# Lab 10

Emilio Dorigatti

2021-01-22

## 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)
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]]
```

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 = (
  # TODO pad the training reviews to the same length we used above
)

x_test =(
  # TODO pad the testing reviews to the same length we used above
)
```

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) %>%
  layer_dense(1, activation = "sigmoid")

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

```
)

summary(model)
```

```
hist = model %>% fit(
  x_train,
  y_train,
  batch_size = 32,
  epochs = 8,
  verbose = 1,
  validation_data = list(x_test, y_test)
)
```

```
plot(hist)
```

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 implement an autoencoder and train it on the MNIST dataset. We will do this in Tensorflow, the library underlying Keras.

Tensorflow is a general numerical computation library that is able to automatically differentiating expressions by using a computational graph, just like we did some time ago. In practice, this means that any optimization problem solvable using analytic differentiation can be coded in Tensorflow. It also handles running the code on GPU and/or distributing it across a cluster of machines.

You should have installed Tensorflow together with Keras, if not refer to this guide here.

```
library(tensorflow)
library(keras)

mnist = dataset_mnist()
x_train = mnist$train$x / 255
input_size = 28 * 28
dim(x_train) <- c(nrow(x_train), input_size)
```

As an introduction, we will train PCA, i.e. an autoencoder with linear encoder $\mathbf{z} = \mathbf{E}\mathbf{x}$ and linear decoder: $\mathbf{x'} = \mathbf{D}\mathbf{z}$.

The first step is to create variables for $\mathbf{E}$ and $\mathbf{D}$.

```
latent_size = 64

weights_encoder = tf$Variable(
  matrix(
    rnorm(input_size * latent_size,  sd=sqrt(6 / (latent_size + input_size))),
    nrow = input_size
  )
)

weights_decoder = tf$Variable(
  matrix(
    rnorm(input_size * latent_size,  sd=sqrt(6 / (latent_size + input_size))),
    nrow = latent_size
  )
)
```

These two variables are managed by Tensorflow, and are called *tensors*:

```
weights_encoder
```

Tensorflow can only operate on data that is stored inside a `tf$Variable`. If you do not create one yourself, Tensorflow will do it automatically, but this will considerably slow down your program.

Now, we can train the autoencoder for one epoch.

```
optimizer = tf$optimizers$Adam(learning_rate = 0.001)

batch_size = 32
num_batches = floor(nrow(x_train) / batch_size)

# tensorflow is very sensitive about the data types of variables:
# all operands must be of the same type, and you have to take care
# of this. the default data type is tf$float64.
#
# try to comment the following line and get familiar with the error message
# when you try to run this chunk of code, because you will encounter it often.
bs = tf$cast(batch_size, tf$float64)

for(i in 1:num_batches) {
  batch = x_train[((i - 1) * batch_size):(i * batch_size - 1),]

  # gradient tapes keep track of the operations that are performed on tensors
  with(tf$GradientTape() %as% tape, {
    # all the computations need to be performed inside this block,
    # otherwise the tape will not be able to compute the gradients

    # here we perform the forward pass, two simple matrix multiplications
    latent = tf$matmul(batch, weights_encoder)
    reconstruction = tf$matmul(latent, weights_decoder)

    # then compute the mean squared error
    loss = tf$reduce_sum(tf$square(reconstruction - batch)) / bs
  })

  # now we use the gradient tape to compute the gradient of the loss
  # with respect to the two matrices
  variables = list(weights_encoder, weights_decoder)
  gradients = tape$gradient(loss, variables)

  # finally, we pass the gradients and the respective variables to the optimizer
  # which will update the variables by performing one step of gradient descent
  optimizer$apply_gradients(list(
    list(gradients[[1]], variables[[1]]),
    list(gradients[[2]], variables[[2]])
  ))

  # we print the loss every once in a while
  if(i %% 250 == 0) {
    cat("Batch", i, "- Loss:", loss$numpy(), "\n")
    flush.console()
  }
}
```

