Łukasz Jaworski 155 NCI

Labolatorium 6

Deep Learning – klasyfikacja opinii użytkowników portalu IMDB.com

W tym zadaniu zbuduję sieć neuronową, która po pobraniu informacji ze strony **IMDB.com,** czy dokładniej opinii pod filmem z tej witryny będzie w stanie określić, czy dana recenzja ma wydźwięk pozytywny czy negatywny. Będzie to możliwe dzięki powiązaniu tekstu wpisanego przez użytkownika ze skalą zadowolenia przypisaną do komentarza.

Do wykonania tego zadania użyję środowiska **R** oraz **R Studio** z wykorzystaniem odpowiednich bibliotek – **Keras** (sieci neuronowe) oraz **TensorFlow** (machine learninng)

Pierwszym krokiem było zainstalowanie wszelkich potrzebnych programów oraz bibliotek.

```
> library(devtools)
> install.packages("keras")
Installing package into 'C:/Users/Documents/R/win-library/3.5'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.5/keras_2.2.4.zip'
Content type 'application/zip' length 3863222 bytes (3.7 MB)
downloaded 3.7 MB

package 'keras' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\AppData\Local\Temp\RtmpIvQATq\downloaded_packages
> library(keras)
> install.packages("tensorflow")
Error in install.packages : Updating loaded packages

Restarting R session...

> install.packages("tensorflow")
Installing package into 'C:/Users/Documents/R/win-library/3.5'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/3.5/tensorflow_1.10.zip'
Content type 'application/zip' length 155622 bytes (151 KB)
downloaded 151 KB

package 'tensorflow' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\AppData\Local\Temp\Rtmp4O9iUy\downloaded_packages
> imdb <- dataset_imdb(num_words = 10000)
Error in dataset_imdb(num_words = 10000) :
  could not find function "dataset_imdb"
> library(tensorflow)
> imdb <- dataset_imdb(num_words = 10000)
Error in dataset_imdb(num_words = 10000) :
  could not find function "dataset_imdb"
> install_tensorflow()
```

Wczytywanie bibliotek pozwala nam uzyskać dostęp do listy zawartych pakietów, które możemy pobrać.

```
     yaml:                    0.1.7-hfabe2cd_1001          conda-forge
     zlib:                    1.2.11-h2fa13f4_1004         conda-forge


Installation complete.

>

Restarting R session...

> imdb <- dataset_imdb(num_words = 10000)
Error in dataset_imdb(num_words = 10000) :
  could not find function "dataset_imdb"
> imdb <- dataset_imdb()
Error in dataset_imdb() : could not find function "dataset_imdb"
> library(keras)
> imdb <- dataset_imdb(num_words = 10000)
>
```

Dzięki bibliotece Keras posiadamy dostęp do wbudowanej bazy 50 tyś. opinii o filmach z serwisu IMDB.com, podzielonych na dwa obszary – testowy oraz uczący. Obydwa zestawy posiadają taka samą ilość opinii pozytywnych, jak i negatywnych.

Funkcja dataset_imdb posiada wartość num_words = 10000, aby wybrać jedynie 10000 najczęstszych występujących słów.

```
> library(keras)
> imdb <- dataset_imdb(num_words = 10000)
> train_data <- imdb$train$x
> train_labels <- imdb$train$y
> test_data <- imdb$test$x
> test_labels <- imdb$test$y
>
```

Po poprawnym wczytaniu listy opinii należy je podzielić pod względem zadań. Stworzyłem więc cztery listy – dwie dotyczące danych (**train_data** i **test_data** – zawierają indeksy konkretnych opinii) oraz dwie dotyczące ich wyników (**train_labels** oraz **test_labels** – listy zer i jedynek, klasyfikujące wypowiedź)

```
> vectorize_sequences <- function(sequences, dimension = 10000) {
+ results <- matrix(0, nrow = length(sequences), ncol = dimension)
+ for(i in 1:length(sequences))
+ results[i,sequences[[i]]] <- 1
+ results
+ }
```

