Informe 2: Deep Learning con R

Capítulo 3 - Iniciándose en las redes neuronales

Carmen Lebrero Cia

Contents

3.4 Clasificando reseñas de películas: Ejemplo de clasificación binaria

La clasificación binaria es el problema d machine learning más aplicado. En este ejemplo aprenderemos a clasificar reseñas de películas como positivas o negativas basandose en el contenido del texto de las reseñas.

3.4.1 El dataset IMDB

Trabajaremos con el dataset IMBD: un conjunto de 50.000 reseñas de una base de datos de películas de internet. Éstas se encuentran divididas en 25.000 reseñas para entrenamiento y 25.000 reseñas de prueba, cada set consiste en 50% de reseñas negativas y 50% positivas.

¿Por qué se utilizan conjuntos separados de entrenamiento y prueba? Porque nunca se debe probar un modelo de machine learning con los mismos datos que has usado para entrenarlo! Solo porque un modelo funcione bien con los datos de entrenamiento no significa que funcione correctamente con datos que nunca ha tratado; y lo que tú quieres es que tu modelo consiga buenos resultados con datos nuevos (dado que ya saber las etiquetas de tus datos de entrenamiento- obviamente no necesitas tu modelo para predecir esto). Por ejemplo, es posible que tu modelo acabe memorizando un mapa entre tus muestras de entrenamiento y su diana, lo cual sería inútil para predecir la clasificación de datos que el modelo no ha visto antes. Profundizaremos más sobre essto en el siguiente capítulo.

Como el dataset MNIST, el dataset IMDB está empaquetado en Keras. Estos datos han sido ya preprocesados: las reseñas (secuencias de palabras) se han convertido en integers, donde cada integer se refiere a una palabra específica en un diccionario.

El siguiente código sirve para cargar el dataset (cuando lo ejecutas por primera vez, unos 80 MB de datos serán descargados).

```
library(keras)
imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
```

El argumento `num_words = 10000` define que nos quedaos solo con 10.000 palabras más utilizadas en los datos de entrenamiento. Las palabras raras se descartan. Esto nos permite trabajar con datos vectoriales de un tamaño manejable.

Las variables `train_data` y `testdata`son listas de reseñas; cada reseña es una lista de índices de palabras (codificando una secuencia de palabras). `train_labels` y `test_labels` son listas de 0s y 1s, donde 0 codifica para *negativo* y 1 para *positivo*:

```
str(train_data[[1]])
```

```
##  int [1:218] 1 14 22 16 43 530 973 1622 1385 65 ...
```

```
train_labels[[1]]
```

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

3.4.2 Preparando los datos

No puedes añadir listas de integers a un red neuronal. Tienes que convertir las listas en tensores. Hay dos maneras de conseguir esto:

- Haz que las listas tengan todas la misma longitud, conviertelas en un tensor integer de forma `(samples, word)`, y luego utiliza como primera capa de tu red una capa capaz de manejar estos tensores (la capa "embeding", que explicaremos más adelante).
- Codifica tus listas con el one-hot para convertirlas en vectores de 0s y 1s. Esto convertirá, por ejemplo, la secuencia [3,5] en un vector 10.000-dimensional que serán todo 0s excepto para los índices 3 y 5, que serán 1s. Puede utilizar como primera capa de tu red una capa densa, capaz de maejar datos de tipo vectores floating-point.

La última solución se puede hacer manualmente para máxima claridad:

```
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
}

x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

Las muestras tienen la siguiente estructura:

```
str(x_train[1,])
```

```
##  num [1:10000] 1 1 0 1 1 1 1 1 1 0 ...
```

También deberías cambiar tus etiquetas de integer a numeric:

