Lab 10

Hüseyin Anil Gündüz

2022-07-12

Exercise 1

In this exercise, we are going to revise the sentiment classifier for IMDB reviews we developed in a previous lab. Earlier, we encoded each review as a single “bag-of-words” vector which had one element for each word in our dictionary set to one if that word was found in the review, zero otherwise. This allowed us to use a simple fully-connected neural network but, on the flip side, we lost all information contained in the ordering and of the words and possible multiple repetitions. Recurrent neural networks, however, are able to process reviews directly. Let’s see how!

The first step is to load the data. For brevity, we only use the 10000 most common words and consider reviews shorter than 251 words, but if you can use a GPU then feel free to use all reviews and all words!

```
library(keras)
imdb <- dataset_imdb(num_words = 10000, maxlen=250)
```

```
## Loaded Tensorflow version 2.2.0
```

```
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb
```

Each review is a vector of numbers, each corresponding to a different word:

```
x_train[[5]]
```

```
##  [1]    1  778  128   74   12  630  163   15    4 1766 7982 1051    2   32   85
## [16]  156   45   40  148  139  121  664  665   10   10 1361  173    4  749    2
## [31]   16 3804    8    4  226   65   12   43  127   24    2   10   10
```

Even though RNNs can process sequences of arbitrary length, all sequences in the same batch must be of the same length, while sequences in different batches can have different length. In this case, however, we pad all sequences to the same length as this makes for much simpler code. Keras provides a function to do so for you called `pad_sequences` (read the documentation!).

```
x_train = (
  pad_sequences(x_train, 250)
)

x_test =(
  pad_sequences(x_test, 250)
)
```

Next, we define our sequential model. The first layer is an *embedding* layer that associates a vector of numbers to each word in the vocabulary. These numbers are updated during training just like all other weights in the network. Crucially, thanks to this embedding layer we do not have to one-hot-encode the reviews but we can use the word indices directly, making the process much more efficient.

Note the parameter `mask_zero`: this indicates that zeros in the input sequences are used for padding (verify that this is the case!). Internally, this is used by the RNN to ignore padding tokens, preventing them from contributing to the gradients (read more in the user guide, link!).

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10001, output_dim = 64, mask_zero = TRUE) %>%
  layer_lstm(units = 32) %>%
```

```r
  layer_dense(1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

summary(model)
```

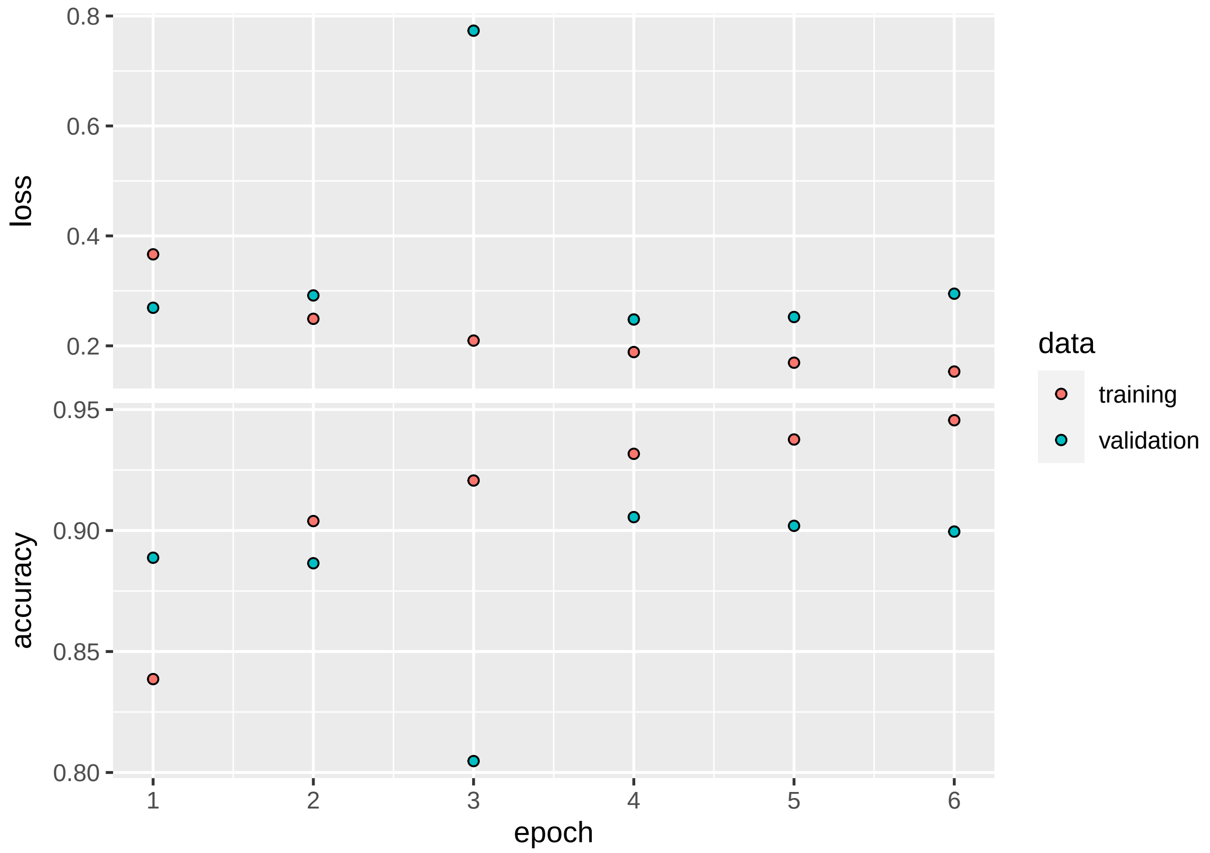
```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                        Param #
## ================================================================================
## embedding (Embedding)               (None, None, 64)                    640064
## ________________________________________________________________________________
## lstm (LSTM)                         (None, 32)                          12416
## ________________________________________________________________________________
## dense (Dense)                       (None, 1)                           33
## ================================================================================
## Total params: 652,513
## Trainable params: 652,513
## Non-trainable params: 0
## ________________________________________________________________________________
```