Kolejnym krokiem jest przygotowanie pozyskanych danych do ich implementacji w sieci neuronowej. W tym momencie posiadamy jedynie listę obiektów, które należy

przekształcić na wektory które będą rozumiane w sieci neuronowej. Powyższy kod tworzy macierz, która będzie określać czy dany rekord (czyli jedno z 10000 słów) na wydźwięk pozytywny czy negatywny (0 lub 1) - dzięki temu możemy utworzyć dwie listy, które będziemy mogli przekazać sieci.

```
> x_train <- vectorize_sequences(train_data)
> x_test <- vectorize_sequences(test_data)
> y_train <- as.numeric(train_labels)
> y_test <- as.numeric(test_labels)
>
```

Przed przekazaniem list sieci neuronowe należy jeszcze zamienić listy etykiet z wartości Integer na Numeric.

Dzięki tym zabiegom do programu dostarczamy jedynie wektory oraz skalary, co znacznie ułatwia nam ustawienie parametrów naszej sieci. Teraz możemy określić jakie właściwości będzie posiadać nasza sieć. Przede wszystkim trzeba się zastanowić z ilu warstw będzie się składać nasza sieć oraz ilości węzłów w jednej warstwie.

Jest to ważne, ponieważ rozmiar warstwy decyduje o tym ile czynników ma wpływ na podjęcie końcowej decyzji – jeśli będzie zbyt mała, to sieć nie będzie w stanie dostatecznie manipulować dostarczonymi danymi do wydania poprawnego wyniku. Jeśli będzie za duża – sieć stanie się dużo bardziej złożona, a sama funkcja może się nauczyć ustawiania konkretnych wartości, co może poprawić skuteczność nauki na danych treningowych, a niekoniecznie podczas testowania.

W mojej sieci wykorzystam trzy warstwy – dwie posiadające 16 węzłów z funkcją aktywacyjną ‘relu’ (**rectified linear unit**) oraz warstwę decyzyjną Sigmoid.

ReLu jest funkcją liniową przyjmującą wartości (0, x), która będzie nam mówić „jak bardzo prawdopodobne jest, że dana recenzja jest pozytywna”.

Funkcja Sigmoid jest funkcją nieliniową o wartościach między 0 a 1, dzięki czemu możemy ją wykorzystać jako skalę prawdopodobieństwa w ostatniej warstwie.

```
> y_test <- as.numeric(test_labels)
>
> library(keras)
> model <- keras_model_sequential() %>%
+ layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
+ layer_dense(units = 16, activation = "relu") %>%
+ layer_dense(units = 1, activation = "sigmoid")
>
```

Ostatnim zadaniem było określenie kryteriów jakości naszego modelu – określać ją będą trzy wartości – **loss** (loss function, funkcja straty – określa jak błędne są nasze przewidywania), **optimizer** (funkcja modyfikująca wagi węzłów, która stara się

otrzymać jak najlepszy wynik przy pomocy funkcji straty) oraz **metrics –** funkcja oceniająca pracę modelu.

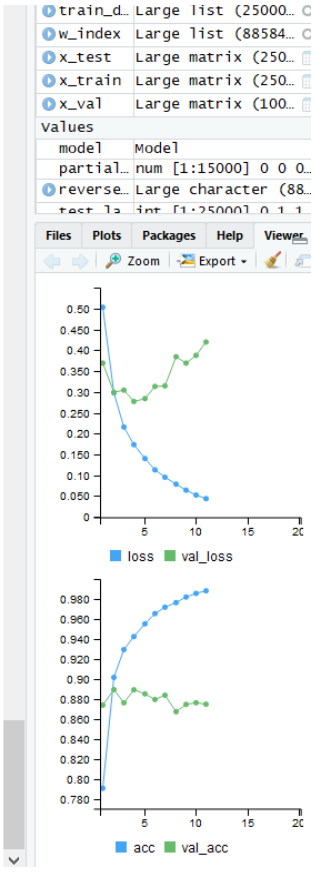
```
+ layer_dense(units = 16, activation = "relu") %>%
+ layer_dense(units = 1, activation = "sigmoid")
> model %>% compile(
+ optimizer = "rmsprop",
+ loss = "binary_crossentropy",
+ metrics = c("accuracy")
+ )
>
```

