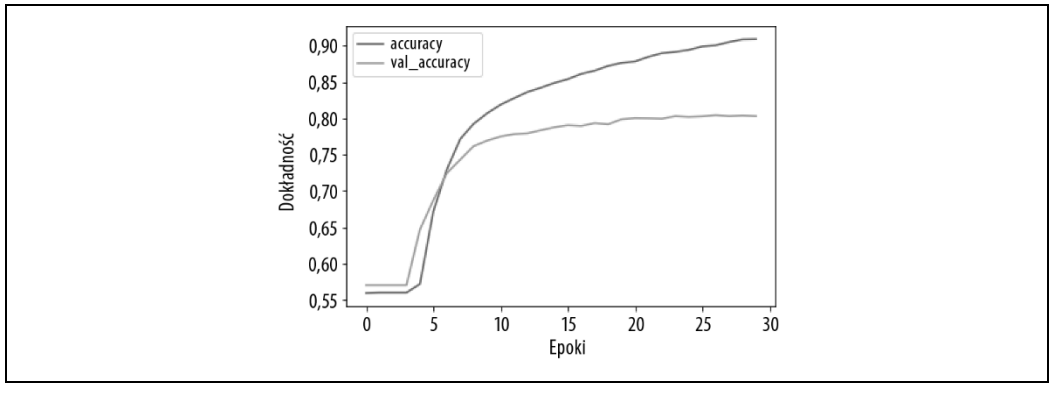
Oprócz zmiany parametru współczynnika uczenia warto również rozważyć użycie opcji dropoutu w warstwach LSTM. Działa ona dokładnie tak samo jak w przypadku warstw gęstych — losowo wybrane neurony są pomijane, aby zmniejszyć ich negatywny wpływ na uczenie.

Dropout można włączyć w warstwie LSTM za pomocą odpowiedniego parametru. Oto przykład:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim, return_sequences=True,
dropout=0.2)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim, dropout=0.2)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Pamiętaj, że włączenie dropoutu znacznie spowolni proces trenowania. U mnie (używałem środowiska Colab) czas trenowania w przypadku pojedynczej epoki wzrósł z około 10 sekund aż do 180 sekund.

Wykres dokładności przedstawiłem na rysunku 7.15.



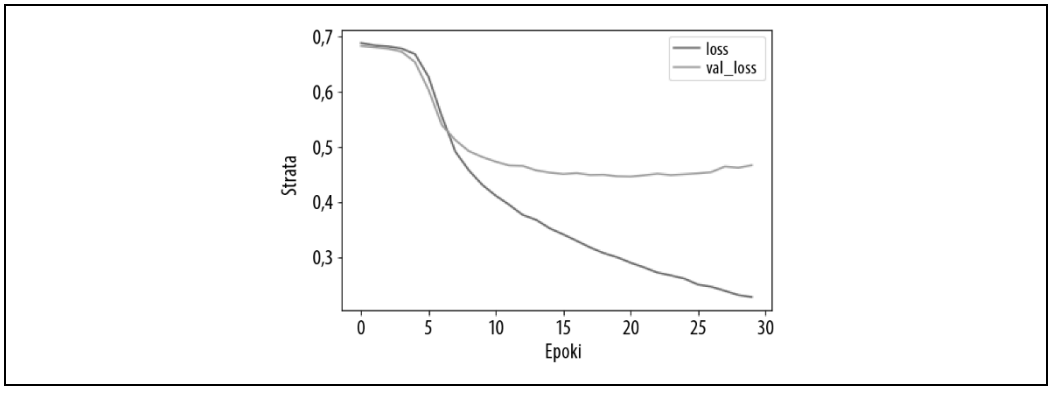
Rysunek 7.15. Wykres zmian dokładności w modelu LTSM wykorzystującym dropout

Jak widać na rysunku 7.15, zastosowanie dropoutu nie ma dużego wpływu na dokładność sieci, co jest jednak dobrym wynikiem! Zawsze przecież istnieje obawa, że zmniejszenie liczby aktywnych neuronów pogorszy działanie modelu. Widzimy jednak, że tak się nie stało.

Włączenie dropoutu ma jednak pozytywny wpływ na wartość straty, co przedstawiłem na rysunku 7.16.