```
y_train <- as.numeric(train_labels)
y_test <- as.numeric(test_labels)
```

Ahora los datos están listos para ser incorporados a una red neuronal.

3.4.3 Construyendo tu red

Los datos de entrada son vectores, y las etiquetas son escalares (0s y 1s): esta es la setup más sencilla que nos podemos encontrar. Un tipo de red que funciona bien con este problema es una pila de capas completamente conectadas (densas) con activaciones `relu`: `layer_dense(units = 16, activation = "relu")`.

El argumento de cada capa densa (16) es el número de unidades ocultas de la capa. Una *unidad oculta* es una dimensión en la representación del espacio de la capa. Puede que recuerdes del capítulo 2 que cada capa densa con activación `relu` implementa la siguiente cadena de operaciones de tensores:

`output = relu(dot(W, input) + b)`

Tener 16 unidades ocultas significa que el peso `W` de la matriz tendrá una forma `(input_dimesion, 16)`: el producto dot con W proyectará los datos de partida en un espacio de representación de 16 dimesiones (después añades el vector bias `b` y aplicas la operación `relu`). ——puedes entender intuitivamente la dimensionalidad de tu espacio de representación como "cuánta libertad estás permitiendo en la red cuando está aprendiendo representación interna". Tener más unidades ocultas (un espacio de representación con más dimensiones) permite a tu red aprender representaciones más complejas, pero hace que la red sea más costosa computacionalmente y puede llevar a patrones no deseados (patrones que mejorasen su actuacion con los datos de entrenamiento pero no con el conjunto de prueba).

Hay dos decisiones clave que tomar en cuanto a las capas:

- Cuántas capas utilizar
- Cuántas unidades ocultas elegir para cada capa

En el capítulo 4, aprenderemos principios formales para guiarnos en la toma de estas decisiones. Por ahora, simplemente tendrás que confiar en nosotros con la siguiente elección de arquitectura:

- Dos capas intermedias con 16 unidades ocultas cada una.

- Una tercera capa que devolverá la predicción escalar con el tipo de reseña.

Las capas intermedias utilizarán `relu` como función de activación, y la última capa utilizará una activación sigmoide dado que tiene que devolver una probabilidad (una puntuación entre 0 y 1, indicando cómo de probable es que la muestra tenga la etiqueta de "1": que la reseña sea positiva). `Relu`(rectified linear unit) es una función que convierte en 0 los vaores negativos. Mientras que una sigmoide "aplasta" valores arbitrarios en el intervalo [0,1], devolviendo algo que puede interpretarse como una probabilidad.

```
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")
```

Finalmente, necesitas escoger una función de pérdida y un optimizador. Dado que te estás enfrentando a un problema de clasificación binaria y el output de tu red es una probabilidad, es mejor utilizar `binary__crossentropy`. No es la única opción viable: podrías utilizar, por ejemplo, `mean_squared_error`. Pero crossentropy es normalmente la mejor opción cuando te enfrentas a modelos en los que el output es una probabilidad, *Crossentropy* es un cuantificador que mide la distancia entre las distribuciones de probabilidad o, en este caso, entre entre la distribución verdadera y tus predicciones.

Aquí tienes el paso en el que configurar el mdelo con el optimizador `rmsprop` y la función de pérdida `binary_crossentropy`. Ten en cuenta que también estás monitorizando la precisión durante el entrenamiento.

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

3.4.4 Validando la aproximación

Para monitorizar la precisión del modelo sobre los datos que no había visto antes, creas un conjunto de validación apartando 10.000 muestras de los datos originales de entrenamiento.

```
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]
```

Ahora entrenas el modelo para 20 epochs (20 iteraciones sobre todas las muestras en los tensored `x_train` e `y_train`), en mini batches (subconjuntos) de 512 muestras. Al mismo tiempo, monitorizarás la pérdida y la precisión en las 10000 muestras que has separado. Esto se hace con el argumento `validation_data`.

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

```
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

En la CPU esto tardará menos de 2 segundos por epoch- el entrenamiento termina en unos 20 segundos. Al acabar con cada epoch, hay una pequeña pausa mientras que el modelo computa su pérdida y precisión en las 10.000 muestras de datos de validación.

Ten en cuenta que `fit()` devuelve un objeto `history`. Vamos a ver qué contine:

```
str(history)
```