Zostały one ustawione według funkcji już zawartych w bibliotece **Keras**, które służą jako jedne z podstawowych funkcji modelowania sieci. Teraz pozostało nam zbudować odpowiedni zbiór walidacyjny, na podstawie którego będziemy widzieć postępy nauki naszej sieci.

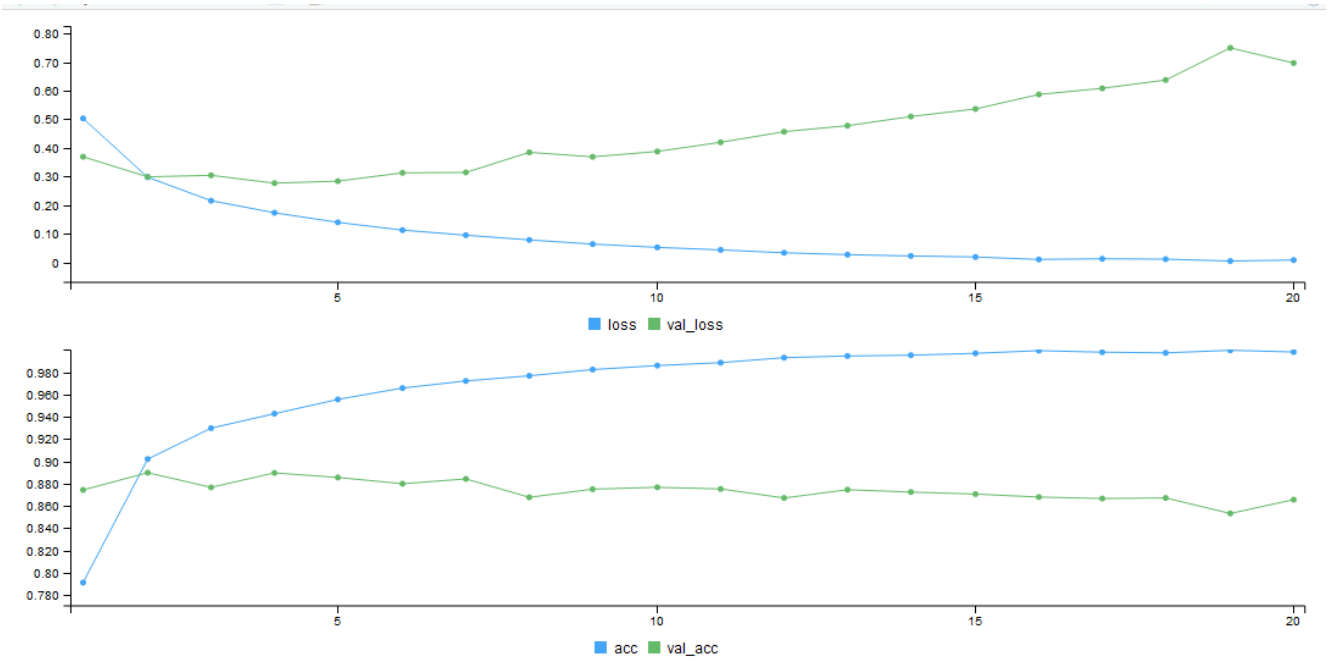
```
> val_indices <- 1:10000
> x_val <- x_train[val_indices,]
> partial_x_train <- x_train[-val_indices,]
> y_val <- y_train[val_indices]
> partial_y_train <- y_train[-val_indices]
>
```