Chociaż krzywe wyraźnie się rozdzielają, znajdują się bliżej siebie niż poprzednio. Strata dla zbioru walidacyjnego stabilizuje się szybko w okolicach wartości 0,5. To znacznie lepiej niż w poprzednim modelu, w którym wynosiła 0,8. Jak pokazuje ten przykład, dropout to kolejne przydatne rozwiązanie, którego można użyć do poprawy wydajności sieci RNN opartych na warstwach LSTM.

Aby uniknąć nadmiernego dopasowania, warto pamiętać o powyżej przedstawionych metodach, a także o procedurach związanych z wstępnym przetwarzaniem danych, które omówiłem w rozdziale 6. Jest jednak jedno rozwiązanie, którego jeszcze nie przetestowaliśmy. Chodzi o określony rodzaj uczenia transferowego, w którym zamiast przeprowadzać żmudny proces trenowania wykorzystuje się wstępnie wytrenowane osadzenia dla słów.



Rysunek 7.16. Wykres zmian straty w modelu LTSM wykorzystującym dropout

Użycie wstępnie wytrenowanych osadzeń w rekurencyjnych sieciach neuronowych

Do tej pory analizowaliśmy przykłady, w których zbiór treningowy zawierał wszystkie dostępne słowa. Za pomocą tego zbioru trenowaliśmy osadzenia, które przed przekazaniem ich do sieci gęstej musiały zostać zagregowane. Z tego rozdziału wiesz już, jak poprawić wyniki działania sieci RNN. Mogliśmy jednak używać jedynie słów oraz etykiet zawartych w zbiorze danych.

Wróćmy do rozdziału 4., w którym przeanalizowaliśmy uczenie transferowe. Co by się stało, gdybyśmy zamiast samodzielnie realizować proces uczenia użyli wcześniej wytrenowanych osadzeń, dla których została już wykonana ciężka praca polegająca na zamianie słów na sprawdzone wektory? Jednym z przykładów takiego rozwiązania jest model GloVe (Global Vectors for Word Representation) (*https://oreil.ly/4ENdQ*), opracowany przez Jeffreya Penningtona, Richarda Sochera i Christophera Manninga z Uniwersytetu Stanforda.

Na podstawie wstępnie wytrenowanych wektorów słów naukowcy stworzyli następujące zbiory danych:

- słownik zawierający 6 miliardów tokenów i 400 tysięcy słów, dostępny w 50, 100, 200 i 300 wymiarach, zawierający treści z Wikipedii i archiwum Gigaword;
- słownik zawierający 42 miliardy tokenów i 1,9 miliona słów, dostępny w 300 wymiarach, zawierający treści uzyskane po przeszukaniu internetu;
- słownik zawierający 840 miliardów tokenów i 2,2 miliona słów, dostępny w 300 wymiarach, zawierający treści uzyskane po przeszukaniu internetu;
- słownik zawierający 27 miliardów tokenów i 1,2 miliona słów, dostępny w 25, 50, 100 i 200 wymiarach, zawierający treści otrzymane po przeszukaniu Twittera z 2 miliardami tweetów.

Ponieważ wektory zostały już wstępnie wytrenowane, zamiast przeprowadzać uczenie od zera można ich w prosty sposób ponownie użyć w kodzie korzystającym z biblioteki TensorFlow. Najpierw należy pobrać bazę GloVe. Zdecydowałem się użyć danych Twittera z 27 miliardami tokenów i słownikiem zawierającym 1,2 miliona słów. Pobierane jest archiwum z 25, 50, 100 i 200 wymiarami.

Aby ułatwić przeprowadzenie testów, udostępniłem wersję 25-wymiarową. Za pomocą poniższego kodu możesz ją pobrać do środowiska Colab:

```
!wget --no-check-certificate \
    https://storage.googleapis.com/laurencemoroney-blog.appspot.com/glove.twitter.27B.25d.zip \
    -O /tmp/glove.zip
```

Jest to archiwum ZIP, więc możesz je rozpakować, aby uzyskać plik o nazwie *glove.twitter.27b.25d.txt*:

```
# Rozpakowywanie osadzeń GloVe
import os
import zipfile

local_zip = '/tmp/glove.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp/glove')
zip_ref.close()
```