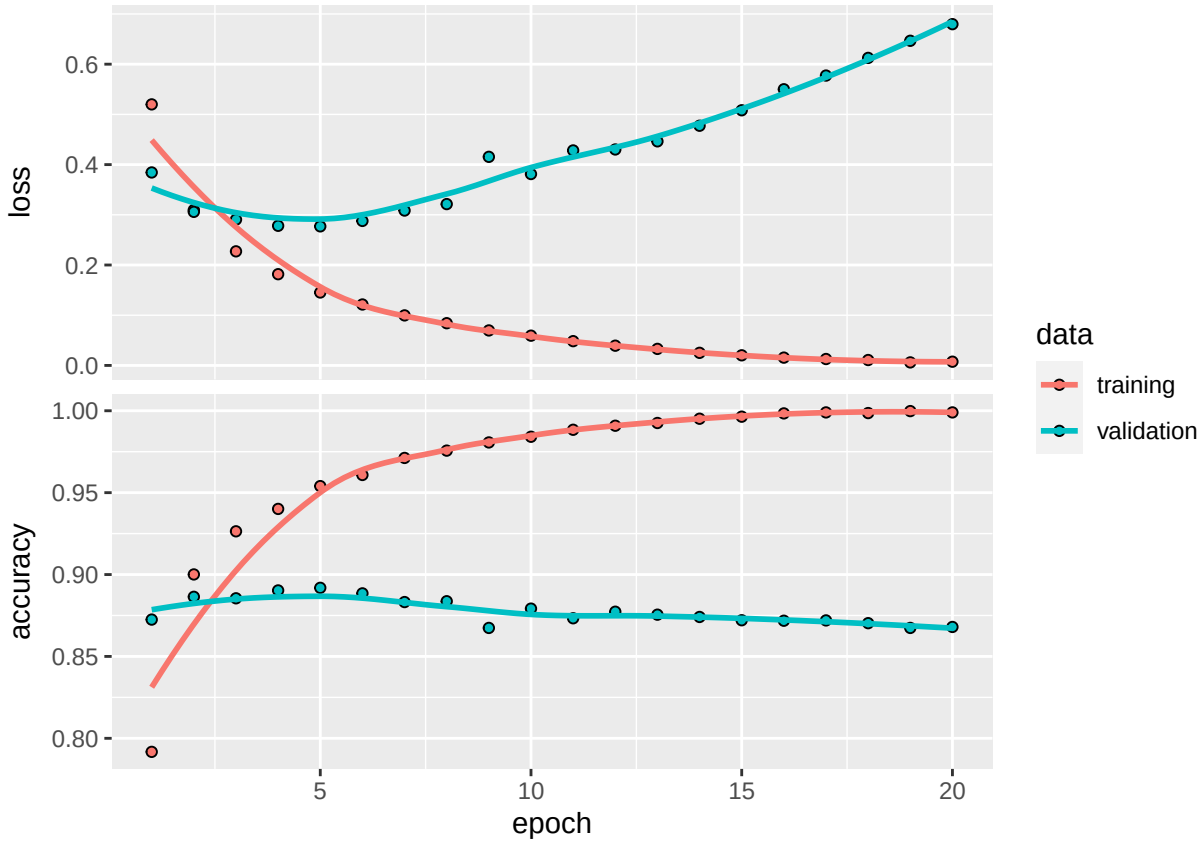
```
## List of 2
##  $ params :List of 3
##   ..$ verbose: int 1
##   ..$ epochs : int 20
##   ..$ steps  : int 30
##  $ metrics:List of 4
##   ..$ loss        : num [1:20] 0.52 0.31 0.227 0.182 0.145 ...
##   ..$ accuracy    : num [1:20] 0.792 0.9 0.926 0.94 0.954 ...
##   ..$ val_loss    : num [1:20] 0.384 0.306 0.291 0.278 0.277 ...
##   ..$ val_accuracy: num [1:20] 0.873 0.886 0.886 0.89 0.892 ...
##  - attr(*, "class")= chr "keras_training_history"
```

El objeto `history`incluye parámetros utilizados para ajustar el modelo (`history-$params`) y las métricas siendo monitorizadas (`history$metrics`).

Además este objeto tiene un método `plot()`que te permite visualizar el entrenamiento y la validación por epoch:

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Como se puede observar, la pérdida de entrenamiento decrece a cada epoch, y la precisión aumenta con cada epoch. Esto es lo que se esera cuando se ejecuta una optimización gradiente descendente - la cantidad que intentas minimizar debería ser menor a cada iteración. Sin embargo, vemos que no sucede lo mismo para la precisión y pérdida de la validación: parece que hay un pico en el cuarto epoch. Este es un ejemplo de lo que ya se ha venido avisando que podría suceder y hay que evitar, un modelo que actúa mejor sobre los datos de entrenamiento y tiene una mala actuación sobre los datos que no ha visto antes. En términos precisos, lo que estás observando se llama *overfitting*: tras el segundo epoch, estas sobreoptimizando los datos de entrenamiento, y al final aprende representaciones que son específicas del conjunto de entrenamiento y no generalizan para datos fuera de este conjunto.

