Resolución de un problema de clasificación con aprendizaje profundo utilizando un subconjunto del conjunto MNIST

Carlos García Gutiérrez (UO139393)

Introducción

La ejecución de esta práctica consta de tres partes:
- Cargar en memoria los datos a utilizar - Crear, entrenar y comparar los resultados de varias configuraciones de redes densas - Crear, entrenar y comparar los resultados de varias configuraciones de redes convolucionales

Carga de datos en memoria

```
library(keras)
```

Obtenemos el dataset MNIST

```
mnist <- dataset_mnist()
```

Definimos una semilla con los dígitos del DNI y generamos una secuencia aleatoria con un tamaño de la mitad del de la lista de imágenes/etiquetas

```
set.seed(53540153)
sample_array <- sample.int(nrow(mnist$train$x), size = floor(.10 * nrow(mnist$train$x)))
#PONER AL 50% ANTES DE ENTREGAR!!!
```

Obtenemos la mitad de las imágenes/etiquetas para entrenar, el conjunto de test es el completo

```
train_images <- mnist$train$x[sample_array,,]
train_labels <- mnist$train$y[sample_array]
test_images <- mnist$test$x
test_labels <- mnist$test$y

#Veamos la dimensión: es un tensor de 3 dimensiones
length(dim(train_images))
```

```
## [1] 3
```

```
dim(train_images)
```

```
## [1] 6000   28   28
```

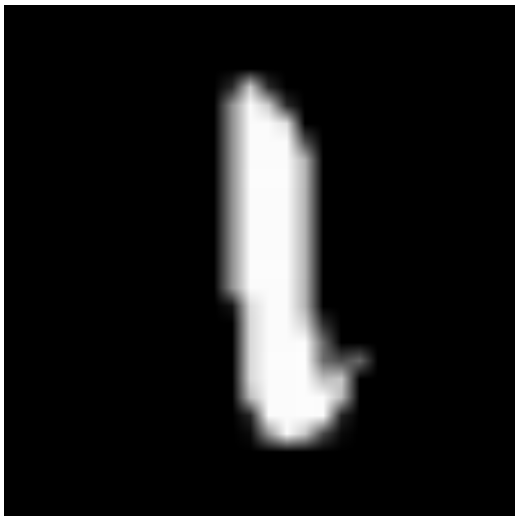
```
#El tipo de datos es

typeof(train_images)
```

```
## [1] "integer"
```

```
#Podemos acceder a una imagen del conjunto de entrenamiento

digit <- train_images[5,,]
plot(as.raster(digit, max = 255))
```



```
#`train_images` y `train_labels` forman el _conjunto de entrenamiento_ a partir de los cuales se realiz
#El modelo se probará con el _conjunto de test_, `test_images` y` test_labels`. Las imágenes se codific

#La función R `str ()` es una forma conveniente de echar un vistazo rápido a la estructura de una matri

 str(train_images)
```

```
##  int [1:6000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
```

```
 str(train_labels)
```

```
##  int [1:6000(1d)] 2 6 7 5 1 7 7 6 5 4 ...
```

```
#El flujo de trabajo será el siguiente: primero alimentaremos la red neuronal con los datos de entrenam

network <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")

summary(network)
```

```
## ________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ========================================================================
## dense (Dense)                       (None, 512)                     401920
## ________________________________________________________________________
## dense_1 (Dense)                     (None, 10)                      5130
## ========================================================================
## Total params: 407,050
## Trainable params: 407,050
## Non-trainable params: 0
## ________________________________________________________________________
```

```
#En las líneas anteriores se define la arquitectura de la red neuronal como una secuencia de dos capas

#Una vez que la arquitectura de la red está definida, es preciso definir el resto de ingredientes. En l

network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

# El operador %>% pasa el objeto que está a su izquierda como
# primer argumento de la función de su derecha

#Antes de entrenar, reconfiguramos los datos a la forma que la red espera y __escalamos para que todos


train_images <- array_reshape(train_images, c(nrow(train_images), 28 * 28))
train_images <- train_images / 255

#El conjunto de entrenamiento está almacenado en un 2-tensor (una matriz) de dimensión (60000, 784)
test_images <- array_reshape(test_images, c(nrow(test_images), 28 * 28))
test_images <- test_images / 255

train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)

#Finalmente ya podemos entrenar la red, que en Keras se hace con la función `fit`. En este caso la red
#en consecuencia. Después de estas 5 iteraciones, la red habrá realizado 2,345 gradientes.
#actualizaciones (469 por _epoch_), y la pérdida de la red será lo suficientemente baja como para que L

network %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)

#Se muestra el valor de la función de pérdida de la red y los datos de entrenamiento sobre los datos de
#El porcentaje de acierto en entrenamiento es de 98.9% en los datos de entrenamiento. Comprobemos que n

metrics <- network %>% evaluate(test_images, test_labels, verbose = 0)
metrics
```

```
## $loss
## [1] 0.212904
##
## $acc
```

```
## [1] 0.9369
```