Każdy wpis w pliku jest słowem, po którym występują wytrenowane współczynniki wymiarów. Aby wczytać plik, należy utworzyć słownik, w którym kluczami będą słowa, a wartościami osadzenia. Oto odpowiedni kod:

```
glove_embeddings = dict()
f = open('/tmp/glove/glove.twitter.27B.25d.txt')
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    glove_embeddings[word] = coefs
f.close()
```

Dzięki słownikowi możesz wyszukiwać współczynniki dla dowolnego słowa, używając go po prostu jako klucza. Na przykład aby się dowiedzieć, jakie osadzenia są związane ze słowem „frog", możesz wykonać poniższą instrukcję:

```
glove_embeddings['frog']
```

Następnie, tak jak poprzednio, użyj tokenizera, aby uzyskać indeks słów dla zbioru zdań. Jednak teraz możesz utworzyć nową strukturę, która nazywa się *macierzą osadzania*. Dzięki niej można będzie użyć osadzeń ze zbioru GloVe (pobranych ze słownika `glove_embeddings`) jako wartości. Wyświetlmy kilka początkowych słów ze zbioru danych:

```
{'<OOV>': 1, 'new': 2, … 'not': 5, 'just': 6, 'will': 7
```

Pierwszy wiersz w macierzy osadzania powinien więc zawierać współczynniki dla słowa `'<OOV>'`, następny dla słowa `'new'` itd.

Macierz możesz utworzyć za pomocą następującego kodu:

```
embedding_matrix = np.zeros((vocab_size, embedding_dim))
for word, index in tokenizer.word_index.items():
    if index>vocab_size - 1:
        break
    else:
       embedding_vector = glove_embeddings.get(word)
       ifembedding_vectoris not None:
    embedding_matrix[index] = embedding_vector
```

Kod ten tworzy macierz przy użyciu dwóch parametrów równych rozmiarowi słownika i wymiarom osadzania. Następnie dla każdego elementu w indeksie słów tokenizera wyszukujesz współczynniki zawarte w słowniku `glove_embeddings` i dodajesz je do macierzy.

Możesz odpowiednio zmodyfikować warstwę osadzania, by używała wstępnie wytrenowanych osadzeń. W tym celu stosujesz parametr `weights`, a także określasz, by warstwa nie była trenowana, ustawiwszy parametr `trainable = False`:

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
                              weights=[embedding_matrix], trainable=False),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim,
                                  return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Podobnie jak w przypadku poprzednich modeli, możesz teraz przeprowadzić trenowanie. Warto jednak rozważyć zmianę rozmiaru słownika. Jedna z optymalizacji, jakie wykonaliśmy w poprzednim rozdziale, aby uniknąć nadmiernego dopasowania, miała na celu zapobiec przeciążeniu osadzeń słowami o niskiej częstości występowania. Użyliśmy do tego mniejszego słownika, zawierającego częściej wykorzystywane słowa. Ponieważ osadzanie słów zostało już wytrenowane za pomocą zbioru GloVe, można spróbować poszerzyć słownik — ale jak bardzo?

Przede wszystkim sprawdź, ile słów z Twojego zestawu zdań zawiera się w zbiorze GloVe. Składa się on co prawda z 1,2 miliona elementów, ale nie ma przecież gwarancji, że zawiera wszystkie Twoje słowa.

Poniżej przedstawię kod, który pozwoli przeprowadzić szybkie porównanie, dzięki któremu się dowiesz, jak duży powinien być Twój słownik.