```r
hist = model %>% fit(
  x_train,
  y_train,
  batch_size = 32,
  epochs = 6,
  verbose = 0,
  validation_data = list(x_test, y_test)
)

plot(hist)
```

```
## Warning: replacing previous import 'lifecycle::last_warnings' by
## 'rlang::last_warnings' when loading 'tibble'

## Warning: replacing previous import 'lifecycle::last_warnings' by
## 'rlang::last_warnings' when loading 'pillar'
```



The model seems to be learning more easily than the simple baseline we created time ago, which had an accuracy of 85-88% on the test data. Let it train for longer and tune the architecture above to reach as high accuracy as possible! (note that evaluating on the same data that you used for early stopping is cheating).

Exercise 2

In this exercise, we are going to build a model that is able to sum two numbers, each given as a sequence of images of handwritten digits. The network will first use a convolutional encoder to transform each digit into a feature vector. These feature vectors will then be processed by a LSTM that will produce as output each digit of the sum.

In doing this, we will learn how to use Keras's functional API to create models with more than one inputs, as well as how to apply the same model independently to each item of a sequence.

Dataset

We are now going to create a synthetic dataset using images from MNIST.

```
library(keras)

mnist = dataset_mnist()

x_train = mnist$train$x / 255
y_train = to_categorical(mnist$train$y)
dim(x_train) <- c(nrow(x_train), 28, 28)
```

The first function we need is used to encode all digits of a number into a one-hot representation. From here on, we use `max_len` to indicate the maximum number of digits in a number. If a number has fewer digits we will pad it with zeros.

```
encode_number_to_onehot <- function(num, max_len) {
  num_str <- sprintf(paste("%0", max_len, "d", sep=""), num)
```

```
  encoded <- array(0, dim=c(max_len, 10))
  for(i in 1:max_len) {
    n <- as.integer(substring(num_str, i, i))
    encoded[i,n+1] <- 1
  }

  encoded
}

encode_number_to_onehot(195, 4)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    0    0    0    0    0    0    0    0     0
## [2,]    0    1    0    0    0    0    0    0    0     0
## [3,]    0    0    0    0    0    0    0    0    0     1
## [4,]    0    0    0    0    0    1    0    0    0     0
```

We now write a function to extract from MNIST the images of each digit in a given number.