Para prevenir este overfitting se podría pausar el entrenamiento tras el tercer epoch. En general, se puede usar muchas técnicas para mitigar el overfitting, éstas se mostrarán en el capítulo 4.

Vamos a entrenar una nueva red desde el inicio para 4 epochs y evaluaremos su actuación en el conjunto de prueba.

```
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")

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

```
model %>% fit(x_train, y_train, epochs = 4, batch_size = 512)
results <- model %>% evaluate(x_test, y_test)
results
```

```
##      loss  accuracy
## 0.2957187 0.8815600
```

3.5 Clasificando teletipos: Un ejemplo de clasificación multiclase

En la sección anterior vimos cómo clasificar vectores de entrada en dos clases mutualmente exclusivas utilizando una red neuronal densamente conectada. ¿Pero qué pasa cuando tienes más de dos clases?

En esta sección, construirás un red para clasificar teletipos Reuters en 26 tipos. Dado que tenemos muchas clases, este problema es un tipo de clasificación multiclase; y dado que cada punto de los datos debe ser clasificado solo en una categoría es, más específicamente, una clasificación multiclase de instancias unietiquetadas. Si cada punto perteneciese a múltiples categorías (en este caso tipos) nos estaríamos enfrentando a un problema de clasificación multiclase multietiquetado.

3.5.1 El dataset Reuters

```
library(keras)

reuters <- dataset_reuters(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% reuters
```

Al igual que con el dataset IMDB, el argumento `num_words = 10000` restringe los datos a las 10.000 palabras más frecuentes que podemos encontrar en los datos.

Tienes 8.982 ejemplos de entrenamiento y 2.246 ejemplos de prueba:

```
length(train_data)
```

```
## [1] 8982
```

```
length(test_data)
```

```
## [1] 2246
```

Cada ejemplo es una lista de integers (índice de palabras):

```
train_data[[1]]
```

```
##  [1]    1    2    2    8   43   10  447    5   25  207  270    5 3095  111   16
## [16]  369  186   90   67    7   89    5   19  102    6   19  124   15   90   67
## [31]   84   22  482   26    7   48    4   49    8  864   39  209  154    6  151
## [46]    6   83   11   15   22  155   11   15    7   48    9 4579 1005  504    6
## [61]  258    6  272   11   15   22  134   44   11   15   16    8  197 1245   90
## [76]   67   52   29  209   30   32  132    6  109   15   17   12
```

3.5.2 Preparando los datos

Puedes vectorizar los datos en el mismo código utilizado en el ejemplo anterior.

```
x_train <- vectorize_sequences(train_data)
x_test <- vectorize_sequences(test_data)
```

Para vectorizar las etiquetas, hay dos posibilidades: puedes convertirlas en un tensor integer o puedes codificarlas mediante one-hot encoding. En este caso, one-hot encoding de las etiquetas consiste en la codificación de cada etiqueta como un vector de ceros con un 1 en el sitio del índice de la etiqueta. Aquí tenemos un ejemplo:

```
to_one_hot <- function(labels, dimension = 46) {
  results <- matrix(0, nrow = length(labels), ncol = dimension)
  for (i in 1:length(labels))
    results[i, labels[[i]] + 1] <- 1
  results
}

one_hot_train_labels <- to_one_hot(train_labels)
one_hot_test_labels <- to_one_hot(test_labels)
```

Recuerda que hay una forma más sencilla de realizar esto en Keras:

```
one_hot_train_labels <- to_categorical(train_labels)
one_hot_test_labels <- to_categorical(test_labels)
```

3.5.3 Construyendo tu red

Este problema de clasificación es parecido al que hemos visto previamente de la clasificación de reseñas de películas: en los dos casos, intentamos clasificar textos cortos. Pero en este caso el número de clases en vez de ser 2 es 46. La dimensionalidad del output es mucho más grande.

En el modelo de apilamiento de capas densas que hemos estado utilizando hasta ahora, cada capa solo puede acceder a la información de la capa anterior. Si una de las capas pierde alguna información de relevancia para el problema de clasificación, esta información no puede recuperarse nunca en una capa posterior: cada capa es un potencial cuello de botella para la información. En el ejemplo de reseñas de películas se utilizaron capas intermedias de dimensión 16, pero un espacio de 16 dimensiones puede ser demasiado limitado para aprender a separar 46 clases diferentes: capas tan pequeñas actuaran como cuellos de botella para la información, perdiendo permanentemente información relevante.

Por esta razón utilizaremos capas más grandes. En este caso utilizaremos 64 unidades.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")
```

ón Hay otras dos cosass que deberías tener en cuenta sosbre esta arquitectura:

- Terminamos la red con una capa densa de tamaño 46. Esto significa que para cada muestra de entrada, la red va a devolver un vector de 46 dimensiones. Cada entrada en este vector (cada dimensión) codificará una clase de salida.

- La última capa usa una activación `softmax`. Esto significa que la red devolverá una *distribución de probabilidad* para las 46 clases distintas - para cada muestra de entrada, la red producirá un vector 46-dimensional, donde `output[[i]]` es la probabilidad de que la muestra pertenezca a la clase `i`. Las 46 puntuaciones sumarán 1 en total.

La mejor función de pérdida que se puede utilizar en este caso es `categorical_crossentropy`. Esto significa que la distancia entre dos distribuciones de probabilidad: entre la distribución de probabilidad de salida de la red y la verdadera distribución de las etiquetas.

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

3.5.4 Validando la aproximación

Vamos a separar 1.000 muestras de los datos de entrenamiento para usarlos como conjunto de validación.

```
val_indices <- 1:1000

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- one_hot_train_labels[val_indices,]
partial_y_train = one_hot_train_labels[-val_indices,]
```

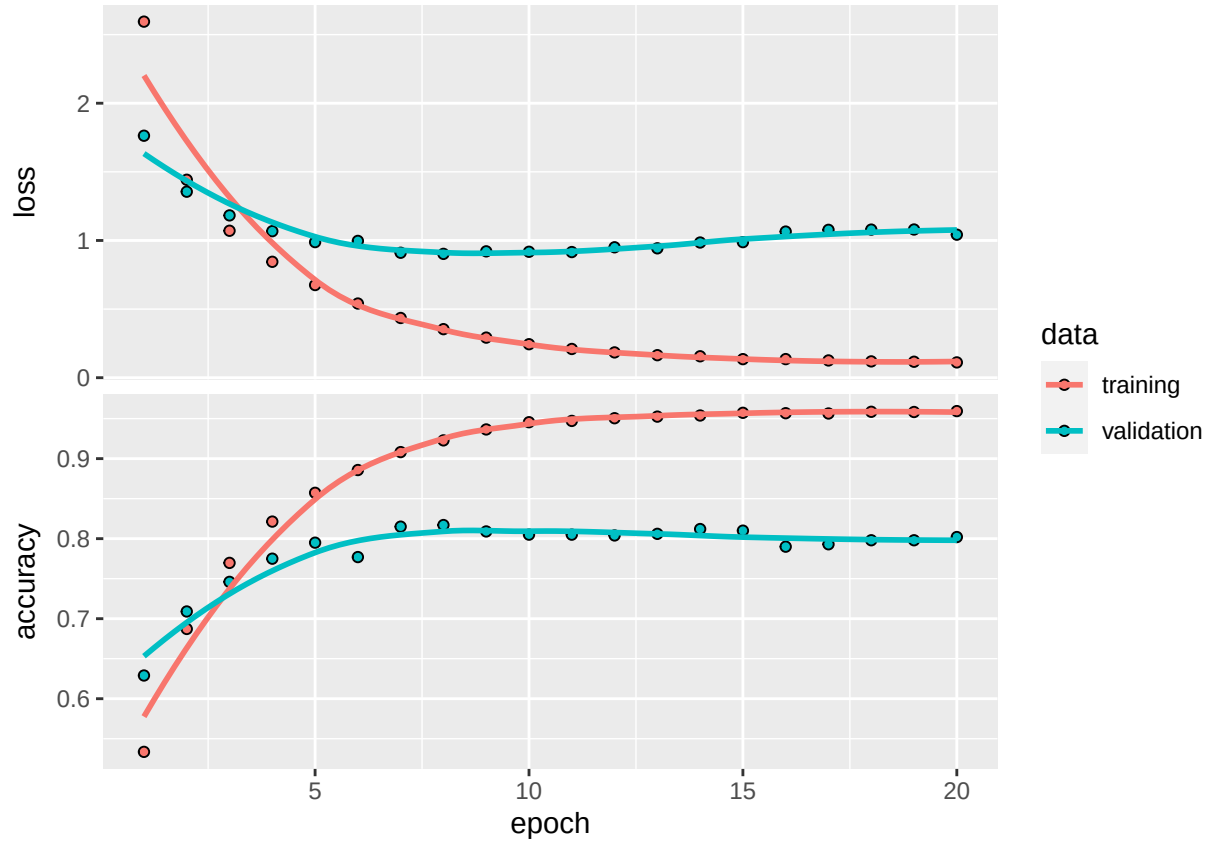
Ahora vamos a entrenar la red para 20 iteraciones.

```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```

Ahora podemos ver las cuervas de pérdida y de precisión

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



La red comienza a sobreajustarse tras la novena iteración. Vamos a volver a entrenar una red y evaluarla.