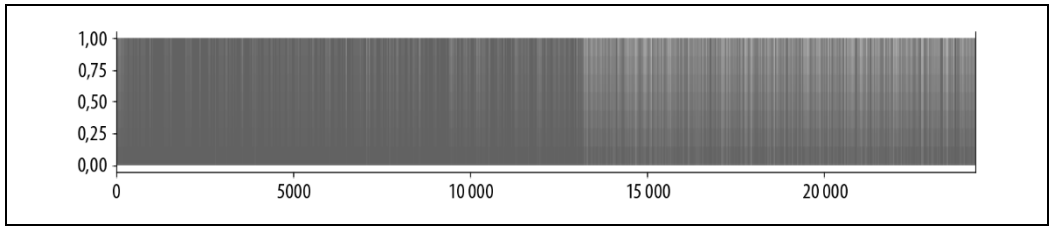
Najpierw uporządkujmy dane. Utwórz listę składającą się z elementów *X* i *Y*, gdzie *X* jest indeksem słowa, a $Y = 1$, jeśli słowo znajduje się w osadzeniach (w przeciwnym razie $Y = 0$). Dodatkowo możesz utworzyć zbiorczy zestaw, w którym będziesz zapisywać współczynnik wystąpień słów. Na przykład słowo `'<OOV>'` o indeksie 0 nie występuje w zbiorze GloVe, więc łączna wartość *Y* będzie równa 0. Kolejne słowo, `'new'`, jest zawarte w GloVe, więc łączna wartość *Y* wyniesie w tym momencie 0,5 (co oznacza, że połowa dotychczasowych słów znajduje się w GloVe). W taki sposób będziesz wyznaczać wartość aż do osiągnięcia końca zbioru:

```
xs=[]
ys=[]
cumulative_x=[]
cumulative_y=[]
total_y=0
for word, index in tokenizer.word_index.items():
  xs.append(index)
  cumulative_x.append(index)
  if glove_embeddings.get(word) is not None:
    total_y = total_y + 1
    ys.append(1)
  else:
    ys.append(0)
  cumulative_y.append(total_y / index)
```

Za pomocą tego kodu narysujesz wykres wartości *X* i *Y*:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(12,2))
ax.spines['top'].set_visible(False)

plt.margins(x=0, y=None, tight=True)
#plt.axis([13000, 14000, 0, 1])
plt.fill(ys)
```

W ten sposób uzyskasz wykres częstości występowania słów (rysunek 7.17).



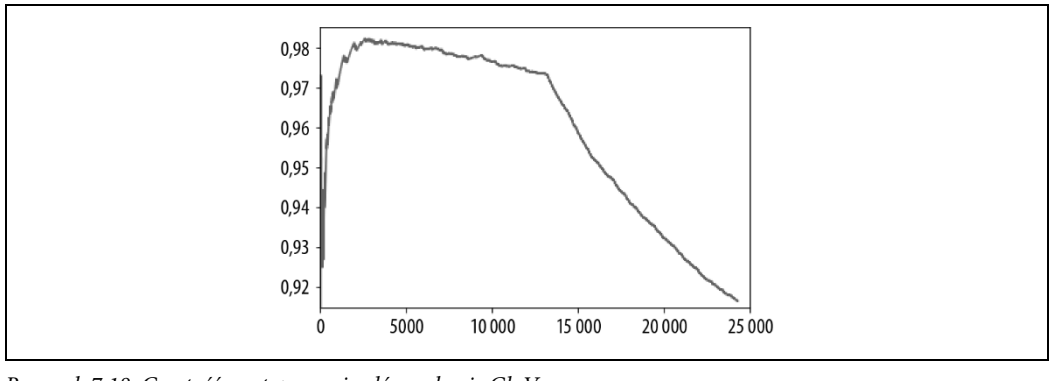
Rysunek 7.17. Wykres częstości występowania słów

Jak widać, gęstość wykresu zmienia się znacząco dla indeksów zawartych w przedziale od 10 000 do 15 000. Słowa, które *nie są zawarte* w osadzeniach zbioru GloVe, zaczynają częściej występować niż te, które się *w nich znajdują*. Ma to miejsce w pobliżu tokenu o numerze 13 000.

Możesz to lepiej zrozumieć, jeśli wykreślisz wykres wartości `cumulative_x` i `cumulative_y`. Oto odpowiedni kod:

```
import matplotlib.pyplot as plt
plt.plot(cumulative_x, cumulative_y)
plt.axis([0, 25000, .915, .985])
```

Wynik przedstawiłem na rysunku 7.18.



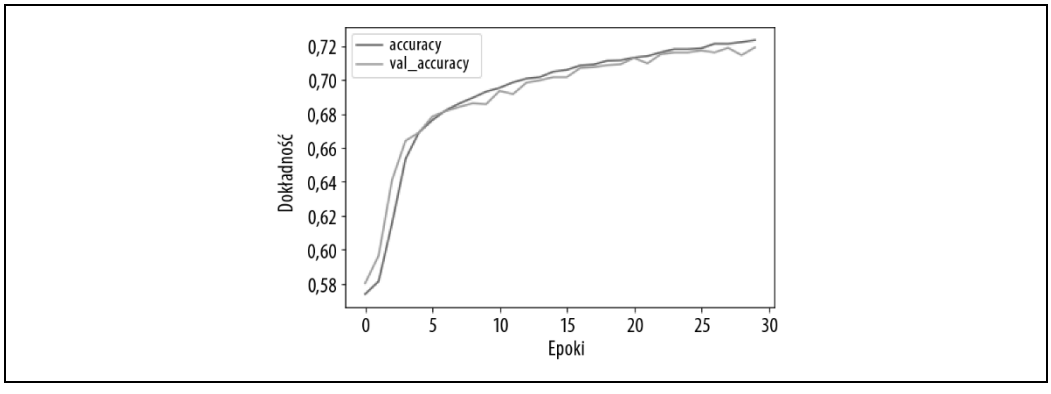
Rysunek 7.18. Częstość występowania słów w bazie GloVe

Możesz również odpowiednio zmodyfikować parametry wywołania `plt.axis`, aby znaleźć i przybliżyć punkt przegięcia, w którym słowa nieobecne w zbiorze GloVe zaczynają częściej występować niż te, które są w nim zawarte. Na podstawie uzyskanego wyniku możesz odpowiednio zmodyfikować wielkość słownika.

Skorzystawszy z powyższej metody, zdecydowałem się użyć słownika o rozmiarze 13 200 (zamiast 2000, który był poprzednio stosowany, aby uniknąć nadmiernego dopasowania) oraz takiej architektury modelu, w której liczba wymiarów `embedding_dim` wynosi 25 ze względu na wykorzystywany zbiór GloVe:

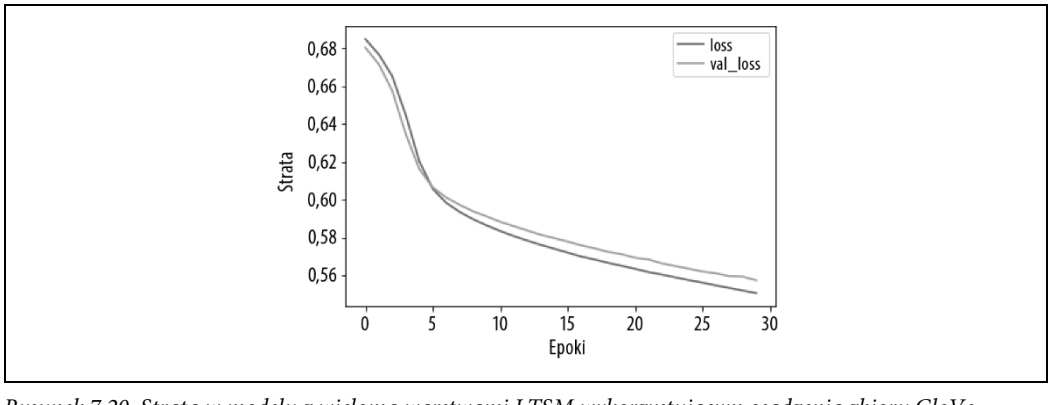
```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, weights=[embedding_matrix],
    ↪trainable=False),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim, return_sequences=True)),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(embedding_dim)),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
adam = tf.keras.optimizers.Adam(learning_rate=0.00001, beta_1=0.9, beta_2=0.999, amsgrad=False)
model.compile(loss='binary_crossentropy',optimizer=adam, metrics=['accuracy'])
```

Trenując sieć przez 30 epok, uzyskaliśmy doskonałe rezultaty. Wykres dokładności przedstawiłem na rysunku 7.19. Dokładność walidacji jest bardzo zbliżona do dokładności trenowania, co wskazuje, że nadmierne dopasowanie nie jest już problemem.