```
encode_number_to_images <- function(num, max_len) {
  images <- array(0, dim=c(max_len, 28, 28, 1))
  num_str <- sprintf(paste("%0", max_len, "d", sep=""), num)
  for(i in 1:max_len) {
    n <- as.integer(substring(num_str, i, i))
    digit_idx <- (1:nrow(x_train))[y_train[,n+1] == 1]
    img_idx <- sample(digit_idx, 1)
    images[i,,,1] <- x_train[img_idx,,]
  }

  images
}
```

Let's now create a synthetic dataset with 25,000 random pairs of numbers and their sum.

```
make_dataset <- function(n_samples, max_len) {
  x1 <- array(0, dim=c(n_samples, max_len, 28, 28, 1))
  x2 <- array(0, dim=c(n_samples, max_len, 28, 28, 1))
  yy <- array(0, dim=c(n_samples, max_len, 10))

  for(i in 1:n_samples) {
    # ensure the sum always has at most max_len digits
    n1 <- sample.int(10**max_len / 2 - 1, 1)
    n2 <- sample.int(10**max_len / 2 - 1, 1)

    x1[i,,,,] <- encode_number_to_images(n1, max_len)
    x2[i,,,,] <- encode_number_to_images(n2, max_len)

    yy[i,,] <- encode_number_to_onehot(n1 + n2, max_len)
  }

  list(x1=x1, x2=x2, y=yy)
}

max_len = 3
train_set <- make_dataset(25000, max_len)
```

The model

Let's now see how to create the model in Keras.

This network will have two inputs, one for each number. The numbers have three digits, each of which is an image of size 28 x 28 x 1. To use the functional API, we need to manually create the input layers and specify the proper input shape (as we always did):

```
in_shape = c(
  max_len, 28, 28, 1
)

first_number_input <- layer_input(shape = in_shape)
second_number_input <- layer_input(shape = in_shape)
```

The network will use the same convolutional encoder for all digits in both numbers. Let us first define this encoder as its own model, a normal CNN:

```
digit_encoder <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3), activation = "relu") %>%
  layer_global_average_pooling_2d()
```

This CNN will transform each digit from a tensor of shape (28, 28, 1) to a vector of size, for example, 64. In order to apply this encoder to all digits of a number, we need to use a special layer called `time_distributed` (read its documentation!). This is how it is used:

```
first_number_input %>% time_distributed(digit_encoder)
```

The input of this sequence of transformations has shape (3, 28, 28, 1) while its output has shape (3, 64). This is because each (28, 28, 1) slice is processed by the CNN into a vector with 64 elements.

After we apply the CNN to both numbers, we need to "merge" the two sequence of vectors. There are several options here, here we choose to concatenate the two vectors in each time-step to produce a single vector of size 128:

```
encoded_numbers <- layer_concatenate(list(
  first_number_input %>% time_distributed(digit_encoder),
  second_number_input %>% time_distributed(digit_encoder)
))
```

This will result in a tensor of shape (3, 128). Let's feed this into a bidirectional LSTM, followed by a dense layer with to perform the final classification for each digit of the result. In order to do so, you may find the `bidirectional` layer useful (documentation here). Also be mindful of returning all hidden states from the LSTM, not only the last one. Refer again to the documentation.

```
model_output <- encoded_numbers %>%
  layer_dropout(0.5) %>%
  bidirectional(layer_lstm(units = 64, return_sequences = TRUE)) %>%
  layer_dense(10, "softmax")
```

We now have all components of the model. Let's then create and compile it:

```
model <- keras_model(
  inputs = list(first_number_input, second_number_input),
  outputs = model_output
)

model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adam",
  metrics=c("accuracy")
)
```