Let us now create a more advanced autoencoder. Tensorflow includes utilities to load and transform data, which is especially convenient when your dataset does not fit in memory and you have to load it from disk and transform it on the fly as training progresses.

```r
if(!require(tfdatasets)) {
  install.packages("tfdatasets")
  library(tfdatasets)
}

batch_size = 32
num_epochs = 15

train_dataset = mnist$train$x %>%
  # with this function, we create a "dataset" from matrices that we have in memory
  # in particular, `mnist$train$x` is a 3D tensor of shape (60000, 28, 28)
  # this function creates samples by moving along the first dimension
  # i.e. the i-th sample is `mnist$train$x[i,,]`
  tensor_slices_dataset() %>%

  # here we transform every element of the dataset
  dataset_map(function(x) {
    # we first flatten it to a vector
    x = k_flatten(x)

    # then convert it to smaller precision
    x = tf$cast(x, tf$float32)

    # finally we normalize from 0 to 1
    x / 255
  }) %>%

  # shuffle the data at the beginning of every epoch
  dataset_shuffle(nrow(mnist$train$x)) %>%

  # iterate over the dataset for a given number of epochs
  dataset_repeat(num_epochs) %>%

  # split the dataset into batches of the given size
  dataset_batch(batch_size)
```

Now, we can use a simple for-loop to iterate over this dataset, and we will receive batches with the right size and the transformations we requested.

Before doing this, let us define more elaborate encoders and decoders. Try to have several dense layers and experiment with different activation functions (e.g. `tf$nn$tanh`, `tf$nn$relu`, etc.).

```r
# utility functions to create the weights and biases for a dense layer
make_weights = function(input_size, output_size) {
  b = sqrt(6 / (output_size + input_size))
  weights = matrix(
    runif(input_size * output_size,  -b, b),
    nrow = input_size
  )

  tf$Variable(weights, dtype = tf$float32)
}

make_bias = function(size) {
  tf$Variable(tf$zeros(c(1, size), dtype=tf$float32))
}

# TODO create the parameters needed for the encoder
```

```
# TODO create the parameters needed for the encoder


# now, create functions to encode and decode the inputs / latent variables
# using the parameters defined above

encoder = function(inputs) {
  # TODO encode the inputs
}

decoder = function(latent) {
  # TODO decode the latent variables
}
```

When we trained PCA earlier, Tensorflow created the computational graph gradually as each operation was executed. After each operation completed, the results were returned to R and sent to the next operation. This is very inefficient, and it is possible to create the graph at once from the R code.

```
# this library automatically creates the computational graph from the R code
if(!require(tfautograph)) {
  install.packages("tfautograph")
  library(tfautograph)
}

optimizer = tf$optimizers$Adam(learning_rate = 0.01)

training_step = autograph(function(batch) {
  # TODO perform the forward pass, computing the loss.
  # remember to use the gradient tape!

  # TODO compute the gradients

  # TODO do one step of gradient descent

  loss
})


# simply wrap the top level function inside autograph to
# automatically generate the computational graph
train = autograph(function(num_batches) {
  i = 0
  for(batch in train_dataset) {
    loss = training_step(batch)

    if(i %% 250 == 0) {
      cat("Batch", i, "- Loss:", loss$numpy(), "\n")
      flush.console()
    }

    i = i + 1
    if(i >= num_batches) {
      break
    }
  }
})
```

```
train(1500)
```

Finally, let us inspect the reconstruction for an image:

```
img = tf$cast(matrix(x_train[5,], nrow = 1), tf$float32)
dec = decoder(encoder(img))$numpy()
rec = matrix(dec, nrow = 28)

rec[rec < 0] = 0
rec[rec > 1] = 1

grid::grid.raster(rec, interpolate = FALSE)
```

Which does not look too different from the input:

```
grid::grid.raster(matrix(x_train[5,], nrow = 28), interpolate = FALSE)
```