```
model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 46, activation = "softmax")

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

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 9,
  batch_size =512,
  validation_data = list(x_val, y_val)
)

results <- model %>% evaluate(x_test, one_hot_test_labels)

results
```

```
##      loss  accuracy
```

```
## 1.0174503 0.7800534
```

Esta aproximación alcanza una precisión de aproximadamente el 79%.

3.6 Prediciendo precios de casas: un ejemplo de regresión

Los dos ejemplos anteriores eran cosiderados problemas de clasificación, donde la meta era predecir una etiqueta discreta de un punto de los datos de entrada. Otro tipo de problema comun en machine learning es la *regresión*, que consiste en predecir un valor continuo de una etiqueta discreta: por ejemplo, predecir la temperatura de mañana dada información meteorológica; o prediciendo el tiempo que un proyecto de software tardará en completarse dadas unas especificaciones.

RECUERDA No confundas *regresión* con un algoritmo de *regresión logística*. La regresión logística no es un algoritmo de regresión sino uno de clasificación.

3.6.1 El dataset Boston Housing Price

Vamos a predecir el precio medio de las casas de un barrio de Boston de mediados de 1970s, dados unos puntos sobre el barrio en ese tiempo, como la tasa de crimen, la tasa de impuestos de la propiedad, y demás. El dataset que utilizaremos tiene una diferencia interesante con respecto a los otros dos ejemplos. Tiene relativamente menos puntos: solo 506, divididos en 404 muestras de entrenamiento y 102 de prueba. Y cada *característica* de los datos de entrada (por ejempl, la tasa de crimen) tiene diferentes escalas. Por ejemplo, algunos valores son proporciones, que obtienen valores entre 0 y 1; otros tienen valores entre 1 y 12, otros entre 0 y 100...

```
library(keras)

dataset <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-% dataset
```

Vamos a mirar los datos:

```
str(train_data)
```

```
##  num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
```

```
str(test_data)
```

```
##  num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
```

Como puedes ver, tenemos 404 muestras de entrenamiento y 102 muestras de prueba, cada una con 13 características numéricas, como la tasa de crimen per capita, número medio de habitaciones, accesibilidad a carreteras...

Las etiquetas son los valores medios de las casas habitadas, en miles de dolares

```
str(train_targets)
```

```
##  num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
```

Los precios están entre 10.000 y 50.000 $.

3.6.2 Preparando los datos

Sería problemátio insertar en una red neuronal valores que tienen rangos muy diversos. La red puede ser capaz de adaptarse automáticamente a estos datos tan heterogeneos, pero haría el aprendizaje muy difícil. Una práctica muy expandida para tratar con estos datos es hacer una normalización: para cada característica del input (una columna en la matriz input), se extrae la media de las características y se divide por la desviación estándar, así la característica esta sobre 0 y tiene una desviación estándar unitaria. Esto se hace fácilmente en R utilizando la función `scale()`.

