9.1.1. Propagacja wsteczna przez czas

No więc jak to całe ustrojstwo się uczy? Dzięki propagacji wstecznej – tak jak każda inna sieć neuronowa. Cofnijmy się na chwilę i spójrzmy na problem, który próbujecie rozwiązać za pomocą tej nowej złożoności. Zwykły RNN jest podatny na problem znikającego gradientu, ponieważ pochodna w danym kroku czasowym jest uzależniona od samych wag, więc cofając się w czasie i łącząc różne delty, po kilku iteracjach wagi (i współczynnik szybkości uczenia się) mogą zmniejszyć gradient aż do 0. Aktualizacja wag na końcu propagacji wstecznej (co byłoby równoznaczne z początkiem ciągu) jest albo minimalna, albo efektywnie wynosi 0. Podobny problem występuje, gdy wagi są dość duże: gradient *eksploduje* i rośnie nieproporcjonalnie do sieci.

LSTM unika tego problemu dzięki samemu stanowi pamięci. Neurony w każdej bramce są aktualizowane za pomocą pochodnych funkcji, do których wpuszczono ich wyjście, a mianowicie tych, które aktualizują stan pamięci przy przejściu do przodu. Tak więc w każdym kroku czasowym, ponieważ normalna reguła łańcuchowa jest stosowana wstecz do propagacji w przód, aktualizacje neuronów zależą tylko od stanu pamięci w tym kroku czasowym i w kroku poprzednim. W ten sposób błąd całej funkcji jest utrzymywany „bliżej" neuronów dla każdego kroku czasowego. Nazywa się to *karuzelą błędu*.

W PRAKTYCE

A więc jak to działa w praktyce? Dokładnie tak, jak proste RNN znane z poprzedniego rozdziału. Wszystko, co zmieniliście, to wewnętrzny sposób działania czarnej skrzynki, która jest warstwą rekurencyjną w sieci. Możecie więc po prostu wymienić warstwę SimpleRNN Kerasa na warstwę LSTM Kerasa, a wszystkie inne elementy klasyfikatora pozostaną takie same.

Wykorzystacie ten sam zbiór danych, przygotowany w ten sam sposób: tokenizujecie teksty i wektoryzujecie je przy użyciu Word2vec. Następnie ponownie uzupełniacie/obcinacie ciągi do 400 tokenów, z których każdy używa funkcji, które zdefiniowaliście w poprzednich rozdziałach. Spójrzcie na następny listing.

Listing 9.2. Ładowanie i przygotowanie danych z IMDB

```
>>> import numpy as np

>>> dataset = pre_process_data('./aclimdb/train')
>>> vectorized_data = tokenize_and_vectorize(dataset)
>>> expected = collect_expected(dataset)
>>> split_point = int(len(vectorized_data) * .8)

>>> x_train = vectorized_data[:split_point]
>>> y_train = expected[:split_point]
>>> x_test = vectorized_data[split_point:]
>>> y_test = expected[split_point:]

>>> maxlen = 400
>>> batch_size = 32
>>> embedding_dims = 300
>>> epochs = 2
```

Zbieracie i przygotowujecie dane.

Dzielicie dane na dwa zbiory: treningowy i testowy.

Definiujecie hiperparametry.

Liczba próbek, którą pokazuje się sieci przed propagacją wsteczną błędu i aktualizacją wag.

Długość wektorów dla tokenów, które tworzycie, by przekazać je do sieci rekurencyjnej.

```
>>> x_train = pad_trunc(x_train, maxlen)
>>> x_test = pad_trunc(x_test, maxlen)
>>> x_train = np.reshape(x_train,
...      (len(x_train), maxlen, embedding_dims))
>>> y_train = np.array(y_train)
>>> x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
>>> y_test = np.array(y_test)
```

Dalszy etap przygotowywania danych polegający na nadaniu każdemu punktowi danych jednakowej długości.

Przekształcenie w strukturę danych numpy.

Następnie możecie zbudować model przy użyciu nowej warstwy LSTM, tak jak pokazano to na kolejnym listingu.