Teraz pozostało nam ustawić funkcję **fitness**, która odpowiada za trenowanie naszej sieci. Ustawiłem ją na 20 iteracji po 512 obiektów, co jest optymalną ilością przy trenowaniu sieci tych rozmiarów.

```
> history <- model %>% fit(
+ partial_x_train,
+ partial_y_train,
+ epochs = 20,
+ batch_size = 512,
+ validation_data = list(x_val,y_val)
+ )
2019-03-07 20:33:59.774736: I T:\src\github\tensorflow\tensorflow\core\platfor
m\cpu_feature_guard.cc:141] Your CPU supports instructions that this TensorFlo
w binary was not compiled to use: AVX2
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 2s 136us/step - loss: 0.5032 -
acc: 0.7912 - val_loss: 0.3693 - val_acc: 0.8744
Epoch 2/20
15000/15000 [==============================] - 2s 112us/step - loss: 0.2978 -
acc: 0.9021 - val_loss: 0.2991 - val_acc: 0.8899
Epoch 3/20
15000/15000 [==============================] - 2s 111us/step - loss: 0.2157 -
acc: 0.9299 - val_loss: 0.3043 - val_acc: 0.8767
Epoch 4/20
15000/15000 [==============================] - 2s 111us/step - loss: 0.1735 -
acc: 0.9429 - val_loss: 0.2770 - val_acc: 0.8897
Epoch 5/20
15000/15000 [==============================] - 2s 119us/step - loss: 0.1401 -
acc: 0.9557 - val_loss: 0.2840 - val_acc: 0.8856
Epoch 6/20
15000/15000 [==============================] - 2s 120us/step - loss: 0.1129 -
acc: 0.9659 - val_loss: 0.3131 - val_acc: 0.8800
Epoch 7/20
15000/15000 [==============================] - 2s 119us/step - loss: 0.0952 -
acc: 0.9723 - val_loss: 0.3145 - val_acc: 0.8843
Epoch 8/20
15000/15000 [==============================] - 2s 116us/step - loss: 0.0787 -
acc: 0.9769 - val_loss: 0.3845 - val_acc: 0.8678
Epoch 9/20
15000/15000 [==============================] - 2s 135us/step - loss: 0.0641 -
acc: 0.9825 - val_loss: 0.3691 - val_acc: 0.8751
Epoch 10/20
15000/15000 [==============================] - 2s 137us/step - loss: 0.0525 -
acc: 0.9861 - val_loss: 0.3878 - val_acc: 0.8768
Epoch 11/20
15000/15000 [==============================] - 2s 107us/step - loss: 0.0438 -
acc: 0.9887 - val_loss: 0.4198 - val_acc: 0.8753
Epoch 12/20
15000/15000 [==============================] - 2s 107us/step - loss: 0.0337 -
acc: 0.9932 - val_loss: 0.4567 - val_acc: 0.8672
Epoch 13/20
 3584/15000 [======>.......................] - ETA: 0s - loss: 0.0223 - acc: 0
.9980
```



Po kilkunastu sekundach otrzymujemy skończony wykres z wynikami trenowania naszej sieci – na przestrzeni 20 epok osiągnęliśmy blisko 98% trafność decyzji.



Jest to bardzo dobry wynik jak na tego typu sieć – przeważnie osiąga się w nich dokładność 85-95%.

Oczywiście można modyfikować działanie sieci na wiele sposobów – poprzez zmianę ilości węzłów w warstwie, ilości warstw czy funkcje wykorzystywane przy ich aktywacji do ustawienia funkcji straty i optymalizacji czy metody trenującej sieć. W moim przypadku działałem na bardzo prostej sieci, jednak przy bardziej złożonych projektach uzyskanie tak wysokich wyników prawdopodobieństwa wymagałoby precyzyjnego zaplanowania budowy sieci oraz wybrania odpowiednich funkcji modyfikujących.