```r
model
```

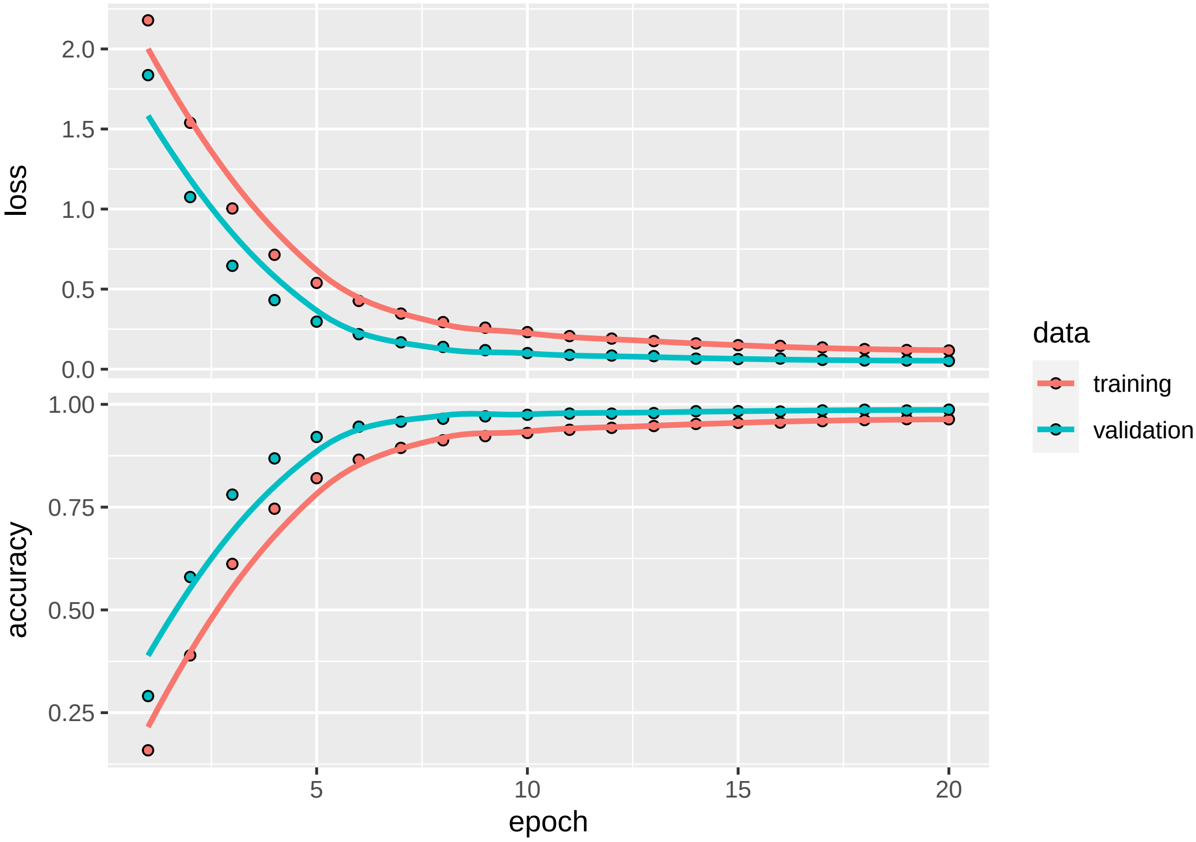
```
## Model: "model"
## ________________________________________________________________________________
## Layer (type)                 Output Shape      Param #  Connected to
## ================================================================================
## input_1 (InputLayer)         [(None, 3, 28, 28 0
## ________________________________________________________________________________
## input_2 (InputLayer)         [(None, 3, 28, 28 0
## ________________________________________________________________________________
## time_distributed_1 (TimeD (None, 3, 128)    92672    input_1[0][0]
## ________________________________________________________________________________
## time_distributed_2 (TimeD (None, 3, 128)    92672    input_2[0][0]
## ________________________________________________________________________________
## concatenate (Concatenate) (None, 3, 256)    0        time_distributed_1[0][0]
##                                                      time_distributed_2[0][0]
## ________________________________________________________________________________
## dropout (Dropout)            (None, 3, 256)    0        concatenate[0][0]
## ________________________________________________________________________________
## bidirectional (Bidirectio (None, 3, 128)    164352   dropout[0][0]
## ________________________________________________________________________________
## dense_1 (Dense)              (None, 3, 10)     1290     bidirectional[0][0]
## ================================================================================
## Total params: 258,314
## Trainable params: 258,314
## Non-trainable params: 0
## ________________________________________________________________________________
```

Training and validation

Finally, let's train this model on the synthetic dataset we created earlier. Since the model has two inputs, we must pass a `list` with two elements to `fit`:

```r
hist <- model %>% fit(
  list(train_set$x1, train_set$x2),
  train_set$y,
  validation_split = 0.2,
  epochs = 20,
  batch_size = 32,
  verbose = 0
)

plot(hist)
```



It is amazing what we achieved with such a small (for the standard of deep learning) model and dataset!