Rysunek 7.19. Dokładność w modelu z wieloma warstwami LTSM wykorzystującym osadzenia zbioru GloVe

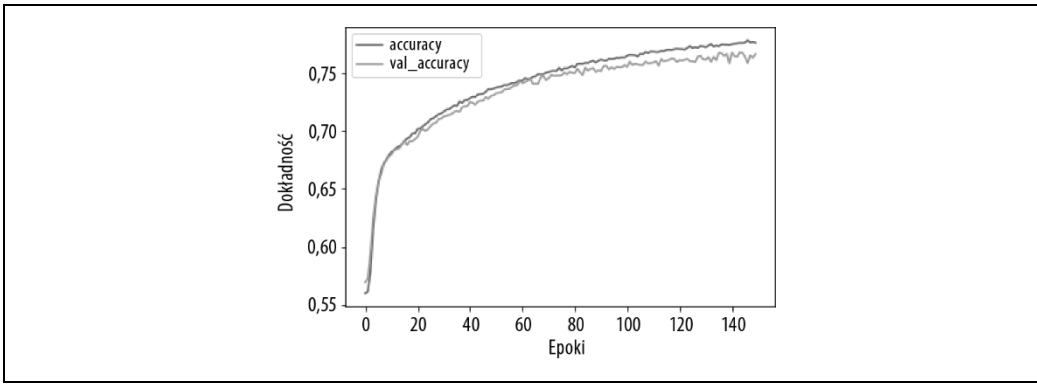
Potwierdza to również wykres strat, który pokazałem na rysunku 7.20. Nie ma już praktycznie różnicy między wartościami straty dla zbiorów walidacyjnego i treningowego. Chociaż dokładność wynosi tylko około 73%, możemy być jej pewni.



Rysunek 7.20. Strata w modelu z wieloma warstwami LTSM wykorzystującym osadzenia zbioru GloVe

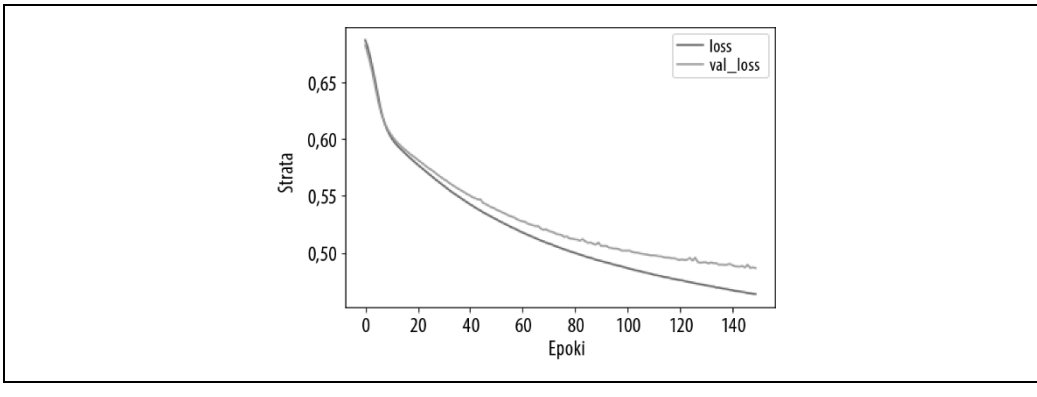
Po wytrenowaniu modelu przez dłuższy czas uzyskujemy bardzo podobne wyniki. Mimo że nadmierne dopasowanie pojawia się w okolicach 80. epoki, model jest nadal bardzo stabilny.

Wykres dokładności na rysunku 7.21 pokazuje, że model został dobrze wytrenowany.



Rysunek 7.21. Dokładność w modelu z wieloma warstwami LTSM wykorzystującym osadzenia zbioru GloVe i trenowanym przez 150 epok

Na wykresie straty z rysunku 7.22 widać, że różnice zaczynają narastać w przybliżeniu od 80. epoki, jednak model jest nadal dobrze dopasowany.



Rysunek 7.22. Strata w modelu z wieloma warstwami LTSM wykorzystującym osadzenia zbioru GloVe i trenowanym przez 150 epok

Wygląda na to, że ten model można wcześniej zatrzymać, czyli po prostu trenować go przez 75–80 epok, aby uzyskać optymalne wyniki.

Przetestowałem go z tekstami pochodzącymi z witryny The Onion, będącej źródłem sarkastycznych nagłówków dla zbioru danych Sarcasm, a także z innymi zdaniami:

```
test_sentences = ["It Was, For, Uh, Medical Reasons, Says Doctor To Boris Johnson, Explaining
↪Why They Had To Give Him Haircut",
"It's a beautiful sunny day",
"I lived in Ireland, so in high school they made me learn to speak and write in Gaelic",
"Census Foot Soldiers Swarm Neighborhoods, Kick Down Doors To TallyHousehold Sizes"]
```

Otrzymane wyniki są następujące (pamiętaj, że wartości bliskie 50% (0,5) oznaczają treści neutralne, bliskie 0 niesarkastyczne, a bliskie 1 sarkastyczne):

```
[[0.8170955 ]
 [0.08711044]
 [0.61809343]
 [0.8015281 ]]
```

Pierwsze i czwarte zdanie, oba zaczerpnięte z witryny The Onion, zostały ocenione jako sarkastyczne (z prawdopodobieństwem wyższym niż 80%). Wypowiedź o pogodzie była zdecydowanie niesarkastyczna (9%), a informację o uczęszczaniu do liceum w Irlandii model uznał za potencjalnie sarkastyczną, ale nie był tego pewny (62%).

Podsumowanie

W tym rozdziale zaprezentowałem rekurencyjne sieci neuronowe, które wykorzystują logikę zorientowaną na sekwencje, dzięki czemu oceniają zdania nie tylko na podstawie zawartych w nich słów, ale także kolejności, w jakiej się one pojawiają. Dowiedziałeś się, jak działa podstawowa sieć RNN, a także w jaki sposób wykorzystuje się w niej warstwy LSTM, aby uzyskać możliwość długotrwałego przechowywania kontekstu. Użyliśmy ich, żeby ulepszyć model wykrywania emocji, który wcześniej stworzyliśmy. Następnie przeanalizowaliśmy problemy związane z nadmiernym dopasowaniem sieci RNN i metody jego unikania, między innymi uczenie transferowe na podstawie wstępnie wytrenowanych osadzeń. Zdobytą wiedzę wykorzystasz w rozdziale 8., by przewidywać słowa. Dzięki temu zaprojektujesz model, który tworzy teksty w postaci wierszy.

Uczenie maszynowe: nie przestawaj zdobywać wiedzy!

Sztuczna inteligencja i uczenie maszynowe znajdują coraz więcej zastosowań w niemal wszystkich istotnych branżach. W technologiach sieci neuronowych tkwi olbrzymi potencjał. Za rozwojem uczenia maszynowego muszą nadążać architekci i programiści: aplikacja, w której wykorzystano technologie sztucznej inteligencji, musi pasować do określonego zastosowania. Poszczególne systemy różnią się od siebie, tak samo jak różne są rozwiązywane przez nie problemy. Sztuczna inteligencja ujawni swoje ogromne możliwości tylko, jeśli inżynierowie dostosują swoje aplikacje do rozwiązywania konkretnych problemów.

Ta książka jest praktycznym podręcznikiem opartym na sprawdzonej metodyce: nauce poprzez pisanie kodu w Pythonie. Aby w pełni z niego skorzystać, nie musisz znać wyższej matematyki. Dzięki praktycznym lekcjom szybko zaczniesz programowo tworzyć konkretne rozwiązania. Dowiesz się, jak można zaimplementować najważniejsze algorytmy uczenia maszynowego, korzystając ze znakomitej biblioteki TensorFlow. Nauczysz się także, w jaki sposób wdrażać modele uczenia maszynowego i tworzyć przydatne aplikacje, które będą działały w różnych środowiskach i na różnych platformach: przykładowo napiszesz aplikację w języku Kotlin w środowisku Android Studio czy też w języku Swift w środowisku Xcode.

W książce między innymi:

- podstawy uczenia maszynowego
- zastosowanie biblioteki TensorFlow do budowy praktycznych modeli
- tworzenie modeli sieci neuronowych
- implementacja widzenia komputerowego i rozpoznawania obrazów
- przetwarzanie języka naturalnego
- implementacja modeli dla urządzeń z systemami Android i iOS

Laurence Moroney pracuje w Google. Kieruje zespołem, który zajmuje się rozwiązaniami wykorzystującymi sztuczną inteligencję. Jest też trenerem: szkoli projektantów oprogramowania w zakresie technik budowy systemów uczenia maszynowego. Często udziela się na kanale TensorFlow w YouTube. Jest znanym na całym świecie prelegentem, a także autorem książek beletrystycznych — napisał kilka dobrze przyjętych powieści science fiction.

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7850-6

9 788328 378506

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 79,00 zł