```
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)
test_data <- scale(test_data, center = mean, scale = std)
```

Observa que las cantidades utilizadas para normalizar los datos de prueba se computan utilizando los datos de entrenamiento.

3.6.3 Construyendo tu red

Dado que hay muy pocas muestras disponibles, se utilizará una red muy pequeña con dos capas ocultas, cada una con 64 unidades. en general, cuantos menos datos de entrenamiento tengas, menos sobreajuste habrá, y utilizando una red pequeña es una manera de mitigar el sobreajuste.

```
build_model <- function() {
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(train_data)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mse",
    metrics = c("mae")
  )
}
```

La red termina con una única unidad y sin activación (será una capa linear). Esto es la típica setup para una regresión escalar (una regresión donde estás intentando predecir un único valor continuo). Aplicar una función de activación puede restringir el rango que el output puede tomar; por ejemplo si aplicas una función de activación `sigmoide` a la última capa, la red solo puede aprender a predecir valores entre 0 y 1. Aquí, dado que la última capa es linear, la capa es libre de aprender a predecir valores de cualquier rango.

Observa que compilamos la red con la función de pérdida `mse` - *error cuadrático medio*, el cuadrado de la diferencia entre las predicciones y las etiquetas. Esto es una función de pérdida ampliamente utilizada en problemas de regresión.

También estás monitorizando una nueva métrica durante el entrenamiento: *error absoluto medio* (MAE) Es el valor absoluto de la diferencia entre las predicciones y los etiquetas. Por ejemplo,un MAE de 0.5 en este problema podría significar que tus predicciones han caído en 500$ de media.

3.6.4 Validando la aproximación utilizando la validación por K-fold

Para evaluar tu red mientras sigues ajustando sus parámetros (como el número de epochs usadas para entrenamiento), podeis separar los datos en conjunto de entrenamiento y de validación, como vimosvisteis en los

ejemplos anteriores. Dado que tenemos tan pocos puntos de datos, el onjunto de validación sería demasiado pequeño (con unos 100 ejemplos). Como consecuencia, las puntuaciones de validación pueden cambiar mucho dependiendo de que puntos elegimos usar para la validación y cuáles elegimos para el entrenamiento.

Lo mejor que podemos hacer en esta situación es utilizar la validación cruzada *K-fold*. Esto consiste en dividir los datos disponibles en K particiones (tipicamente K = 4 o 5).

```
k <- 4
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)

num_epochs <- 100
all_scores <- c()
for (i in 1:k) {
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()
  model %>% fit(partial_train_data, partial_train_targets,
                epochs = num_epochs, batch_size = 1, verbose = 0)

  results <- model %>% evaluate(val_data, val_targets, verbose = 0)
  all_scores <- c(all_scores, results[[2]])
}
```

```
## processing fold # 1
## processing fold # 2
## processing fold # 3
## processing fold # 4
```

Si ejecutamos esto con `num_epochs = 100` obtenemos los siguientes resultados

```
all_scores
```

```
## [1] 2.626514 2.537258 2.198102 2.578079
```

```
mean(all_scores)
```

```
## [1] 2.484988
```

Las diferentes ejecuciones muestran diferentes puntuaciones de validación, desde 2,1 a 2,8. La media (2,4) es una métrica más creíble que cualquier otra puntuación - esta es la cuesión de la validación cruzada.

Vamos a intentar entrenar una red un poco más grande: 500 epochs. Para mantener un record de cómo de bien el modelo se comporta en cada epoch, modificaremos el loop de entrenamiento para guardar la puntuación de validación.

```
num_epochs <- 500
all_mae_histories <- NULL
for (i in 1:k){
  cat("processing fold #", i, "\n")

  val_indices <- which(folds == i, arr.ind = TRUE)
  val_data <- train_data[val_indices,]
  val_targets <- train_targets[val_indices]

  partial_train_data <- train_data[-val_indices,]
  partial_train_targets <- train_targets[-val_indices]

  model <- build_model()

  history <- model %>% fit(
    partial_train_data, partial_train_targets,
    validation_data = list(val_data, val_targets),
    epochs = num_epochs, batch_size = 1, verbose = 0
    )
  mae_history <- history$metrics$val_mae
  all_mae_histories <- rbind(all_mae_histories, mae_history)
}
```