Listing 9.3. Budowanie sieci LSTM Kerasa

```
>>> from keras.models import Sequential
>>> from keras.layers import Dense, Dropout, Flatten, LSTM
>>> num_neurons = 50
>>> model = Sequential()
>>> model.add(LSTM(num_neurons, return_sequences=True,
...               input_shape=(maxlen, embedding_dims)))
>>> model.add(Dropout(.2))
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
>>> model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
>>> model.summary()
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (None, 400, 50)           70200
_________________________________________________________________
dropout_2 (Dropout)          (None, 400, 50)           0
_________________________________________________________________
flatten_2 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 20001
=================================================================
Total params: 90,201.0
Trainable params: 90,201.0
Non-trainable params: 0.0
```

Keras sprawia, że implementacja jest prosta.

Spłaszczacie wyjście z LSTM.

Warstwa składająca się z jednego neuronu, która będzie dawać na wyjściu liczby zmiennoprzecinkowe z zakresu między 0 a 1.

Wytrenujcie i zapiszcie model tak jak poprzednio, jak pokazano to na następnych dwóch listingach.

Listing 9.4. Uczenie modelu LSTM

```
>>> model.fit(x_train, y_train,
...           batch_size=batch_size,
...           epochs=epochs,
...           validation_data=(x_test, y_test))
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [==============================] - 548s - loss: 0.4772 -
acc: 0.7736 - val_loss: 0.3694 - val_acc: 0.8412
Epoch 2/2
```

Trenujecie model.

```
20000/20000 [==============================] - 583s - loss: 0.3477 -
acc: 0.8532 - val_loss: 0.3451 - val_acc: 0.8516
<keras.callbacks.History at 0x145595fd0>
```

Listing 9.5. Zachowaj to na później

```
>>> model_structure = model.to_json()
>>> with open("lstm_model1.json", "w") as json_file:
...     json_file.write(model_structure)

>>> model.save_weights("lstm_weights1.h5")
```

Zapiszcie jego strukturę, byście nie musieli robić tej części ponownie.

Jest to ogromny skok, jeśli chodzi o dokładność walidacji w porównaniu z prostym RNN zaimplementowanym w rozdziale 8 przy użyciu tego samego zestawu danych. Możecie zacząć dostrzegać, jak duży zysk jesteście w stanie osiągnąć, wyposażając model w pamięć, gdy związki tokenów są tak ważne. Piękno tego algorytmu polega na tym, że uczy się on *związków* między tokenami, które widzi. Sieć jest teraz w stanie wymodelować te relacje, szczególnie w sposób, który motywowany jest wybranym przez was zadaniem i funkcją kosztu.

Jak blisko jesteście w tym przypadku poprawnej identyfikacji pozytywnego lub negatywnego wydźwięku? Jasne, jest to wąski podobszar o wiele poważniejszego problemu w przetwarzaniu języka naturalnego. Jak na przykład można by modelować humor, sarkazm lub niepokój? Czy można je modelować razem? To zdecydowanie obszar, w którym toczą się intensywne badania. Jednak prowadzenie prac nad każdym z tych zagadnień osobno, choć wymaga dużej ilości ręcznie etykietowanych danych (a każdego dnia jest ich więcej), jest z pewnością realną ścieżką, a umieszczanie tego rodzaju dyskretnych klasyfikatorów w waszym potoku jest pełnoprawnym sposobem postępowania w gęstej przestrzeni problemów.

9.1.2. Próba ognia

Ta część jest zabawna. Mając wyszkolony model, możecie zacząć wypróbowywać różne przykładowe frazy i sprawdzać skuteczność modelu. Spróbujcie go oszukać. Użyjcie radosnych słów w negatywnym kontekście. Spróbujcie fraz długich, krótkich, sprzecznych. Zerknijcie na listingi 9.6 i 9.7.

Listing 9.6. Ponowne załadowanie waszego modelu LSSTM

```
>>> from keras.models import model_from_json
>>> with open("lstm_model1.json", "r") as json_file:
...     json_string = json_file.read()
>>> model = model_from_json(json_string)

>>> model.load_weights('lstm_weights1.h5')
```