```
## processing fold # 1
## processing fold # 2
## processing fold # 3
## processing fold # 4
```

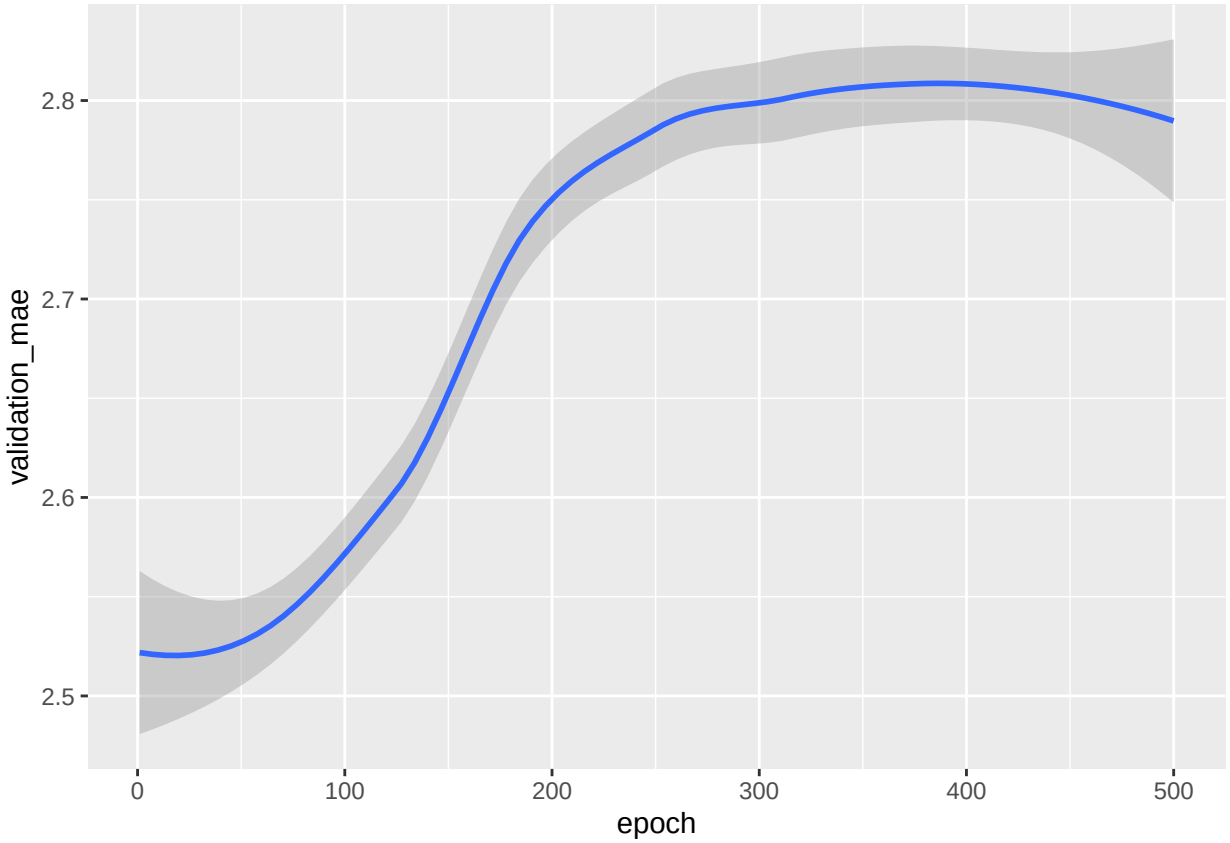
Ahora puedes computar la media de MAE por iteración

```
average_mae_history <- data.frame(
  epoch = seq(1:ncol(all_mae_histories)),
  validation_mae = apply(all_mae_histories, 2, mean)
)
```

Vamos a representarlo

```
library(ggplot2)
ggplot(average_mae_history, aes(x=epoch, y = validation_mae)) + geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



De acuerdo con esta gráfica, la validación MAE deja de mejorar significantemente después de 46 iteraciones. Tras este punto, empieza a sobreajustarse.

```
model <- build_model()

model %>% fit(train_data, train_targets,
              epochs = 80, batch_size = 16, verbose = 0)

result <- model %>% evaluate(test_data, test_targets)
```

```
result
```

```
##      loss       mae
## 17.865377  2.624015
```